MAPQATOR : A System for Efficient Annotation of Map Query Datasets

Mahir Labib Dihan[1], **Mohammed Eunus Ali**[1], **Md Rizwan Parvez**[2]
[1]Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology (BUET)
[2]Qatar Computing Research Institute (QCRI)
{mahirlabibdihan, mohammed.eunus.ali}@gmail.com, mparvez@hbku.edu.qa

Abstract

Mapping and navigation services like Google Maps, Apple Maps, Openstreet Maps, are essential for accessing various location-based data, yet they often struggle to handle natural language geospatial queries. Recent advancements in Large Language Models (LLMs) show promise in question answering (QA), but creating reliable geospatial QA datasets from map services remains challenging. We introduce MAPQATOR, a web application that streamlines the creation of reproducible, traceable map-based QA datasets. With its plug-and-play architecture, MAPQATOR enables seamless integration with any maps API, allowing users to gather and visualize data from diverse sources with minimal setup. By caching API responses, the platform ensures consistent ground truth, enhancing the reliability of the data even as real-world information evolves. MAPQATOR centralizes data retrieval, annotation, and visualization within a single platform, offering a unique opportunity to evaluate the current state of LLM-based geospatial reasoning while advancing their capabilities for improved geospatial understanding. Evaluation metrics show that, MAPQATOR speeds up the annotation process by at least 30 times compared to manual methods, underscoring its potential for developing geospatial resources, such as complex map reasoning datasets. The website is live at: https://mapqator.github.io/ and a demo video is available at: https://youtu.be/7_aV9Wmhs6Q.

.

1 Introduction

In recent years, mapping and navigation services have transformed the way individuals access and interact with location-based information. Platforms such as Google Maps[1] and Apple Maps[2] have become essential tools, providing users with features like route planning, nearby points of interest (POIs), and contextual data, including reviews and operating hours. However, while these services offer extensive geospatial data, they often struggle with understanding and processing natural language queries. This limitation hampers their effectiveness for users seeking to obtain specific information or engage in more complex question-answering (QA) tasks.

Recent advancements in multi-agent and tool-augmented large language models (LLMs) demonstrate significant promise for complex reasoning, decision-making, and generation tasks across various application domains, including those that interact with domain-specific tools such as maps (Liu et al., 2023; Qin et al., 2023). Notable tasks like WebArena (Zhou et al., 2023) and VisualWebArena (Koh et al., 2024) have been proposed with practical real-life applications involving map usage. However, despite these developments, there remains no straightforward method for LLMs to access the vast databases of map services. Currently, there are no dedicated platforms designed to efficiently annotate language-map reasoning tasks, such as question answering. This gap leads to significant challenges in creating reliable datasets for training and evaluating LLMs for geospatial reasoning tasks, as many existing approaches rely on manual data collection methods that result in inconsistencies, lack of reproducibility, and difficulties in tracking the origins of information.

To address these issues, we present MAPQATOR, a web application designed to streamline the creation of map-based QA datasets. MapQaTor empowers researchers to seamlessly integrate with any map API in a plug-and-play manner, enabling them to gather, visualize, and annotate geospatial data with minimal setup. By caching API responses, the platform ensures a consistent ground truth, which enhances the reliability of the datasets, even as

[1]https://mapsplatform.google.com/
[2]https://www.apple.com/au/maps/

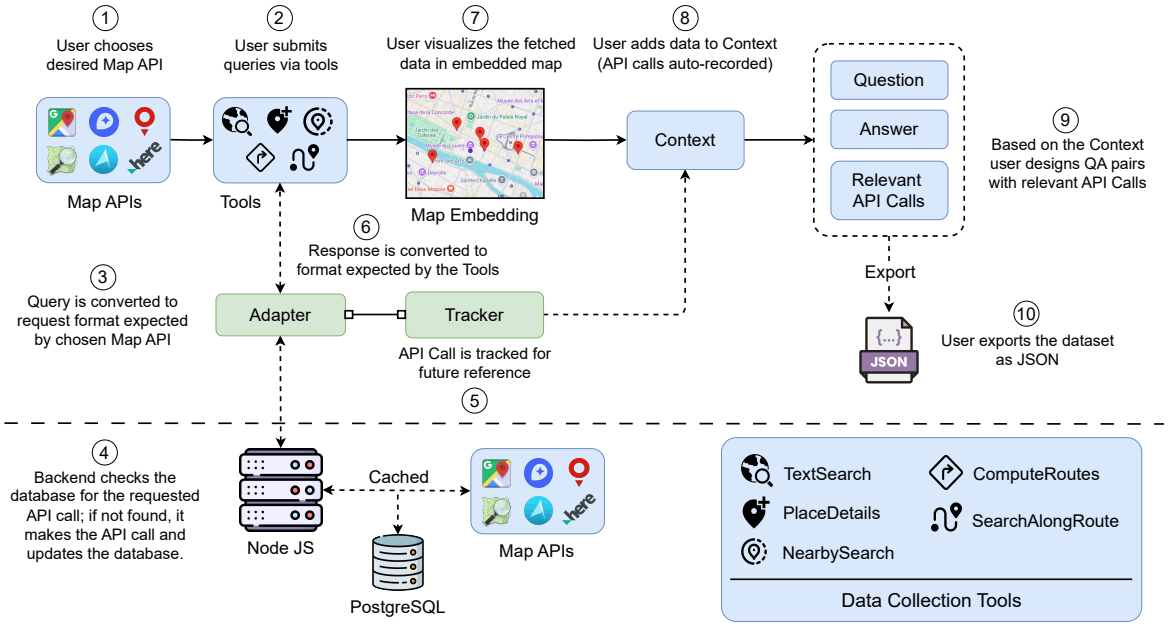

Figure 1: Overview of the annotation and visualization process of MAPQATOR .

real-world information evolves over time.

In summary, in this demo we have made the following key contributions:

1. We propose a novel framework, MAPQATOR, first of its kind, which simplifies the creation of reproducible map-based QA datasets and reduces reliance on manual data collection through seamless integration with any map API (e.g., Google Maps, Apple Maps, Openstreet maps, etc.) in a plug-and-play manner.
2. We provide visualization tools that facilitate better understanding and annotation of geospatial information.
3. We implement caching of API responses to ensure a consistent ground truth, enhancing the reliability of QA tasks over time.
4. We evaluate MAPQATOR to estimate its usefulness and efficiency.

We have published the code on GitHub[3] under the Apache 2 license.

2 MAPQATOR

MAPQATOR is a web-based platform designed to streamline the creation of reproducible, map-based question-answering (QA) datasets that can be used to evaluate and advance the geospatial reasoning abilities of large language models (LLMs). By integrating with any map API in a plug-and-play manner, MAPQATOR enables users to efficiently gather, annotate, and visualize map data to support complex, location-based QA tasks. This section details the main components of the platform, its architecture, and its unique features. Figure 1 outlines the proposed framework, which enables users to interact with map APIs by submitting queries, processing responses, and visualizing data. The framework allows users to design question-answer pairs and export the dataset in JSON format for downstream applications. The whole working flow is show using ten key steps. Example of step-by-step annotation process is provided in Appendix E.

