RIDEWISE: Enhancing Biking Journeys with LLM-Assisted Navigation

Nicholas Gardner
Department of Computer Science
Rochester Institute of Technology

I. INTRODUCTION

NAVIGATION apps are ubiquitous and are very effective at guiding vehicular traffic through constantly changing roads. However, they were not designed for all vehicles. Their main audience is cars. And this preference is reflected in available features. Easily accessible are options to avoid toll roads or freeways. But no such parallel exists for avoiding roads without separate bike lanes. Cars, and the roads they travel on, are almost impervious to most weather conditions. But for cyclists, sections of routes can quickly become impassable due to inclement weather conditions. And yet these sections will continue to be recommended by navigation apps because they are designed for a vehicle that is not impacted by anything except the most extreme of weather conditions.

It is important to note that navigation apps designed for bicycling do exist. Apps like BikeMap and Komoot are popular alternatives to traditional navigational giants due to their cyclist-specific optimizations. Such advantages include being able to take into account the user's fitness level and difficulty of routes, crowd-sourcing route issues specific to bicycles, supplying other user's previous routes, and providing more detailed information about surface types. However, these apps too fall short. They too follow the paradigm of set-and-forget navigation where the user finishes the same route that they began.

In practice, it can be common for weather or street conditions to require a route modification during the journey. Another issue is that these apps rely on free alternatives to Google Maps which are less accurate and updated less frequently. This fact represents an additional barrier to entry: potential users must learn how to interact with a new mapping technology. According to various sources [1], Google Maps has somewhere between a 60 and 80% market share. This is a monstrous amount of control over one market, and represents the greatest problem faced by many of these specialized apps: going through the effort of finding and learning a new navigation system is too much for many casual riders.

This paper presents RideWise: a mobile-optimized web app that allows the user to modify navigational choices via a text-based interface. This interface is connected to a *Large Language Model* (LLM) that parses the request and triggers the various changes that are necessary to facilitate it. Maps and geocoding are implemented using the Google Maps API, ensuring locations are accurate and up-to-date as well as presenting a familiar interface to users. Geoapify is used for navigation due to its improved flexibility and customizability to . All supported modifications are aimed at the needs of bicycling – including (but not limited to) requesting preferred surface/path type, routing through various (potentially scenic) waypoints, and avoiding problematic sections of routes. Additionally, RideWise is designed to assist with another common use case of bicycling – leisure. RideWise is connected to the TrailAPI [2] to allow users to request trails by location, difficulty, type, etc.

Evaluation was conducted using a small dataset of human-created routes and modification requests. ChatGPT [3] and Llama 2 [4] are compared...

II. RELATED WORK

The primary technical challenge in this work is engineering a system of connected LLM instances that parse the user's request and execute the needed operations to make it happen. *Attention is All You Need* [5] began the current explosion in complexity and effectiveness of LLMs by describing the transformer – a model architecture that can learn complex embeddings for words. However, it was not until ChatGPT [3] was released by OpenAI that these models were seen as capable of more than just coherent generation and vector encoding. The most recent version of ChatGPT [3] is capable of completing the tasks necessary for RideWise with minimal engineering. However, it is not free and available to all. Every API call costs some fraction of a cent (depending on the number of tokens in the input and output). This requirement is inhibitive for open scientific advancement, and therefore other options were explored.

In the open-source space, there is currently only one option that is vastly superior to all others. Llama 2 [4] is a family of LLMs, released by Meta, that aimed to match premium options in effectiveness and safety. These models range in size from 7 billion to 70 billion parameters, and were pre-trained on 2 trillion tokens. The *chat* model variants, one of which is used in this work, were additionally fine-tuned using *Reinforcement Learning Human Feedback* (RLHF). As shown in [4], Llama 2 70B outperforms all other state-of-the-art LLMs on six different common academic benchmarks. Additionally, the Open LLM Leaderboard [6], published by HuggingFace, shows Llama 2 model variants in essentially every position at the top of the leaderboard.

