Towards AI · Follow publication

Member-only story

Building an AI-Powered Smart Travel Planner with Multi-Agent AI and LangGraph

10 min read · Jun 21, 2025

Vikram Bhat Follow

Listen Share More

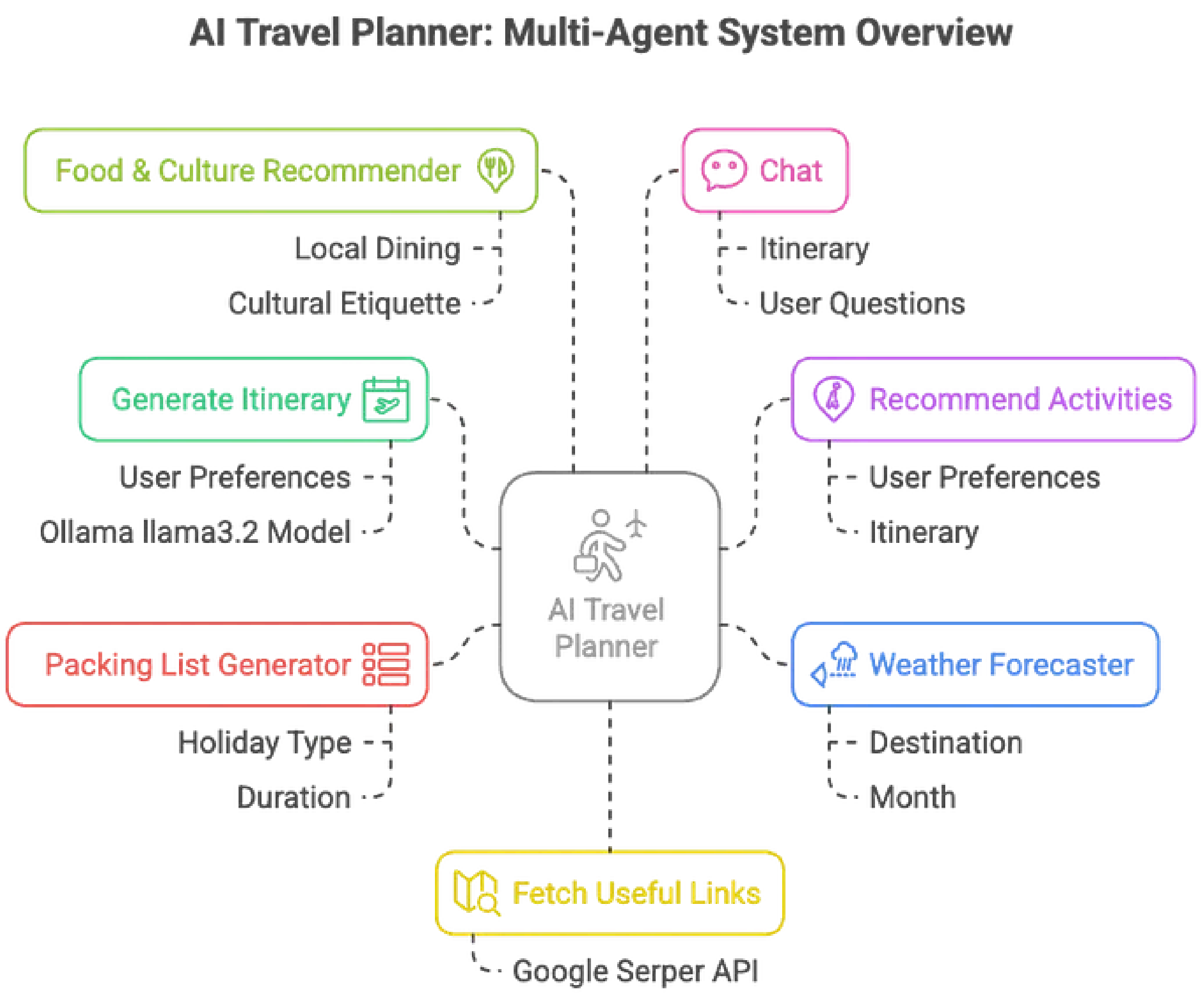


Image generated using napkin.ai

In today's rapidly evolving AI landscape, the ability to synthesize diverse information into personalized, actionable travel plans is more valuable than ever. Whether it's a spontaneous weekend escape or a meticulously curated vacation, travelers increasingly seek intelligent systems that can fetch real-time data from the web and convert it into meaningful itineraries, activity suggestions, and travel insights.