2.1 Plug-and-Play Architecture

A core design principle of MAPQATOR is its adaptable architecture, allowing seamless integration with multiple map APIs. The platform incorporates an adapter layer that standardizes API requests and responses, making it compatible with any map service (e.g., Google Maps, Openstreet Maps, etc.). For each map API, MAPQATOR employs a custom adapter (Details in A) that translates incoming requests into the format required by the target API and then converts the API's responses back to a standardized format that the platform can process. This plug-and-play architecture allows for easy expansion, enabling users to incorporate different map providers with minimal configuration.

[3] https://github.com/mapqator/

Tool	API Provider	API Endpoint
Text Search	Google Maps	Text Search (New) | Places API
		Text Search | Places API
	OpenStreetMap	Search queries | Nominatim
	Mapbox	Suggest | Search Box API
	TomTom	Point of Interest Search
	HERE	Discover | Geocoding and Search
	Azure Maps	Search - Get Search Fuzzy
Place Details	Google Maps	Place Details (New) | Places API
	OpenStreetMap	Place details | Nominatim
	Mapbox	Retrieve | Search Box API
	TomTom	Place by ID
	HERE	Lookup | Geocoding and Search
	Azure Maps	Search - Get Search Fuzzy
Nearby Search	Google Maps	Nearby Search (New) | Places API
	TomTom	Nearby Search
Compute Routes	Google Maps	Get a route | Routes API
	OpenStreetMap	Routing API | GraphHopper
	TomTom	Calculate Route
Search Along Route	Google Maps	Search along route
	TomTom	Along Search Route

Table 1: Current API Support for Data Collection Tools in MAPQATOR

2.2 Caching Mechanism

To enhance efficiency and ensure consistency, MAPQATOR caches API responses in a PostgreSQL database. This caching mechanism not only reduces the number of repeated API calls, saving time and resources, but also ensures that the ground truth data remains consistent over time. By storing API responses in a JSONB column, the platform enables efficient retrieval of previously fetched data, which is particularly valuable when querying the same locations or routes multiple times. The caching mechanism thereby contributes to faster performance and more reliable QA dataset creation, even as real-world map data continues to evolve.

2.3 Visualization Tools

For visualizing geospatial data, MAPQATOR utilizes the Google Maps JavaScript API[4] to display places and routes directly on an embedded map. Users can view places as markers and visualize route paths, offering an interactive and intuitive experience for exploring map data. This visualization capability supports users in understanding spatial relationships, which is essential for designing complex map-based questions and ensuring that annotations accurately reflect the geospatial context. For further details, please refer to Appendix B.

[4] https://developers.google.com/maps/documentation/javascript/overview

2.4 Data Collection Tools

In MAPQATOR , we have implemented five essential tools/features that leverage the capabilities of various map APIs for efficient data collection. The current API support for these tools is summarized in Table 1, highlighting the specific APIs integrated with each tool and illustrating how easily new APIs can be incorporated to meet diverse user needs. For demonstration purposes, we have ensured that each tool integrates with at least two different APIs.

2.4.1 Text Search

This feature allows users to perform searches for specific places or types of locations using natural language queries as shown in Figure 2. By utilizing the search capabilities inherent to most map APIs, users can quickly retrieve relevant results, enhancing the efficiency of data collection.

2.4.2 Place Details

With this tool, users can obtain detailed information about selected locations (see Figure 3). By accessing attributes such as name, address, ratings,

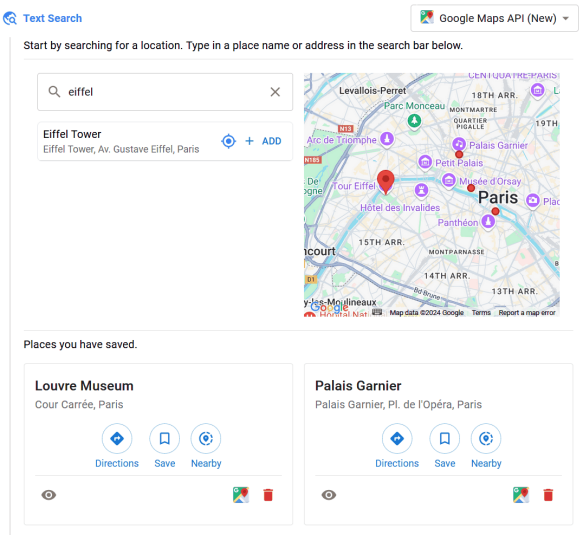

Figure 2: Search for a place

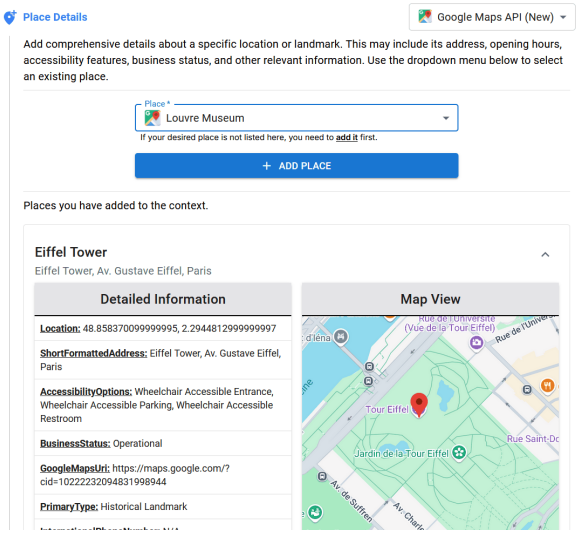

Figure 3: Fetch full details of a place

opening hours, and reviews, users can enrich their understanding of each place, which is crucial for creating accurate QA pairs that capture essential context.

2.4.3 Nearby Search

This feature empowers users to discover points of interest in close proximity to a specified location (see Figure 4). By leveraging the nearby search functionality commonly available in map APIs, users can explore related locations, providing valuable context for their QA pairs.

