RIDEWISE: LLM-Enhanced Bicycle Navigation

Nicholas Gardner
Department of Computer Science
Rochester Institute of Technology

Abstract – RideWise is a navigational system that uses an LLM interface to allow for text and speech input to modify routes and request trail information. A simple user interface communicates with a Flask server to relay user requests. The server runs an LLM instance and queries multiple times to determine the nature of each user request and extract relevant details. These details are thoroughly-checked for errors before being passed to several APIs that return the modified routes or trail information for display by the user interface. To test the efficacy of this system, a small dataset of real routes and modifications were created manually. Models compared were a single LLama 2 instance, an ensemble of Llama 2 instances, and a ChatGPT-3.5 instance.

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 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 web app that allows the user to modify navigational choices via a text- or speech-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 built-in functionality for avoiding waypoints and preferring path types. 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 rating, difficulty, length, and distance away.

Evaluation was conducted using a small dataset of human-created routes and modification requests. ChatGPT [3], Llama 2 [4], and an ensemble of Llama 2 models are compared to show efficacy of each model/technique, as well as the effectiveness of the system overall.

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 ChatGPT was used only as a reference point during evaluation.

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 aim 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. One possible approach for this work would be to fine-tune the model on a custom dataset. However, creating a sufficiently large and balanced dataset is prohibitively challenging and often leads to overfitting. 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 prompting*. 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). A much smaller dataset of examples is necessary for this technique, massively reducing the development requirements for each LLM task. 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 focuses on template design, task deconstruction (into more manageable pieces), and prompt ensembles to enhance the capability of a pre-trained model.

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. Additionally, users can request to view local trails, and can specify requirements related to rating, difficulty, etc. These requests are routed through the Flask server, where it is first determined whether the request relates to route modification or trail information. If it is a route modification, the request is split on periods before being dispatched to the instantiated LLM. From there, the LLM splits each remaining 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 the 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 route modification system architecture as well as an example input flow. If the request is instead related to trail information, it is passed to the instantiated LLM to extract user requirements. The TrailAPI is queried with these requirements and the results are passed to the client to render as a collection of markers with infoWindows on the map.

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 three buttons: one for initiating speech-to-text capabilities, one for clearing current route modifications, and one for submitting. The page is rendered using HTML, with JavaScript to handle interactivity and Bootstrap CSS for styling. When a route is received from the Flask server, it is rendered as GeoJSON on the data layer of the dynamic map. If instead trail information is received, each trail is rendered as a marker with an attached infoWindow that contains details supplied by the TrailAPI.

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 passes the request to an instantiated LLM to determine if the request is related to route modification or trail information. Depending on this outcome, one of two paths are taken.

1) Route Modification

If the request is related to route modification, the server first splits the request on the period character ('.') as this is a common separator (known exception of middle name characters with periods is handled prior). 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