Besides choice of model, it is imperative to use best practice prompting techniques in order to yield the best results for

LLMs. In the original ChatGPT paper [3], it was shown that LLMs can approach the accuracy of state-of-the-art fine-tuned language models through a process known as *few-shot learning*. Essentially, the model is given a few examples of input and correct output at the time of inference as conditioning (with no weight updates allowed). However, this process is not fool-proof. Zhao et al. [7] found that LLMs are highly sensitive to changes in the prompt. A change as simple as flipping the order of examples in the prompt could change the accuracy from state-of-the-art to no better than random chance. Further, LLMs are biased towards answers that appear frequently in the prompt, answers that appear near the end of the prompt, and tokens that are common in their pre-training dataset. To resolve these issues, a few different solutions have been proposed. Zhao et al. [7] showed improvement through *contextual calibration* – determining prompt bias by supplying a neutral input (such as "N/A") and modifying model output after calculation. White et al. [8] provides a series of prompt patterns that lead to more consistent and effective prompts. And in Pitis et al. [9], it was shown that prompt ensembles are more effective than few-shot prompting alone. This work uses a combination of these methods to enhance the capability of a pre-trained model alone.

III. System Architecture

There are three primary components that form RideWise: client, Flask server, and LLM. The user interacts with the client, selecting a start and destination as well as zero or more change requests. Currently implemented is the ability to add waypoints to the route, to avoid locations or roads, and to indicate the style of route that is preferred. These requests are routed through the Flask server, where they are initially split before being dispatched to the instantiated LLM. From there, the LLM splits each request into discrete items. Next, the LLM decides, for each request, which style of change is desired. These changes are then communicated back to the Flask server, which loads them into a persistent JSON file. On completion of request evaluation, the server conducts the various API querying and routing logic necessary to determine the final route to render. This route is then passed to the client which displays the updated route to the user. Refer to Figure 1 for a visual depiction of the system architecture as well as a different example input flow.

A. Client

The client is the interface that the user interacts with. On opening the web page, the user is greeted with a familiar navigational setup: input boxes for the start and destination, as well as a dynamic map. These boxes are connected to the Google Maps API, allowing for region-biased autocompletions. The dynamic map is rendered by the same API service, ensuring consistent results. Unique to this application, there is an additional input box for submitting text-based route requests. Accompanying this text box are two buttons, one for initiating speech-to-text capabilities, and one for clearing current route modifications. The page is rendered using HTML, with JavaScript to handle interactivity and Bootstrap CSS for styling. When the route is received from the Flask server, it is rendered as GeoJSON on the data layer of the dynamic map.

B. Flask Server

The first task that the Flask server handles is routing requests from the client. When the web page is requested, the home page is returned for rendering. When the user submits their change request, the server first splits the request on the period character ('.') as this is a common separator. Next, these request chunks are sent to the LLM for processing. The first step breaks the chunks up further into discrete units. For instance, "stop at the park and use trails" is split into two separate requests: "stop at the park" and "use trails". The server then sends these units to a different LLM instance that determines what type of change is being requested. "stop at the park" gets translated into "add_waypoints | the park" while "use trails" becomes "prefer_path_type | trails". These processed requests are then used for the Flask server's other major task: map routing.

With no change requests, the client simply needs to request a route from the direction service and render it. This is handled easily by Google Maps API, leading to its use as the original direction service in this work. However, it quickly became clear when implementing change requests that Google Maps API would be insufficient for this task. While Google Maps provides a biking layer for its dynamic map (that shows where trails, bike paths, and other biking features exist), it does not provide such details through the routing API. Additionally, no functionality is included for avoiding roads or locations. For these reasons, it was decided to pivot to the Geoapify Routing API.

Handling changes is conducted in a multi-step process, with different error-catching mechanisms meant to account for any model hallucinating that occurs. First, the JSON file containing change requests is fetched. Next, each requested waypoint is geocoded (to convert from a place name to a set of latitude/longitude coordinates) before being added to the request string. All geocoding attempts conducted by RideWise are region-biased to the bounds of the route, preferring nearby locations to reduce unwanted inclusions of same-named locations. Additionally, if the waypoint is unable to be geocoded, or the geocoded location is farther away from the destination than a specified threshold distance, then the waypoint is discarded. Next, the desired path type is determined (a simple "in" check to determine if trails/roads/city streets are preferred).