2.4.4 Compute Routes

This feature finds alternate directions/routes from one place to another for different travel modes (see Figure 5). Users can view different routes and

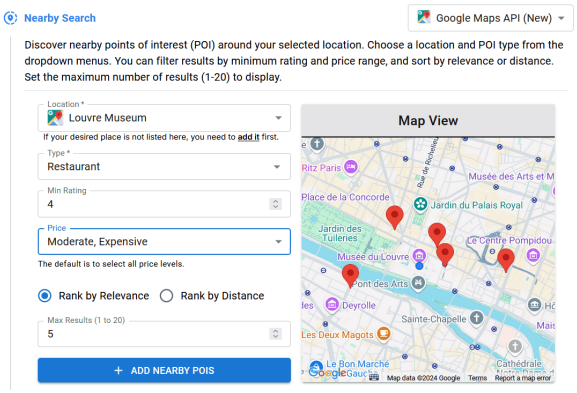

Figure 4: Search Nearby Places

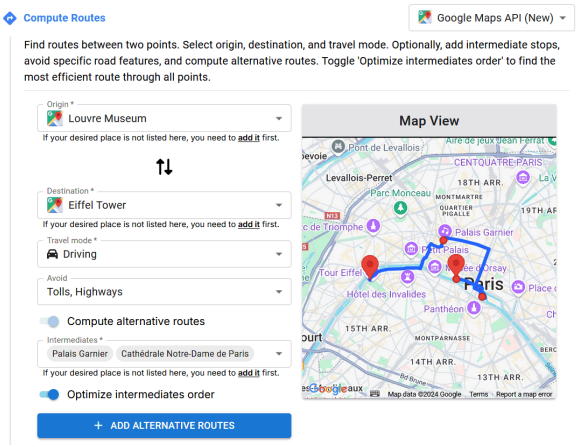

Figure 5: Find routes between places

step-by-step navigation. This functionality is particularly useful for developing questions related to navigation. For multi-stop routes, intermediate waypoints can be optimized, which is useful for trip-related queries.

2.4.5 Search Along Route

This feature allows users to identify points of interest along a defined route (see Figure 6). Users can input their desired path and receive a list of relevant locations, enhancing the contextual richness of their QA pairs by exploring how different places relate to one another during travel.

After collecting necessary data, users need to store it as Context, which represents a collection of API calls and their responses (Details in Appendix D.1).

2.5 Question Design and Annotation

The Question Design and Annotation feature in MAPQATOR facilitates the creation and management of questions, enhancing the process of generating high-quality QA pairs (see Figure 7). It supports four question formats: Yes/No, Single

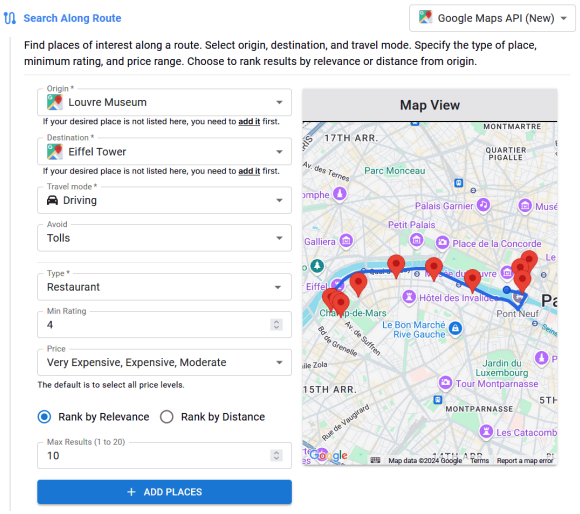

Figure 6: Search places along a route

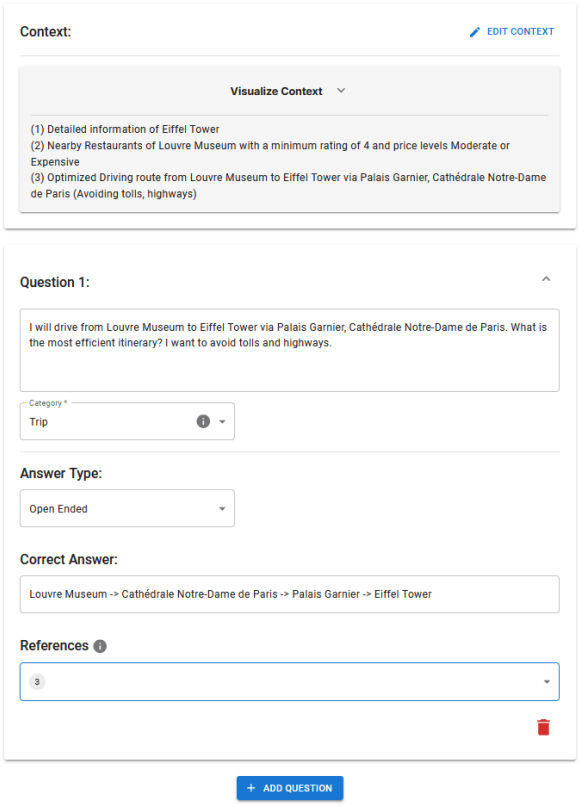

Figure 7: Create a question, provide options, and annotate the correct answer.

Choice, Multiple Choice, and Open Ended, allowing users to select the format that best suits their needs (Details in Appendix D.3).

Users can assign categories to each question, enabling better organization and retrieval based on thematic relevance. Also, while writing question/answer user will get Place Name suggestions to ensure consistency and uniqueness (Details in Appendix D.2).

Since multiple questions can be designed based on a single Context and not all the information (e.g., API responses) in the Context is relevant to each QA pair, users need to specify which pieces of information in the Context are needed to answer each question (Details in Appendix D.4). This process is crucial, as it allows the system to link relevant API calls to the corresponding QA pairs seamlessly. This integration streamlines the question-creation process while ensuring that each QA pair is backed by verifiable data sourced from the underlying map APIs. By maintaining a clear connection between questions and API calls, MAPQATOR enhances the reliability and traceability of the generated datasets, making it easier for researchers to validate and reproduce results.

2.6 Tracking and Traceability

MAPQATOR includes a tracking feature that logs all API calls made during the annotation process. As users interact with the platform, every API call—whether retrieving nearby points of interest, route information, or other geospatial data—is automatically logged. This logging process ensures that each interaction is documented, capturing the request details, response data, and the corresponding timestamps. This feature allows users to associate specific API calls with each question-answer pair, ensuring that the origin of each answer is traceable to a specific data source. Additionally, this could provide valuable insights for training API retrieval models (Qin et al., 2023). This level of traceability is crucial for creating reproducible datasets, as it allows users to review and verify the data used for each question and answer, reinforcing the reliability of the QA dataset.

2.7 Technology Stack

The following section details the technology stack employed in the development and operation of our system.

Frontend: The user interface is built using Next.js, a React framework that provides server-side rendering and static site generation.

Backend: Our server-side logic is implemented in Node.js with Express.js, providing a robust and scalable environment for handling API requests and managing business logic.

Database: We utilize PostgreSQL for its relational database capabilities, supporting complex queries and ensuring data integrity.

2.8 Application Scenarios

MAPQATOR is primarily designed to support the creation of a comprehensive and everyday-use geospatial question answering (QA) dataset, with the potential to benchmark large language models (LLMs), assess their capabilities, and identify areas for improvement in geospatial reasoning tasks. Using MAPQATOR's plug-and-play architecture, users have the flexibility to evaluate the richness and capabilities of any available map service across a wide range of location-based services.

3 Experiments and Evaluation

Evaluating MAPQATOR presented unique challenges due to the lack of any existing system with similar capabilities for creating and managing map-based question-answering datasets. As a first-of-its-kind platform, direct comparisons to other tools were infeasible. Instead, we designed an experiment to measure the efficiency of MAPQATOR in comparison to manual data collection methods currently used by researchers (see Table 2).