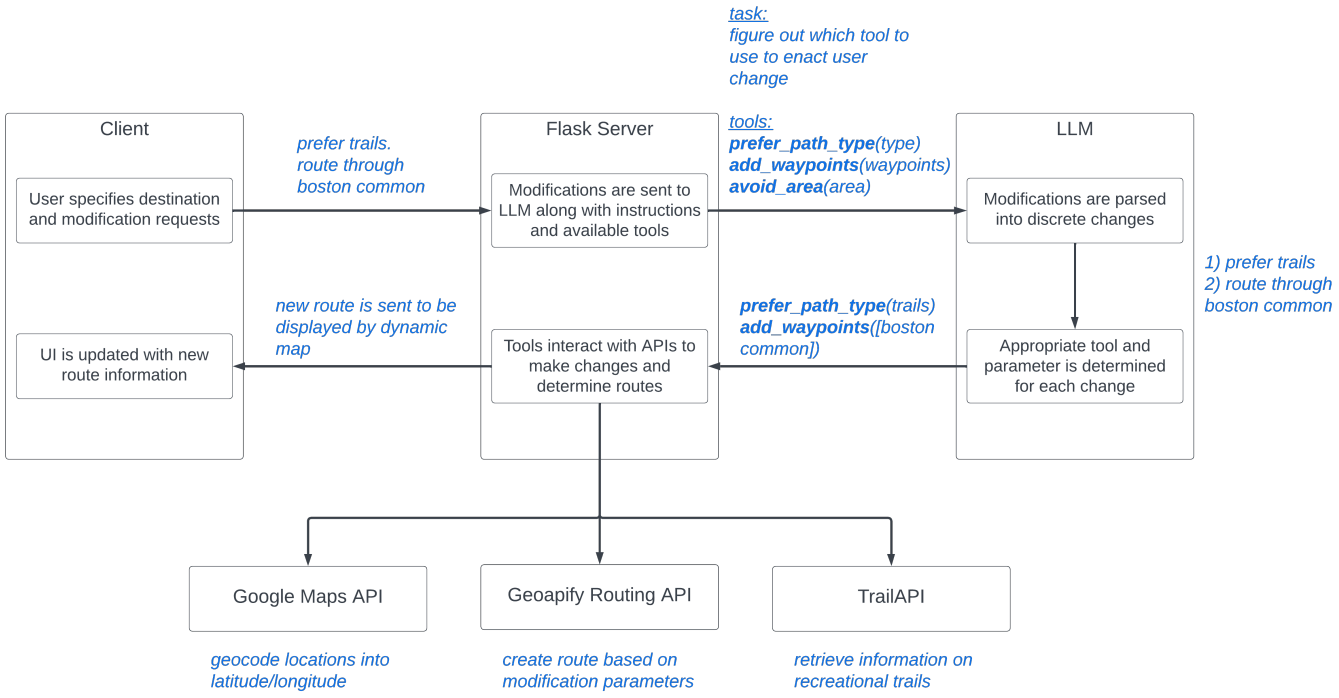

Fig. 1. System architecture of RideWise is shown in black. Accompanying the architecture is a sample input with descriptions at each step of the operations that are conducted in response (shown in blue).

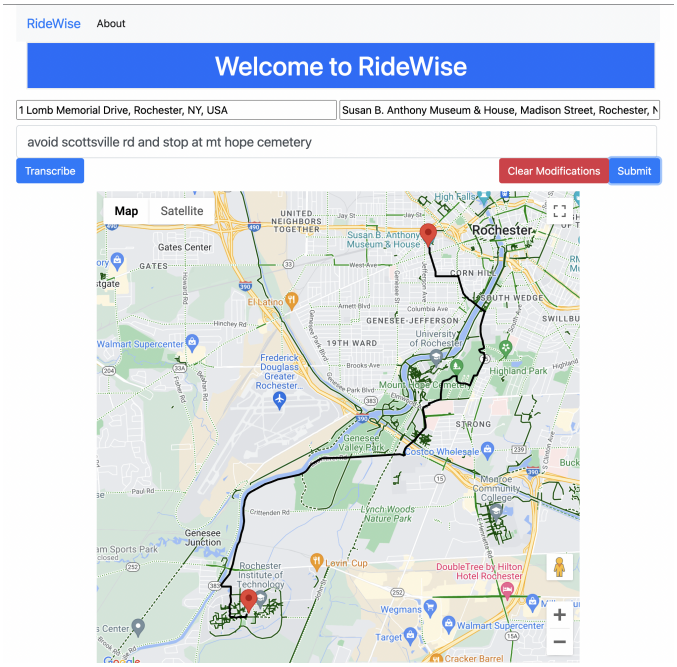

Fig. 2. Example of user interface with modified and rendered route.

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 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". An avoid waypoint is placed at each step where a detection occurs (or every third step if the overlap is sufficiently large). Assuming no detection 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.

2) Trail Information

Alternatively, if the request is related to trail information, it is first passed to the LLM to extract user requirements. Supported requirements include difficulty, distance, rating, and length.

For instance, if the user inputs:

Looking for intermediate trails with rating above 3

The model outputs:

Difficulty(Intermediate) | Rating(> 3)

After necessary requirements are extracted, the TrailAPI is queried for trails nearby the current starting location. At this step, the only restriction that can be applied is radius of search. This returns a collection of JSON objects with information about each located trail. This collection is then filtered to match the user's remaining requirements (difficulty, rating, length). After filtering, the remaining collection is sent to the client where it is rendered as trail markers with attached infoWindows that contain the information provided by the TrailAPI.