Finally, avoid locations are handled. This requires some additional logic as Geoapify only avoids latitude/longitude locations, which is not applicable out-of-the-box to avoiding roads or paths. A route is first generated using our currently known components – waypoints, path type, and previous avoid locations. Then, each avoid parameter is checked to determine if it is a road or path along the route. For instance, "scottsville rd" would be detected if the route step's name was: "Scottsville Road, Rochester 14623". If a detection occurs, then the start and end coordinates for that route step are determined and an avoid waypoint is placed at each. Assuming no detection

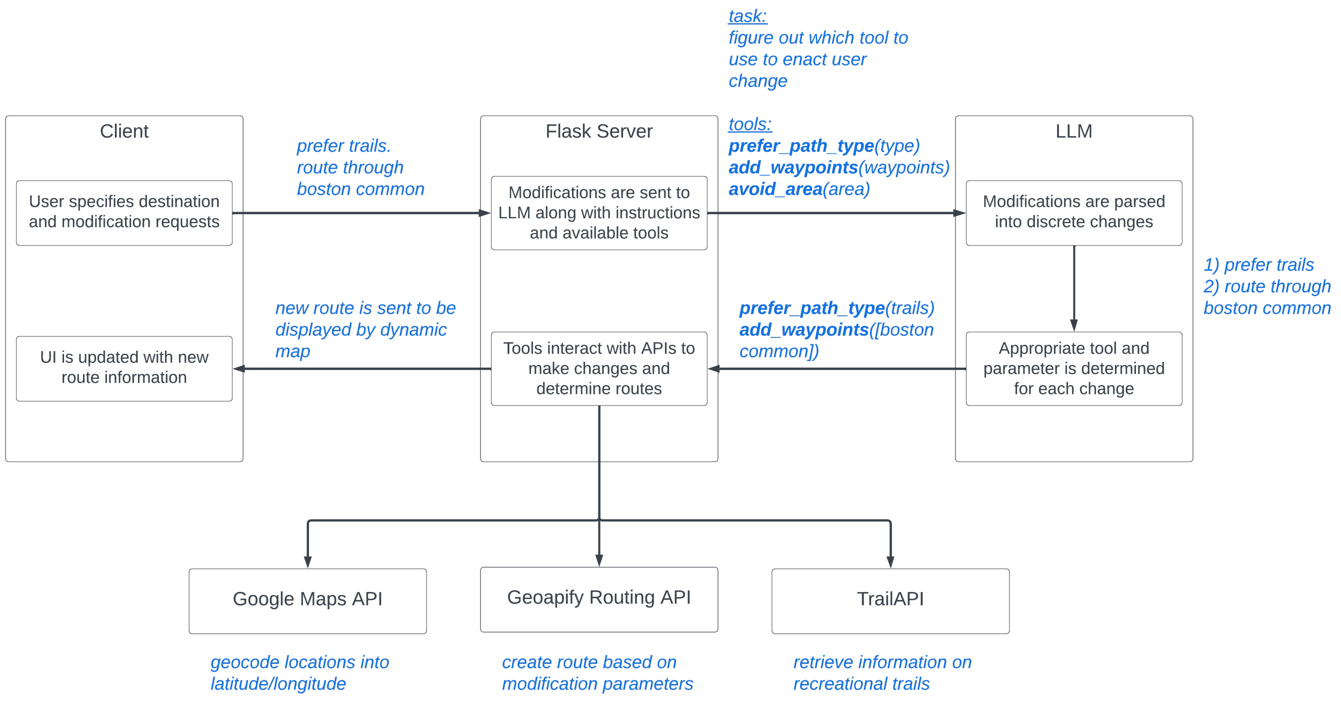
task:
figure out which tool to use to enact user change
Client
User specifies destination and modification requests
UI is updated with new route information
prefer trails. route through boston common
new route is sent to be displayed by dynamic map
Flask Server
Modifications are sent to LLM along with instructions and available tools
Tools interact with APIs to make changes and determine routes
tools:
prefer_path_type(type)
add_waypoints(waypoints)
avoid_area(area)
prefer_path_type(trails)
add_waypoints([boston common])
LLM
Modifications are parsed into discrete changes
Appropriate tool and parameter is determined for each change
1) prefer trails
2) route through boston common
Google Maps API
Geoapify Routing API
TrailAPI
geocode locations into latitude/longitude
create route based on modification parameters
retrieve information on recreational trails