To simulate the manual method, two human participants were tasked with retrieving map data from Google Maps using common search functions and then manually copying the necessary textual information to ensure traceability. This included gathering information on Place Details, Nearby Search, Route Computation, and Search Along Route. We then used MAPQATOR to retrieve the same data, noting the time taken for each method across these four features. The results demonstrate a significant improvement in data retrieval speed, with MAPQATOR requiring at least 30 times less time than the manual approach.

Task	MAPQATOR	Manual
Place Details	10.17 sec	487 sec
Nearby Search	12.50 sec	456 sec
Compute Routes	14 sec	516.5 sec
Search Along Route	15.66 sec	476 sec

Table 2: Quantitative comparison between our system and manual methods

4 Related Works

Recent research has highlighted the potential of map data in mimicking real-world planning tasks through various tools (Xie et al., 2024; Zheng et al., 2024). Additionally, studies emphasize the significance of caching API call results to establish a stable database for evaluation purposes (Guo et al., 2024; Xie et al., 2024). The development of web-based platforms for integrating geospatial data has also been explored, focusing on streamlining data collection and enhancing the usability of geospatial information for research and development (Choimeun et al., 2010; Cai and Hovy, 2010; Zheng et al., 2014).

While tool-calling datasets like ToolBench (Qin et al., 2023) and APIBank (Li et al., 2023) include location-based tasks, their data collection processes lack traceability and reproducibility. This limitation highlights a significant gap in the current landscape: the development of datasets for geospatial question answering is still in its infancy. Existing resources often fail to capture the rich contextual information provided by modern map services. Therefore, there is a pressing need for innovative approaches that effectively leverage the extensive data available from map services to create comprehensive geospatial QA datasets.

5 Conclusion

In this paper, we have proposed a novel framework, MAPQATOR, first of its kind, to automatically fetch rich contextual map service data, which forms the basis to develop language-map benchmark datasets for evaluating SoTA LLMs. Our developed web platform simplifies data collection for users by offering precise spatial information, user-friendly search, and efficient data retrieval by using Map APIs. Our application also enables user to create geospatial questionnaire. Experimental evaluation suggests that MAPQATOR is highly effective in developing geospatial question answer datasets. We believe this approach introduces a new task in geospatial question answering, which has the potential to open a new research direction in the intersection of language models and spatial reasoning. Furthermore, our framework can be adapted to other domains.

Limitations

Despite the capabilities of MAPQATOR, several limitations should be acknowledged. The platform utilizes several paid map APIs, which may incur costs based on usage. During the current public demonstration period, users can explore its features without immediate expenses; however, in the

long run, users will need to host the platform independently and integrate their own API keys to access paid functionalities. This requirement necessitates an understanding of the pricing structures associated with the various APIs, potentially impacting accessibility for some users. The platform's functionality is heavily dependent on the availability and stability of external map APIs, meaning that any changes or deprecations in these APIs could negatively impact performance. The quality of the generated QA pairs is contingent on the retrieved data and users' ability to formulate meaningful questions, which can introduce variability in dataset quality. The evaluation metrics used might not encompass all aspects of usability, possibly overlooking qualitative user feedback. In addition to map service data, other platforms such as Trip Advisor can also be a rich source of additional context for geospatial queries.

References

Congxing Cai and Eduard Hovy. 2010. Summarizing textual information about locations in a geo-spatial information display system. In *Proceedings of the NAACL HLT 2010 Demonstration Session*, pages 5–8.

S Choimeun, N Phumejaya, S Pomnakchim, and Chantana Chantrapornchai. 2010. Tool for collecting spatial data with google maps api. In *U-and E-Service, Science and Technology: International Conference UNESST 2010, Held as Part of the Future Generation Information Technology Conference, FGIT 2010, Jeju Island, Korea, December 13-15, 2010. Proceedings*, pages 107–113. Springer.

Zhicheng Guo, Sijie Cheng, Hao Wang, Shihao Liang, Yujia Qin, Peng Li, Zhiyuan Liu, Maosong Sun, and Yang Liu. 2024. Stabletoolbench: Towards stable large-scale benchmarking on tool learning of large language models. *arXiv preprint arXiv:2403.07714*.

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Chong Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Ruslan Salakhutdinov, and Daniel Fried. 2024. Visualwebarena: Evaluating multimodal agents on realistic visual web tasks. *arXiv preprint arXiv:2401.13649*.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. Api-bank: A comprehensive benchmark for tool-augmented llms. *arXiv preprint arXiv:2304.08244*.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. 2023. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688*.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*.

Jian Xie, Kai Zhang, Jiangjie Chen, Tinghui Zhu, Renze Lou, Yuandong Tian, Yanghua Xiao, and Yu Su. 2024. Travelplanner: A benchmark for real-world planning with language agents. *arXiv preprint arXiv:2402.01622*.

Huaixiu Steven Zheng, Swaroop Mishra, Hugh Zhang, Xinyun Chen, Minmin Chen, Azade Nova, Le Hou, Heng-Tze Cheng, Quoc V Le, Ed H Chi, et al. 2024. Natural plan: Benchmarking llms on natural language planning. *arXiv preprint arXiv:2406.04520*.

Yuxin Zheng, Zhifeng Bao, Lidan Shou, and Anthony KH Tung. 2014. Mesa: A map service to support fuzzy type-ahead search over geo-textual data. *Proceedings of the VLDB Endowment*, 7(13):1545–1548.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Yonatan Bisk, Daniel Fried, Uri Alon, et al. 2023. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*.

A Adapter Layer

The Adapter Layer in MAPQATOR provides base classes for various tools, including 'TextSearch', 'PlaceDetails', 'NearbySearch', 'ComputeRoutes', and 'SearchAlongRoute'. To integrate a new API under any tool, developers need to extend the specific base class and implement two methods: 'convertRequest' and 'convertResponse'. These methods standardize the request and response formats to align with MAPQATOR's requirements, ensuring consistent interaction across APIs. Figure 8 demonstrates an implementation example of extending the 'TextSearch' base class to support the TomTom API. Note that "key:TOMTOM_API_KEY" is used as a placeholder, which will be replaced in the backend using environment variables to prevent exposing the API key on the frontend.

```
class TomTomApi extends TextSearch {
  constructor() {
    super();
    this.family = "tomtom";
  }

  convertRequest = (query) => {
    return {
      url: "https://api.tomtom.com/
      search/2/poiSearch/" + query + ".
      json",
      method: "GET",
      params: {
        key: "key:TOMTOM_API_KEY",
        limit: 5,
        language: "en-US",
      },
    };
  };

  convertResponse = (data) => {
    const places = data.results.map((
    place) => ({
      id: place.id,
      displayName: {
        text: place.poi.name,
      },
      shortFormattedAddress: place.
      address.freeformAddress,
      location: {
        latitude: place.position.lat,
        longitude: place.position.lon,
      },
    }));
    return { places };
  };
}
```

