TravelAI: An Agentic Travel Planner

Author: Daniel Orshansky, Westlake HS
Mentor: Prof. Alex Dimakis

Abstract

This report describes the design and optimization of an AI travel agent that provides users with real-time travel information through an agentic flow. The system combines web search, Retrieval-Augmented Generation (RAG), and API integrations, which are coordinated by a task delegator that determines actions based on user prompts. A key challenge in this system is optimizing the task delegator to accurately identify tasks while minimizing excess LLM calls. Through techniques such as in-context learning and fine-tuning, the accuracy of the task delegator was significantly improved, improving the reliability and performance of the AI travel agent.

Introduction

Increasingly powerful LLMs enable new use cases in a wide range of fields. Modern techniques, such as RAG and agentic flow, allow LLMs to generate highly-accurate information and be integrated seamlessly into many applications, interacting with humans, APIs, or other LLMs. This report describes the design of an AI travel agent that provides users with the most up-to-date travel information for planning trips. The AI system is implemented as an agentic flow, which combines live web searching with RAG and API integrations to answer user queries using the latest available information and deliver details about flights, hotels, and restaurants. The web search is used to find relevant information from the internet, which is necessary to augment LLM's parametric knowledge with information not utilized in their training.

The primary challenge in developing TravelAI was creating a mechanism to determine when the system should engage in conversation, perform a web search and RAG on the results, or call an API to execute a specific task. The challenge is that, often, multiple actions need to be performed in response to one user prompt. For example, a user might say "Hello! Can you find me a ticket to Vienna from Austin? And, would I need a visa to go there?" In this case, the AI system would need to respond to the greeting using an LLM, find flights through a flight-reservation API, and search the web for information on travel requirements.

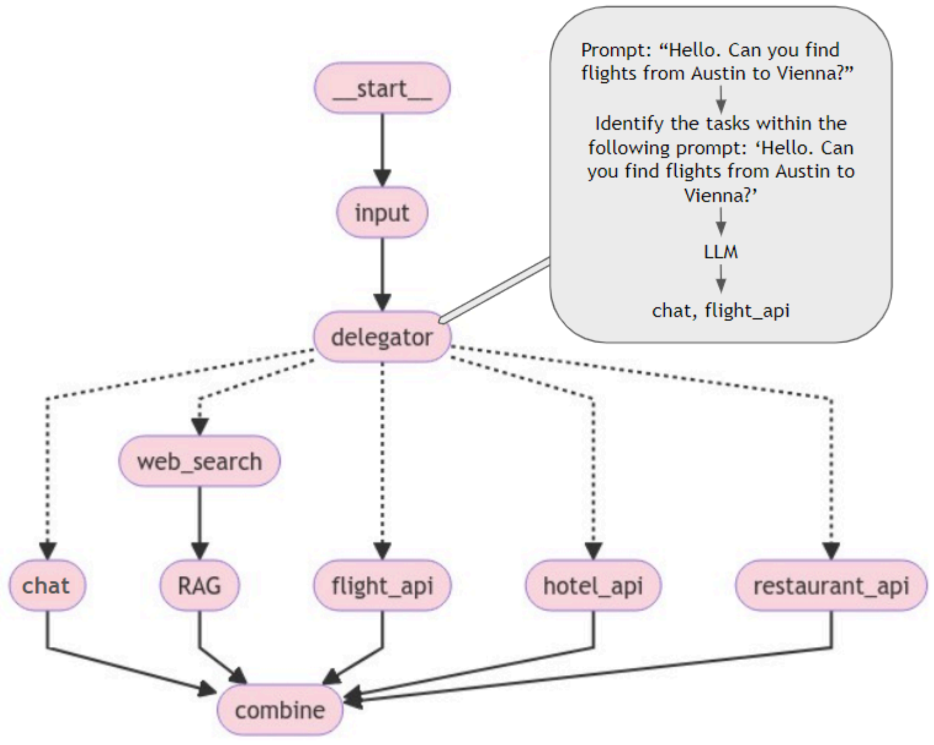
A naive approach to determine these tasks would be to use an LLM as a binary classifier for each task. E.g., for any user prompt, the system would ask an LLM if the prompt required a web search, and, separately, if it required flight information, etc. While this approach is simple and robust, it is very costly in LLM calls. Calling an LLM for each possible task type for every user prompt is very expensive and increases latency.