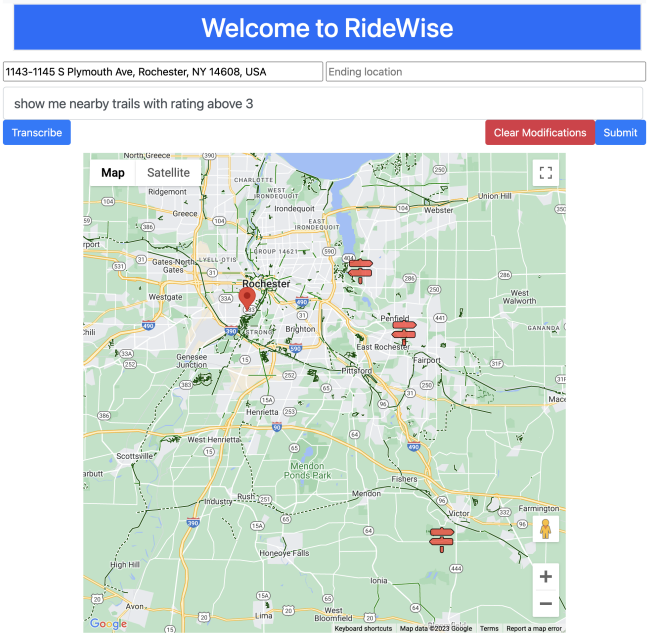

Fig. 3. Discovered trails displayed as markers on the user interface map.

C. LLM

As discussed earlier, Llama 2 [4] is used as this model is significantly more effective than any other open source model currently [4]. Specifically, a C++ distribution of the 13B-chat model 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.

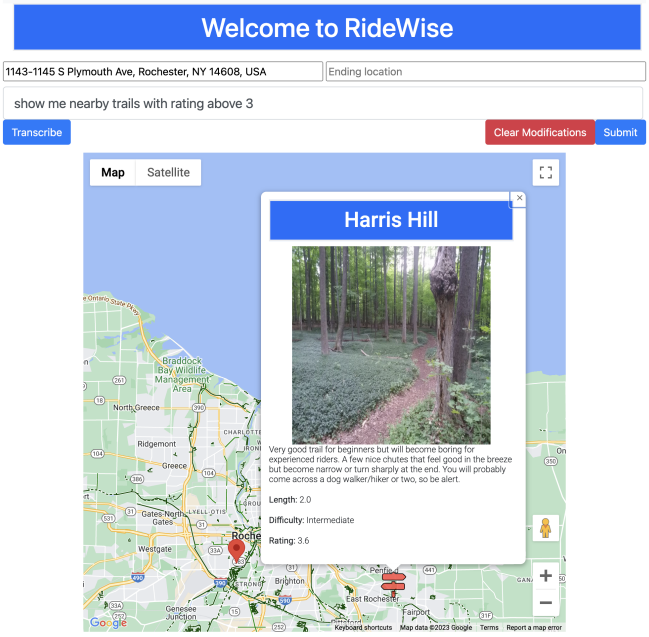

Fig. 4. Selected trail with attached infoWindow containing information gathered from the TrailAPI.

The first approach of 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 multiple tasks. The first task is to determine if the request is related to route modification or trail information. The second task is to separate the input into discrete requests, and the third and fourth tasks are to process those discrete requests (depending on whether it is a route modification or trail information request).

The first task is quite simple in theory. Essentially, the model decides in a one word answer whether the input is intended to modify a route (*Modification*) or get information about local trails (*Trail*). This step is necessary in order to send the request to the right subsequent steps to be processed. However, this task can be a bit tricky at times, as inputs for modification such as "prefer trails" are very similar to some inputs for trail information such as "looking for trails". In practice, the LLM model is quite effective at this task and has seen very few mistakes during testing. See Appendix A-A for the prompt

template for this task.

An example of the second 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 three). 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. See Appendix A-B for the prompt template for this task.

The third task, which occurs for route modifications, 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. See Appendix A-C for the prompt template for this task.

The fourth and final task, which occurs for trail information requests, is to extract user requirements for the retrieved trails. For instance, "routes with rating above 2 and length of 5 miles or less" should become, "Rating($>$ 2) | Length($<$ 5)". As this task is the secondary purpose of this application, it was explored and tested much less thoroughly. However, in small-scale testing, it was reasonably effective at extracting requirements consistently with little hallucination. See Appendix A-D for the prompt template for this task.