Figure 8: Implementing the TomTom API Adapter for Text Search in MAPQATOR

B Visualization Tools

For visualizing geospatial data, MAPQATOR utilizes the Google Maps JavaScript API to display places (see Figure 10) and routes (see Figure 11) directly on an embedded map. Markers are used in the map to represent places. It just needs the latitude and longitude of the place. On the other hand, routes are actually encoded polylines[5]. But embeded map expects a set of coordinates along the path to visualize the route. So, we need to first decode the encoded polyline and convert it to a set of latitude and longitude. The decoding algorithm is shown in Figure 9.

```
function decodePolyline(polylineStr) {
  let index = 0, lat = 0, lng = 0;
  const coordinates = [];
  const changes = { latitude: 0,
  longitude: 0 };

  while (index < polylineStr.length) {
    for (const unit of ["latitude", "
    longitude"]) {
      let shift = 0;
      let result = 0;

      while (index < polylineStr.length)
       {
        let byte = polylineStr.
        charCodeAt(index++) - 63;
        result |= (byte & 0x1f)<<shift;
        shift += 5;
        if (byte < 0x20) break;
      }

      changes[unit] = result & 1 ? ~(
      result >> 1) : result >> 1;
    }

    lat += changes.latitude;
    lng += changes.longitude;

    coordinates.push({ lat: lat / 1e5,
    lng: lng / 1e5 });
  }

  return coordinates;
}
```

Figure 9: Polyline Decoding Algorithm

C Exclusion of Temporal Variations in Routing APIs

To ensure reproducibility in the dataset, we have implemented specific measures to eliminate temporal variations in routing responses. These measures are

[5] https://developers.google.com/maps/documentation/utilities/polylinealgorithm

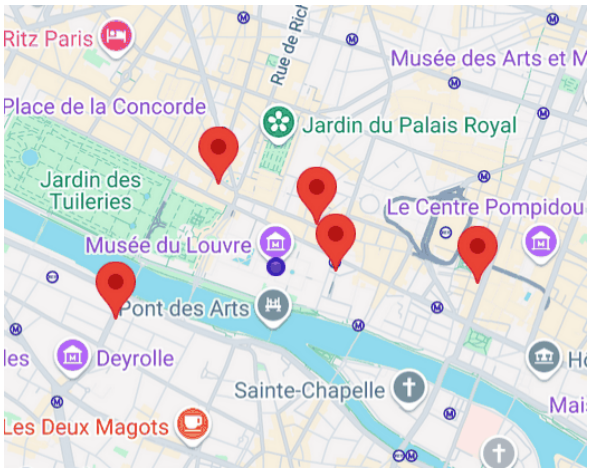

Figure 10: Set of markers indicating different places

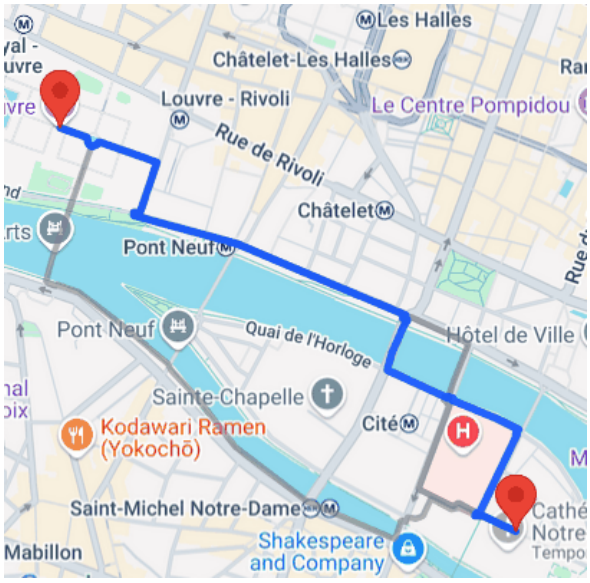

Figure 11: Visualizing routes between places

essential for maintaining consistency across identical queries, regardless of when they are made.

Traffic Awareness Setting: In MAPQATOR, routing APIs are set to "TRAFFIC_UNAWARE," meaning no real-time traffic data is used in generating routes and travel times. Real-time traffic data would introduce variability in travel times, as identical requests made at different times would yield different results based on current traffic conditions. By disabling traffic awareness, we ensure that each query consistently returns the same travel time, independent of real-time conditions, thereby enhancing dataset reproducibility.

Exclusion of Transit Mode: We have also excluded the "TRANSIT" travel mode from 'ComputeRoutes' and 'SearchAlongRoutes' tools. Transit routes, which include public transportation options like buses and trains, are heavily influenced by the specific time a request is made due to schedules, service frequency, and operational hours. Including transit mode would introduce further temporal variability, as the same query could yield different results depending on schedule changes, service frequency, and operational hours. This would undermine the consistency of the dataset, as identical queries could yield different travel times and routes.

Benefits of These Measures: By setting routes to be traffic-unaware and omitting transit mode, we ensure that:

- Responses remain consistent: Identical queries yield the same routes and travel times, regardless of request timing.
- Benchmarking is simplified: Model evaluations can focus purely on spatial and routing reasoning without accounting for real-time variations.
- Reproducibility is maintained: The dataset provides a stable baseline for testing and comparing model performance over time.

These adjustments support our goal of creating a reproducible benchmark dataset, free from the temporal dependencies that could otherwise complicate consistent evaluation of geospatial reasoning tasks.

D Question Design and Annotation

D.1 Context Preview

During data collection process, users need to save the necessary fetched information, which we will refer to as Context. Context consists of a collection of API calls and their responses. After creating a context, annotators will proceed to create questions. They can view the created context in two mode.

1. **Summarized Context:** A textual summarized view as a list of fetched information. One liner for each information (see Figure 12).
2. **Visual Context:** In visual context, full details of each information in the context is shown with corresponding Map View ((see Figure 12).

D.2 Place Name Suggestion

Annotators gather different places using TextSearch tool. At the time of writing question,

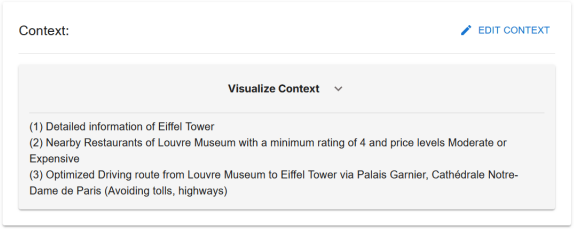

Figure 12: Summarized Context

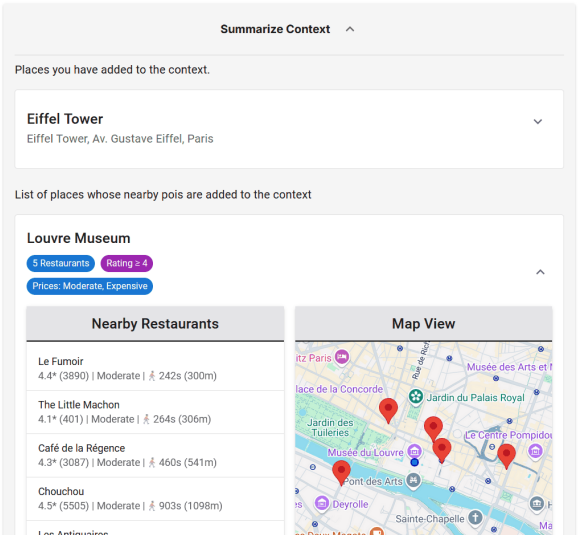

Figure 13: Visual Context

on pressing '@' annotator will get the option to choose from available place names. On pressing enter, the selected name will be entered on the cursor position. This ensures the consistency of place names between context and questions in a hassle free way. This is also applicable in answer field.