A more efficient approach is to provide the LLM with information about all potential task types at once and to then instruct it to identify and output all tasks present within the user prompt. This is better because it reduces the number of LLM calls required to identify tasks to one. However, the open-ended nature of this task delegator involves more complex reasoning for the LLM, making the system potentially more prone to errors without further optimization.

Once the system has determined what actions to take, it calls each corresponding agent to perform its task and generate an output. Finally, the system combines the outputs of the individual agents into one final response for the user.

System Design

The TravelAI system is designed as an agentic flow. It takes a user prompt, feeds it to a delegator agent, which identifies the tasks required to respond to the user, and then calls the agents to perform their tasks. Once each agent completes its task, the delegator combines all of their outputs into a final, cohesive response. The entire flow can be represented by the graph below:


__start__
input
delegator
Prompt: "Hello. Can you find flights from Austin to Vienna?"
Identify the tasks within the following prompt: 'Hello. Can you find flights from Austin to Vienna?'
LLM
chat, flight_api
web_search
chat
RAG
flight_api
hotel_api
restaurant_api
combine

The input node waits for a user prompt and passes it to the task delegator. The task delegator details each of the potential tasks (chat, web_search, flight_api, hotel_api, and restaurant_api) within an LLM prompt and instructs the LLM to identify the tasks within the user prompt. Often, however, users do not provide all of the necessary information for a task. For example, they may not specify check-in dates when asking for hotel reservations. To address this, the system also instructs the LLM to identify tasks that require more information.

The system proceeds to call the agents designated by the delegator. If the task is flagged as needing more information, the agent prompts the user for the missing details.

The LLM and API agents use the relevant information from the user prompt and feed it to either an LLM or the corresponding API. The API agents then use one extra LLM call to incorporate the returned data from the API into a response to the user.

The web-search RAG first uses an LLM to create a search query out of the user prompt. It uses the query to pull a fixed number of top websites from the web. However, using the raw HTML of a website for RAG is very inefficient. Raw HTML is full of tags, which are useless to the LLM. Instead, the system converts the raw HTML of the websites to markdown, which is much more understandable, and puts it into a vector store. Finally, it pulls the chunks most relevant to the query, adds them as context to the LLM prompt, and returns the output.

The delegator agent is the most common origin of inaccuracies within the system. Even using a high-performance proprietary LLM, such as GPT-4, the agent often makes mistakes as to whether API tasks need more information and whether it should make a web search or converse with the user. The next section describes the methods used to optimize and evaluate the delegator agent.

Evaluation

Synthetic Data Generation

In order to evaluate the delegator agent, test data is required in the form of synthetic user prompts and the tasks within those prompts.

To do this, for each prompt, a random subset of tasks is selected; on average, each prompt contains 2.4 tasks. Then, GPT-4o is prompted to create a user prompt containing the chosen tasks. The selected tasks are then treated as labels for the generated prompt. The LLM is configured with a temperature of 1.0 to promote variation among generated prompts.

To evaluate a strategy to optimize the delegator agent, it is on 50 synthetically generated prompts. The accuracy is then computed as the fraction of tasks which are correctly labeled.

Optimizations

Two LLM-augmentation techniques are explored to improve the performance of the delegator agent: in-context learning and fine-tuning.

In-context learning is a prompt engineering technique that incorporates examples of the desired behavior into an LLM prompt (also known as few-shot prompting). In-context learning is achieved by appending examples of user prompts and expected outputs to the end of the delegator LLM prompt. This strategy does not involve changing the weights of the model.

The delegator was evaluated using GPT-4o with and without in-context learning on the same synthetic dataset. The results are shown in the following table.

	Accuracy (% of correctly labeled tasks / total)
GPT-4o (control)	67%
GPT-4o with in-context learning	77%

The results show a 10% increase in the accuracy of the delegator when in-context learning is added. Yet, while in-context learning does produce a significant improvement, it still does not provide a satisfactory accuracy on its own.