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 all tasks. 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 two, this balance is achieved by including an equal number of examples of "split-necessary" and "split-unnecessary" instances. For task three, 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. However, the single largest boost in performance came from fixing a seemingly small error that caused major issues. Originally, the model was passed the system template with the user input appended. This resulted in the input ending with something like the following "User: prefer trails". After some debugging, it was discovered that the model would often tack on additional requirements before moving on to output the model response. For example, "User: prefer trails and stop at the National Museum of History". This would result in the model outputting both the real requirements and the ones that it hallucinated. Simply by appending a newline and "Assistant:" to the end of the model input (communicating that the LLM should begin at the assistant response and not before), errors and hallucinations were decreased to almost zero.

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.

As development and testing for this project was conducted on consumer hardware, it was not feasible to run multiple instances in parallel as this would exceed available GPU memory. Instead, one LLM instance is queried multiple times for each task and the overall output is determined by treating each query response as a vote. As the temperature is greater than zero, each query response is different so a reasonable facsimile of an ensemble is created. For instance, if the "ensemble" is of size 3, then the model is queried 3 times and a list of responses is returned: [[the park, fire station], [fire station, history museum], [the park, government building]]. Given this array, the server would choose to return [the park, fire station] as the overall response, as these options occur in over half of the ensemble responses. The fundamental thinking behind this approach is that the model will hallucinate more (as temperature is increased) but each response will hallucinate in different directions and the overlap will contain the items that are actually desired.

IV. Experiment

To evaluate the effectiveness of this system, a dataset was manually-created. 10 *multi* routes were created that incorporate each of the three route modification options and are realistic, feasible routes. The following is an example of one of these routes:

Start:

> *725 Averill Ave, Rochester, NY 14607, USA*

End:

715 W Main St, Rochester, NY 14611, USA

Modificaton:

> *I want to ride on roads and route through the Strong Museum of Play and the Susan B. Anthony Museum & House. Skip E Broad st*

Five of these routes are located in Rochester, New York, and each of the remaining five are located in different cities across the United States. A small amount of data augmentation was performed additionally by passing each of these routes to ChatGPT-3.5 and asking it to paraphrase and re-order. This doubled the number of *multi* routes to 20. Additionally, in order to simulate potentially more realistic situations, these routes were broken up into each of their 3 components (avoid, prefer, waypoint) and stored as 60 *single* routes. In order to allow for fast development iteration and evaluation to check results, the assessment system was automated.

First, the output of the LLM is compared against the desired output for the route. If there are no mistakes, the test is marked as *correct*. If all desired elements are included but some additional elements are also added (hallucination), the test is marked as *include*. For *single* tests, if there are elements in one of the categories that is not being tested, the test is marked as *extra*. Finally, if necessary elements are missed, the test is marked as *miss*.

Next, a route is determined from the generated modifications and this route is checked to see if it matches the desired modifications. If the route includes any paths that were intended to be avoided, is missing any waypoints, or the path type is wrong, then the route modification is considered a *miss*. For *multi* tests, this means that if any of the components are missing, the entire route test is considered failed.

This separation of tests between the LLM output and the route modification allows for independent analysis of the LLM implementation versus the functional error-handling done by the server. Additionally, due to time constraints and the secondary nature of the trail information implementation, it was decided to save evaluation of trail information requests for future work.

V. Results

The most important evaluation for this application is whether the route is successfully modified according to the user's specifications. Figures 5 and 6 show error percentages for *single* and *multi* tests respectively. Displayed on each plot are results for the single model and ChatGPT (shown as dashed lines), and ensemble performances for ensembles of size three, five, and seven models. Ensemble performances are tested with varying temperature levels.

For the *single* test evaluation, performance is excellent for most models tested. Error percentage for ChatGPT is 5%, and is 8.3% for the single Llama 2 model instance. Performance for the ensembles varies across model temperature, but generally floats around the performance of the single model until temperature exceeds 0.5. Important to note here as well, several of the failures come from an inability of the geocoder to properly geocode certain waypoints. For instance, Mt. Hope Cemetery in Rochester is particularly troublesome and leads to two route failures in both *single* and *multi* tests. While a more pristine testbed might have excluded troublesome waypoints, it was decided that including these is a more representative evaluation of real-world performance (where some waypoints will have issues with geocoding). Additionally, this test does not seem to motivate using an ensemble. Some ensemble temperatures showed better performances than the single model, but the very small potential performance gain is outweighed by the inherent variance in results and increased inference time.