Figure 14: Suggesting available places from the context

D.3 Answer Formats

We incorporate four distinct answer formats, which provide flexibility in how users interact with the system. These formats are crucial for creating a diverse set of training and evaluation data for large language models (LLMs). By supporting a range of response types, the dataset captures various ways that models must handle user queries and generate appropriate answers. The four answer formats are as follows:

1. **Open Ended:** The Open Ended format allows for unrestricted, free-text responses, enabling models to generate detailed and contextually rich answers. This format tests the model's ability to understand and generate responses that may require creativity, explanation, or reasoning. It is particularly useful for questions that demand nuanced answers or elaboration.

Figure 15: Answer format: Open Ended

2. **Yes/No:** The Yes/No format tests the model's ability to provide binary, definitive answers. This format is valuable for questions that require a clear-cut response, such as confirming or denying information. The model's performance in this format reflects its ability to handle factual questions with straightforward answers.

Figure 16: Answer format: Yes/No

3. **Multiple Choice:** The Multiple Choice format allows the model to choose from a predefined set of possible answers, potentially selecting more than one option. This format challenges the model to identify the most relevant responses from a set of alternatives and is useful for assessing its ability to discern multiple correct answers or filter out incorrect options.

4. **Single Choice:** Similar to Multiple Choice, but with the restriction that only one option

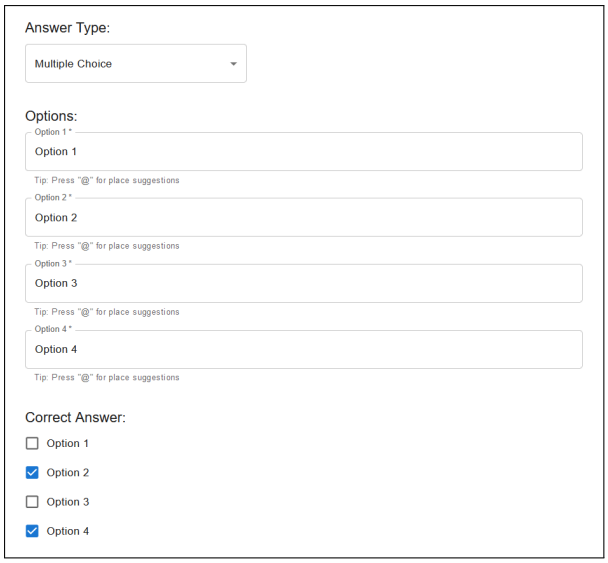

Figure 17: Answer format: Multiple Choice

can be selected. This format is used to evaluate the model's ability to determine the most appropriate single answer from a set of alternatives. It tests the model's decision-making and accuracy in selecting the correct response.

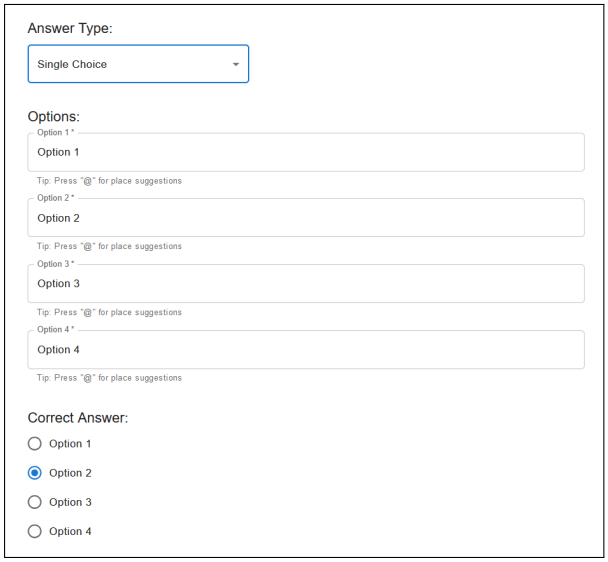

Figure 18: Answer format: Single Choice

D.4 Reference Field

After creating a QA pair, annotator need to select which of the informations in the context is actually relevant to this pair (see Figure 19). We are focusing on the informations fetched through PlaceDetails, NearbySearch, ComputeRoutes and SearchAlongRoute tool. This step is necessary to create a well-grounded and traceable dataset. This ensures that all the necessary information to answer a question is available in the context. Under the hood, corresponding api calls (Relevant API Calls) to each information are linked with the QA pair (see Fig. 35).

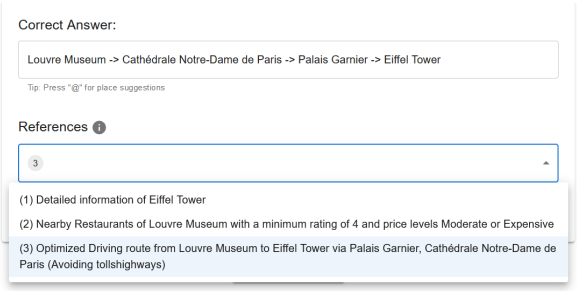

Figure 19: Choosing relevant informations to provide the correct answer.

E Example Annotation

On entering MAPQATOR, we will see 3 major steps (see Fig. 20). First we need to design a context, which contains all the necessary information to design question-answer (QA) pairs. Then we will create QA pairs. And finally we can review and save the dataset.

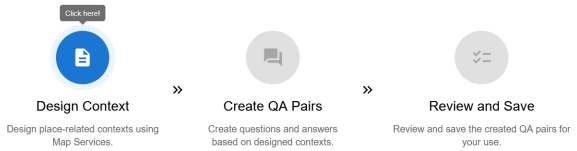

Figure 20: Major steps of MAPQATOR

As described in section 2.4, we can design context using 5 data collection tools (see Fig. 21).

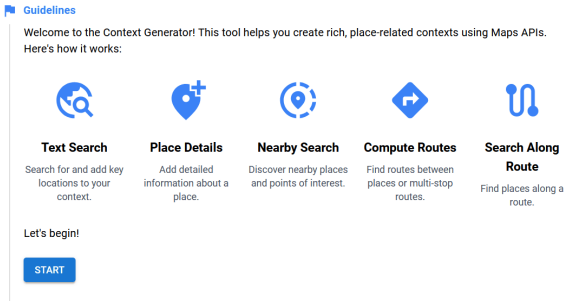

Figure 21: Overview of data collection tools integrated into MAPQATOR, showcasing essential functionalities.

Now let's start designing a context. Let's say we want to create questions on Louvre museum and Eiffel tower. For that we first need to search them

using TextSearch tool and add them to context (see Fig. 22).