Fine-tuning, the process of training pre-trained models further on specific datasets, is one approach to further improve performance. Because OpenAI does not give access to weights for any GPT models, Llama 3.1 is used for fine-tuning.
Fine-tuning can be used in conjunction with in-context learning, and the fine-tuned models discussed below are evaluated using in-context learning in their prompts.

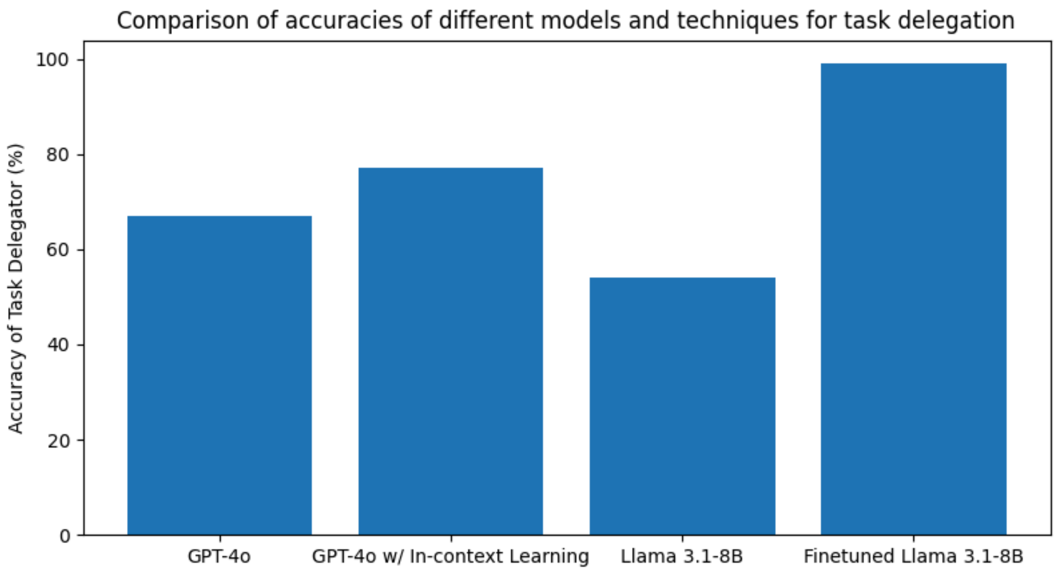
A synthetic dataset of 300 example prompts and task labels, created using the synthetic data generator described previously, was created for fine-tuning.
To make the fine-tuning process more compute-efficient, QLoRA was used on a 4-bit quantized model of Llama 3.1-8B Instruct. QLoRA is a form of PEFT (Parameter Efficient Fine Tuning), which appends a layer of weights onto a quantized pre-trained model, and fine-tunes only those weights. This significantly decreases the number of weights being affected, allowing the process to be run on lower-performing hardware. The weights were trained for 6 epochs. The model achieved a final training loss of 0.03, starting from an initial loss of 2.5.

The delegator was then evaluated both on the fine-tuned model and on the base Llama model, for comparison:

	Accuracy (% of correctly labeled tasks / total)
Llama 3.1-8B Instruct (control)	54%
Fine-tuned Llama 3.1-8B Instruct	99%

The fine-tuned Llama model achieves a nearly perfect accuracy and almost doubles the accuracy of the base model. These results demonstrate the power of finetuning and how fine-tuning can allow open-source models to achieve better performance than proprietary models on specific tasks.
Given that error within the delegator agent was the primary cause of inaccuracy within the entire TravelAI system, this strategy significantly improves its overall performance.

The following graph compares the accuracy of all of the evaluated optimization methods for the task delegator:



Conclusion

This report describes a set of design decisions needed to engineer a performant agentic AI system using a variety of AI methodologies, including RAG, in-context learning, and fine-tuning. The findings demonstrate the power of fine-tuning when combined with synthetic data generation and the ability of open-source models to offer performance competitive with that of proprietary models.