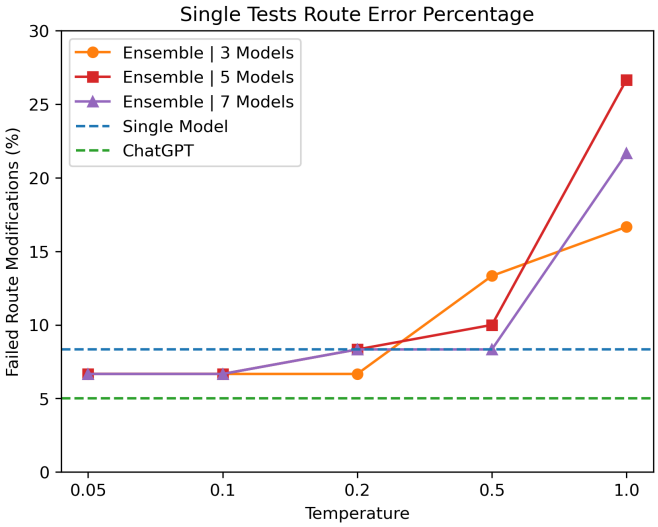

Fig. 5. Evaluation of route modification success for tests that only assess one modification type. Note here that the scale of the y-axis is different than for the *multi* test evaluation.

The *multi* test evaluation for route modification tells a somewhat different story. Much more of the system is being tested here. The input request must be properly split into its component parts and *each* part must be selected correctly and reflected in the end route for success to be declared. Here the single model implementation saw a (total route) success percentage of 75%, while ChatGPT was able to reach a success percentage of 85%. This is considerably worse than the *single* test evaluation, and reflects the complexity of the problem. Additionally, ensemble performance is considerably worse here. The seven LLM ensemble outperforms the single model for one temperature, but is worse at other temperatures. Here it seems that the inherent variance and hallucinations incurred by increasing the temperature makes a challenging task even more difficult.

While the output of the overall system is the most important aspect to evaluate, it is also valuable to understand the effectiveness of the LLM implementation individually. The next set of figures, 7 and 8, show evaluations for the output of the LLM processes. While results were collected for three, five, and seven model ensembles, only the five model ensemble results are shown here as this size of ensemble was reasonably representative of the group. Shown in dashed lines on each plot is the performance of the single Llama model, with ChatGPT not shown (having performed slightly better than the single model in every category). For the *single* tests, the number of

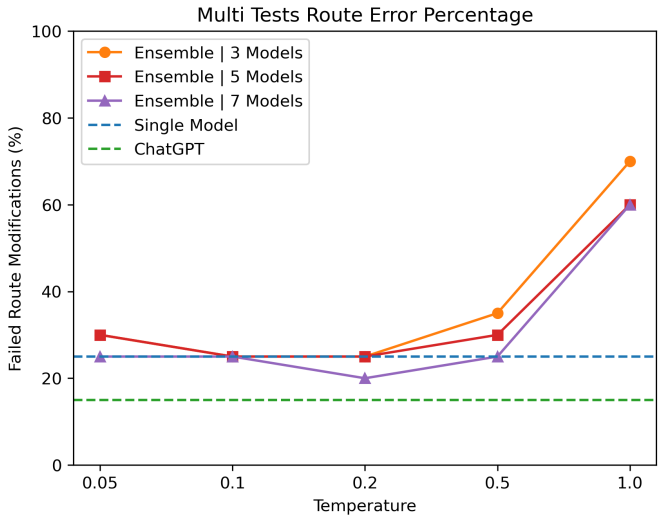

Fig. 6. Evaluation of route modification success for tests that assess all three modification types.