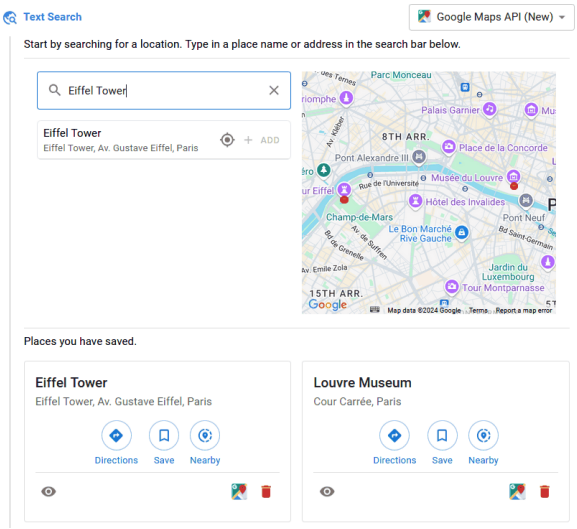

Figure 22: Example use of TextSearch tool

Next, we can search for the nearby restaurants of Eiffel Tower (see Fig. 23). Once we submit query, nearby places of Eiffel Tower will be added to the context and we can see the list of places with some key details (see Fig. 24).

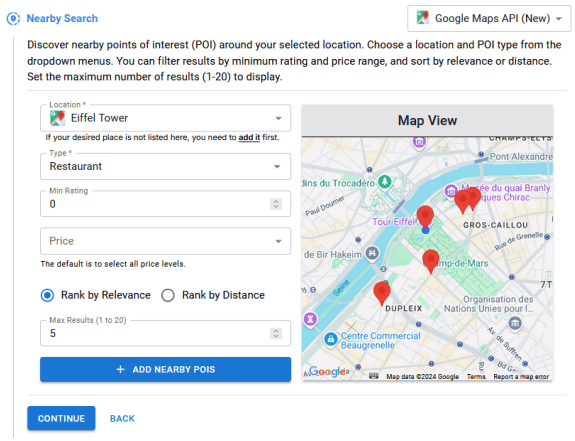

Figure 23: Example use of NearbySearch tool

We can also search for available routes from Louvre Museum to Eiffel tower using ComputeRoutes tool (see Fig. 25). After submitting query, all the available routes with distance, duration and step-by-step navigation will be added to the context (see Fig. 26).

Thus we can finish designing context, and move forward to create questions based on the context. We can start with an example question "Fastest time to go from Louvre Museum to Eiffel Tower by car?" and assign it to "Routing" category (see Fig. 27).

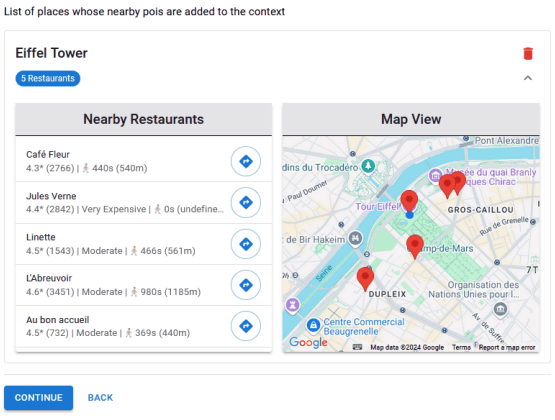

Figure 24: Nearby restaurants of Eiffel Tower

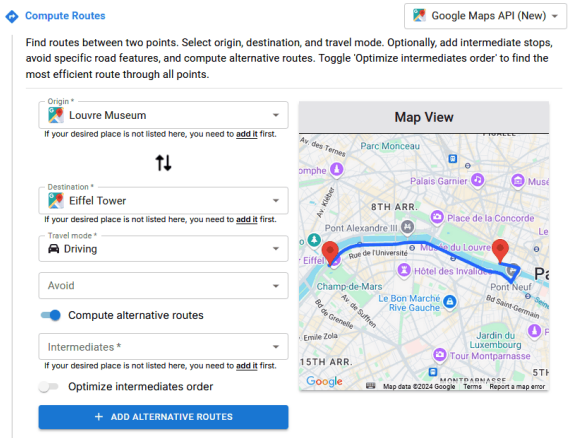

Figure 25: Example use of ComputeRoutes tool

Then we need to provide answer to the question. For our example question, we can get the answer from the context, which is "15 mins" (see Fig. 26). We can choose the answer type and provide the ground truth (see Fig. 28).

Upto this point, only the annotator knows which information in the context is used to derive the ground truth. To maintain traceability we need to link the QA pair with relevant information in the context. In this example, only the routing information from Louvre Museum to Eiffel Tower is relevant. Nearby Restaurants of Eiffel Tower is irrelevant. So, we will select "Driving route from Louvre Museum to Eiffel Tower" in "References" field (see Fig. 29).

Similarly, we can add a second question "Nearby restaurant of Eiffel Tower with over 2000 user ratings and at least 4.5* rating" (see Fig. 30), provide answer (see Fig. 31) and link relevant informations (see Fig. 32).

Finally, we can review the context and QA pairs (see Fig. 33) and download the dataset in JSON

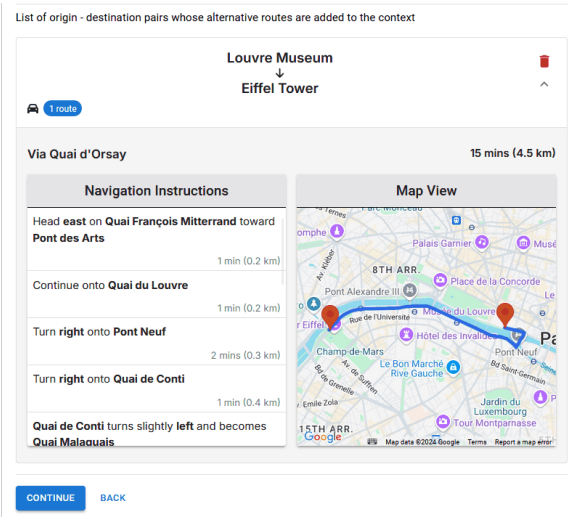

Figure 26: Available routes from Louvre museum to Eiffel tower by car

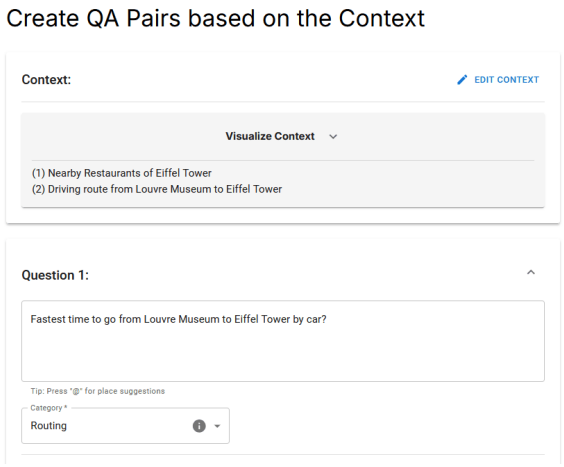

Figure 27: Example question creation (1)

format (see Fig. 34). Downloaded dataset format is shown in Figure 35.