Fig. 1. System architecture of RideWise is shown in black. Accompanying the architecture is a sample input with descriptions at each step of the operations that are conducted in response (shown in blue).

occurs (ruling out road/path), then the location is geocoded to determine if it is a known location. If it is, then it is added to the request string. If it is not, then the avoid location is discarded.

With all of the changes added to the request string, Geoapify is queried again. Returned is a GeoJSON route object that can be rendered by most major mapping platforms. This route is returned to the client, signifying the completion of the server's tasks related to this round of requests.

C. LLM

As discussed earlier, this paper uses Llama 2 [4] as this model is significantly more effective than any other open source model currently [4]. Specifically, a C++ distribution of the 13B parameter version is utilized, and the model weights are quantized to reduce space and inference time [10]. The framework, LangChain, is used to facilitate inference and supply system prompts prior. Additionally, no fine-tuning is performed, as instead this paper explores the capabilities of a few-shot model.

The first approach at this work naively utilized only one request to the LLM. Using ChatGPT [3], it is possible to pass the entire request string, as well as a list of tools, and the correct tool(s) will be selected the vast majority of the time. While Llama 2 is significantly more effective than other open source models, it is not more effective than premium models like ChatGPT. In early iterations, it was found that not only does Llama 2 not interact well with tools, it is entirely incapable of coherently and consistently executing the required selection task. To make the problem simpler, it was decided to break the LLM component into two tasks. The first task is to separate the input into discrete requests, and the second task is to process those discrete requests into function calls and parameters.

An example of the first LLM task is as follows: "use trails and stop at the boston common and the empire state building" should become "use trails | stop at the boston common and the empire state building". This task seems quite simple from a human perspective, but it was found to be quite challenging for Llama 2 in practice (perhaps even more so than task two). In this example, one cannot simply split on the word "and", because it is a valid splitting criteria in one instance and not in another. Additionally, simple verb/noun differentiation may not be possible due to ambiguity in some words (house can be both a noun and a verb). Instead, the LLM must pick up on context clues and determine which clauses are related and which begin a new thought.

The second task is to determine what type of change is being requested (from the list of implemented functionality). "route through the park and the cemetery" needs to become "add_waypoints | the park | the cemetery". The complexity here lies in translating synonyms and related concepts to their corresponding implemented functionality. And as stated earlier, this task was seemingly impossible for Llama 2 when multiple requests were present. However, after splitting requests, the LLM is able to perform adequately most of the time.

As the model is essentially a black box, it is not feasible to explain what is going on under-the-hood. But, there were a few techniques that significantly improved performance across both tasks. The request splitting template (appendix A-A) and function selection template (appendix A-B) are included to add context to this discussion. First, standardizing the system prompt template aided effectiveness in completing each task *and* improved output format consistency. This involved moving away from free-form, dialogue-based prompts and towards more formal and consistent language.

From:

> *For instance, if the user inputs: 'prefer trails', you should output: prefer_path_type('trails').*

To:

> *User: prefer trails*
> *Assistant: prefer_path_type | trails*

Another change that was inspired by literature review was balancing representative examples for each outcome. LLMs are very sensitive to imbalanced samples, and will predict the majority class if the system prompt is imbalanced [7]. In the prompt for task one, this balance is achieved by including an equal number of examples of "split-necessary" and "split-unnecessary" instances. For task two, this is achieved by including an equal number of avoid, waypoint, and path type examples.

Additional improvement was gained by reducing unique place names, where unique place name refers to names like "Mount Hope Cemetery" or "Susan B. Anthony House". When encountering another unique place name during inference, the model tended to select a corresponding unique place name example, rather than selecting by intended purpose.