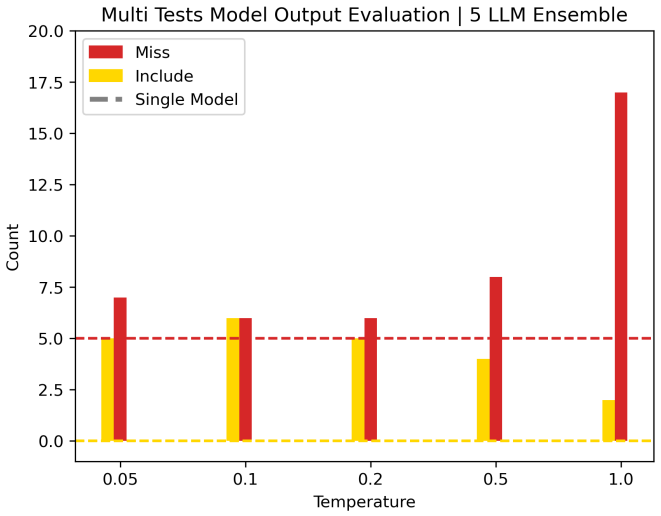

Fig. 8. Evaluation of LLM output congruency for tests that assess all three modification types. Displayed here are results for the five model ensemble, with single model results marked as corresponding dashed lines for comparison.

misses is greater than the single model for all temperatures, and the number of *extra* is similar. However, the big difference between the ensemble and the single model is the number of *includes*. The *includes* represent occurrences of hallucination and are zero for the single model while reaching a count of six or higher for the ensemble.

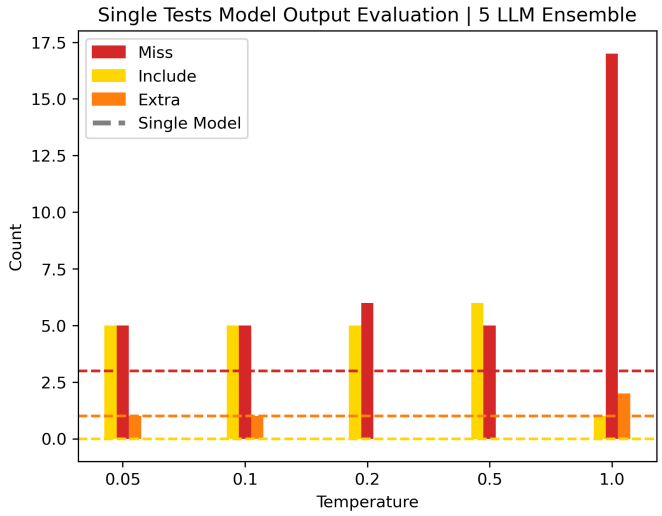

Fig. 7. Evaluation of LLM output congruency for tests that only assess one modification type. Displayed here are results for the five model ensemble, with single model results marked as corresponding dashed lines for comparison.

For the *multi* tests, a similar result is seen to the *single* tests. *Misses* are quite similar to the single model (for temperatures less than 0.5), but i*ncludes* are higher. Note here that the number of tests is 20, so each *miss*/*include* is considerably more damaging overall than for the *single* tests.

These results would seem to imply that the single model should have had considerably higher route modification success than the ensembles, but this was not the case. As seen in Figures 5 and 6, the ensembles saw similar and sometimes higher success rate. This is likely due to the success of the error-checking code implemented to defend against hallucinations, ensuring that waypoints can be geocoded and are not too far away. Additionally, what the automatic evaluation considers *misses* (any output that doesn't match the desired output) may still have been sufficient to succeed at rendering the route.

An important aspect to understand additionally is the speed of inference. Depending on the complexity of the input, an LLM instance needs to conduct inference 3-5 times. As these tests were conducted on consumer-level hardware, each inference took approximately between 2 and 2.5 seconds to complete. This means that the end-to-end system takes between 6 and 12.5 seconds to complete any given route modification request. If an ensemble is involved, then this end-to-end estimate is multiplied by the number of models in the ensemble. While the speed of execution is less important for a research effort like this, a potential future application utilizing this work would need to address this computation speed issue for usability. A simple solution would be to use a commercial LLM option, where inference is conducted on hardware optimized for such a purpose.

VI. Future Work

As this work is primarily a proof-of-concept, there exists a reasonable amount of future work to complete before this system could be used in production. The most important future work would be to add the infrastructure necessary to deploy this application beyond local hosting. The most important issue to tackle here would be how to host the LLM computation. Most consumer hardware (especially phones, which would be the likely devices to use for such an application) would struggle to handle large models like the one used in this project. Until hardware catches up, the most economical choice would be to use OpenAI's servers by querying ChatGPT. Each user

would simply put their OpenAI key into the application prior to use, ensuring billing is directed to them.