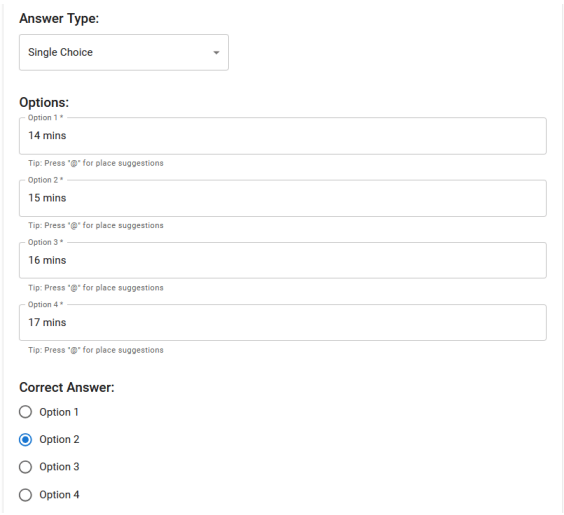

Figure 28: Example answer creation (1)

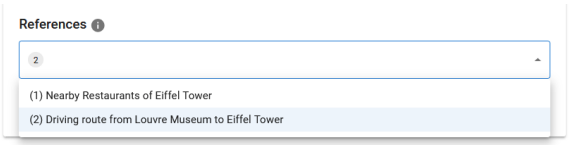

Figure 29: Choosing relevant informations (1)

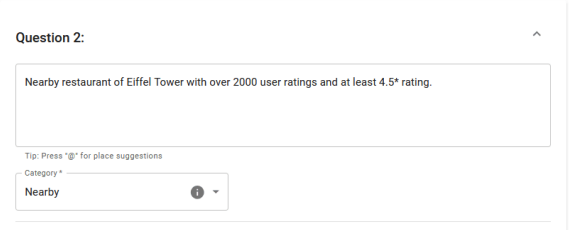

Figure 30: Example question creation (2)

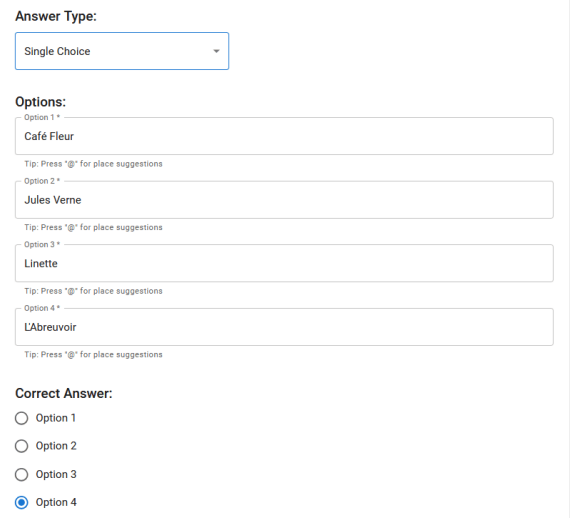

Figure 31: Example answer creation (2)

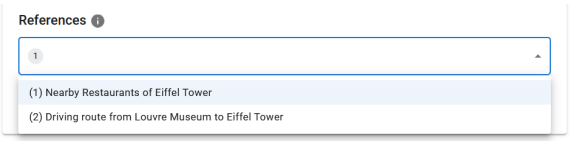

Figure 32: Choosing relevant informations (2)

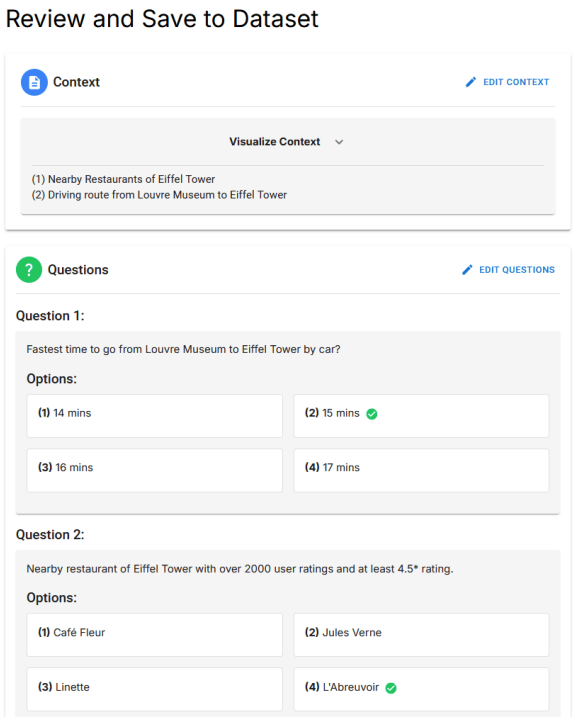

Figure 33: Review context and QA pairs

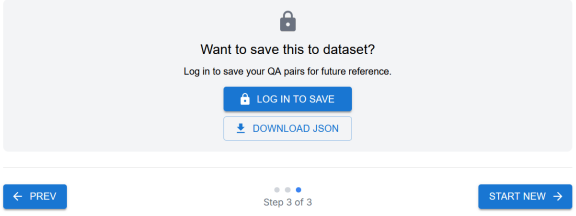

Figure 34: Download dataset as JSON


```
{
  "questions": [
    {
      "title": "Fastest time to go from
      Louvre Museum to Eiffel Tower by
      car?",
      "answer": {
        "type": "single-choice",
        "options": [
          "14 mins","15 mins",
          "16 mins","17 mins"
        ],
        "correct": 1
      },
      "classification": "routing",
      "relevant_api_calls": [ 98006 ]
    },
    {
      "title": "Nearby restaurant of
      Eiffel Tower with over 2000 user
      ratings and at least 4.5* rating
      .",
      "answer": {
        "type": "single-choice",
        "options": [
          "Cafe Fleur","Jules Verne",
          "Linette","L'Abreuvoir"
        ],
        "correct": 3
      },
      "classification": "nearby",
      "relevant_api_calls": [ 12492 ]
    }
  ],
  "api_call_logs": [
    {
      "url": "https://routes.googleapis.
      com/directions/v2:computeRoutes",
      "method": "POST",
      "headers": { .... },
      "data": {
        "origin": { .... },
        "destination": { .... },
        "travelMode": "DRIVE",
        ....
      },
      "uuid": 98006,
      "result": { .... }
    },
    {
      "url": "https://places.googleapis.
      com/v1/places:searchText",
      "method": "POST",
      "headers": { .... },
      "data": {
        "locationBias": { .... },
        "includedType": "restaurant",
        ....
      },
      "uuid": 12492,
      "result": { .... }
    },
    { .... }
  ],
}
```


Figure 35: Example dataset where each QA pair is linked with relevant api calls