Outside of prompt engineering, other techniques were explored and documented as part of this study. Prompt ensembles have been shown to be more effective than a single prompt instance [8], motivating an investigation into their use despite their impact on inference speed. Essentially, these ensembles involve setting the temperature of the model (how willing is it to be stochastic and creative) to a higher value and combining a collection of such models into a voting body. More details on this to come as the investigation occurs...

Appendix A
Prompt Templates

A. Request Splitting

You are a helpful assistant who parses text and splits the text into discrete requests. Return ONLY the response to the last user response and nothing more.

User: I want to ride on trails and avoid main street Assistant: I want to ride on trails | avoid main street

User: stop at boston common and the empire state building
Assistant: stop at boston common and the empire state building

User: avoid 42nd ave
Assistant: avoid 42nd ave

User: prefer roads and route through 120 charles rd and the golden gate bridge
Assistant: prefer roads | route through 120 charles rd and the golden gate bridge

User: avoid johnson bridge and 17 Madison St
Assistant: avoid johnson bridge and 17 Madison St

User: pass through the airport and use city streets
Asssistant: pass through the airport | use city streets

User: Avoid the downtown area and stop at the park and the museum.
Assistant: Avoid the downtown area | stop at the park and the museum.

User: stop at the park
Assistant: stop at the park

User: Stay away from the noisy downtown street, stop at the coffee shop and the library.
Assistant: Stay away from the noisy downtown street | stop at the coffee shop and the library.

User: I want to ride on roads
Asssistant: I want to ride on roads

User: don't go on fipson dr and preferred trails
Assistant: don't go on fipson dr | preferred trails

User: skip welter ave
Assistant: skip welter ave

User:

B. Function Selection

You are a helpful assistant who parses text and determines what change the user is requesting. A user will pass in text, which you should parse to determine which change is requested. Return ONLY the response to the last user request and nothing more.

Change options are as follows:
1. avoid_area – add if the user requests to avoid a particular area, or to not take a particular route.
2. add_waypoints – add if the user requests to add additional stops to their route, or if they want to route through a destination.
3. prefer_path_type – add if the user specifies a type of path surface that is preferred.

User: prefer trails
Assistant: prefer_path_type | trails

User: stop at boston common
Assistant: add_waypoints | boston common

User: avoid main street
Assistant: avoid_area | main street

User: route through north ave and city hall
Assistant: add_waypoints | north ave | city hall

User: avoid johnson bridge and 17 Madison St
Assistant: avoid_area | johnson bridge | 17 Madison St

User: I want to ride on roads
Asssistant: prefer_path_type | roads

User:

References

[1] L. Ceci, "Top u.s. mapping apps by reach 2018," Aug 2023. [Online]. Available: https://www.statista.com/statistics/865419/most-popular-us-mapping-apps-ranked-by-reach/

[2] J. Barber, "Introducing trailapi," Sep 2021. [Online]. Available: https://www.singletracks.com/mtb-trails/introducing-trailapi/

[3] T. B. B. et al., "Language models are few-shot learners," 2020.

[4] H. T. et al., "Llama 2: Open foundation and fine-tuned chat models," 2023.

[5] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2023.

[6] E. Beeching, C. Fourrier, N. Habib, S. Han, N. Lambert, N. Rajani, O. Sanseviero, L. Tunstall, and T. Wolf, "Open llm leaderboard," 2023. [Online]. Available: https://huggingface.co/spaces/HuggingFaceH4/open_llm_leaderboard

[7] T. Z. Zhao, E. Wallace, S. Feng, D. Klein, and S. Singh, "Calibrate before use: Improving few-shot performance of language models," 2021.

[8] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, and D. C. Schmidt, "A prompt pattern catalog to enhance prompt engineering with chatgpt," 2023.

[9] S. Pitis, M. R. Zhang, A. Wang, and J. Ba, "Boosted prompt ensembles for large language models," 2023.

[10] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A survey of quantization methods for efficient neural network inference," 2021.