Beyond that, it would be necessary to further optimize the UI for the likely intended target, phones. Current choices were made for the development environment, but are not optimal for the devices that navigation most often occurs on. Additionally, current navigation apps most often break the navigation into pieces that are shown step-by-step to the user. This functionality is not currently implemented in RideWise but would be necessary prior to actual use. Due to time constraints and its secondary nature to this project, the trail information system was not evaluated. Future work on this project could develop and conduct such an evaluation. Finally, development could be conducted to support additional route modifications as necessary.

VII. Conclusion

In this project, a proof-of-concept system was created that incorporates an LLM interface into a standard navigation system. To test the efficacy of this system, a small dataset of real routes and modifications were created manually. Models compared were a single LLama 2 instance, an ensemble of Llama 2 instances, and a ChatGPT-3.5 instance. Across all tests, ChatGPT performed best, but the single Llama 2 instance was able to perform quite similarly despite being an open-source model. The ensemble models also performed reasonably well, but their performance does not justify their increased inference cost. For all three model types, the overall system is generally effective and validates the error-checking functionality that is implemented as a major element of this project. Much work remains before this system could be used in a production setting, but this project demonstrates that such an application is possible given current technology constraints.

Appendix A
Prompt Templates

A. Modification or Trail

You are a helpful assistant who parses text and determines whether the user text is a modification request or a request for information on nearby trails.

A user will pass in text, which you should parse to determine which type of request is being made. Return ONLY the response to the last user request and nothing more.

Request options are as follows:
1. Modification – add if the user requests to modify their route.
2. Trail – add if the user requests information on nearby trails.

User: what parks are good for biking
Assistant: Trail

User: list nearby trails
Assistant: Trail

User: stop at boston common
Assistant: Modification

User: route through north ave and city hall
Assistant: Modification

User: show me Nature Park routes
Assistant: Trail

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

User: Looking for intermediate trails with rating above 3
Assistant: Trail

User: prefer trails
Assistant: Modification

User: {user input}
Assistant:

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

User: route through boston common and the empire state building
Assistant: route through boston common and the empire state building

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

User: stop at 120 charles rd and the golden gate bridge and prefer roads
Assistant: stop at 120 charles rd and the golden gate bridge | prefer roads

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
Assistant: pass through the airport | use city streets

User: skip 12th st and stop at Shelly McFarlin Park and the Natural History Museum.
Assistant: skip 12th st | stop at Shelly McFarlin Park and the Natural History Museum.

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

User: add 1050 wilkins dr and jackson library to stops, stay away from west ham
Assistant: add 1050 wilkins dr and jackson library to stops | stay away from west ham

User: I want to ride on roads
Assistant: 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: {user input}
Assistant:

C. 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: don't go on fipson dr
Assistant: avoid_area | fipson dr

User: stick to bike lanes
Assistant: prefer_path_type | bike lanes

User: add 1050 wilkins dr and jackson library to stops
Assistant: add_waypoints | 1050 wilkins dr | jackson library

User: {user input}
Assistant:

D. Trail Information Extraction

You are a helpful assistant who parses text and extracts user requirements from their request. A user will pass in text, which you should parse to determine which requirements the user specifies.

Return ONLY the response to the last user request and nothing more.

Request options are as follows:
1. Difficulty - difficulty rating of the trail. Options here are easy, intermediate, and hard.
2. Distance - how far from the user's current location the trail is.
3. Rating - rating of the trail. Options here are floats between 0 and 5.
4. Length - length of the trail.

User: list nearby trails
Assistant: No changes

User: Looking for intermediate trails with rating above 3
Assistant: Difficulty(Intermediate) | Rating($>$ 3)

User: show me Nature Park routes
Assistant: No changes

User: routes with rating above 2 and length of 5 miles or less
Assistant: Rating($>$ 2) | Length($<$ 5)

User: what are good places to ride nearby
Assistant: No changes

User: what trails are within 10 miles and are easier than advanced
Assistant: Distance($<$ 10) | Difficulty($<$ Advanced)

User: Looking for easy trails
Assistant: Difficulty(Easy)

User: tell me about trails nearby
Assistant: No changes

User: {user input}
Assistant:

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.