This blog introduces a **Multi-Agent Travel Itinerary Planner** built using **LangGraph** and powered by the **LLaMA 3.x** model. The system brings together a collaborative team of agents, each with a specific responsibility:

- **Generate Itinerary Agent** — crafts day-by-day travel plans based on user preferences
- **Recommend Activities Agent** — suggests unique local experiences

- **Fetch Useful Links Agent** — retrieves relevant travel guides using the Google Serper API
- **Weather Forecaster Agent** — provides weather expectations for the trip
- **Packing List Generator Agent** — curates a personalized packing checklist
- **Food & Culture Recommender Agent** — offers insights into local cuisine and etiquette
- **Chat Agent** — answers follow-up questions conversationally

Built with a user-friendly **Streamlit interface** , this system showcases how modern LLMs and agent-based workflows can come together to deliver a seamless travel planning experience.

You can find the complete code on GitHub:
👉 MultiAgents-with-Langgraph-TravelItineraryPlanner

Follow along to learn how to build this project step-by-step — from environment setup to multi-agent orchestration — and discover how you can tailor the system to your own travel needs.

. . .

Prerequisites to Run the Project

1. Python 3.10+

Ensure Python **3.10 or higher** is installed on your system. You can verify your version by running:

```
python --version
```

2. Install Required Python Libraries

The project depends on several Python libraries. Install all dependencies with:

```
pip install -r requirements.txt
```

Some key libraries include:

- **streamlit** — for the interactive web interface
- **langgraph** — for building the multi-agent graph workflow
- **langchain-community** — for integrating Ollama and external tools
- **python-dotenv** — to manage environment variables securely
- **fpdf** — for generating downloadable PDF itineraries
- **serper-wrapper** — to fetch web results via Google Serper API

3. Ollama + LLaMA 3.x Model

The core of this planner runs on **LLaMA 3.x** via **Ollama**. Make sure you:

- Install Ollama from ollama.com
- Start the Ollama server locally
- Download and run the LLaMA 3.x model with:

```
ollama run llama3
```

This allows your agents to use powerful local LLM capabilities.

4. API Keys

The **Fetch Useful Links Agent** uses the **Serper.dev** API to retrieve real-time travel tips and guides from the web. To set this up:

- Sign up at Serper.dev
- Generate your API key
- Create a `.env` file in the root directory and add:

```
SERPER_API_KEY=your_api_key_here
```

5. Project Structure (for modular code)

Agents are split into separate modules, your directory might look like this:

```
.
├── travel_agent.py
├── agents/
│   ├── generate_itinerary.py
│   ├── recommend_activities.py
│   ├── fetch_useful_links.py
│   ├── weather_forecaster.py
│   ├── packing_list_generator.py
│   ├── food_culture_recommender.py
│   └── chat_agent.py
├── utils_export.py
├── .env
├── requirements.txt
└── README.md
```

. . .

Step-by-Step Code Walkthrough

1. Importing Dependencies & Setting Up the Environment

The project begins by importing essential Python libraries and loading environment variables. This includes core libraries like `json`, `dotenv`, and `tempfile`, as well as LangChain components such as `ChatOllama` for interfacing with the local LLaMA 3.2 model. The `.env` file holds sensitive configuration such as API keys.

```python
import streamlit as st
import json
from typing import TypedDict, Annotated
from langgraph.graph import StateGraph, END
from langchain_core.messages import HumanMessage
from langchain_community.chat_models import ChatOllama
from langchain_community.utilities import GoogleSerperAPIWrapper
from dotenv import load_dotenv
import os
```

```
# Load environment variables
load_dotenv()

# Initialize LLM
llm = ChatOllama(model="llama3.2", base_url="http://localhost:11434")

# Initialize Google Search utility
search = GoogleSerperAPIWrapper()
```

2. Importing Agents

Once the environment is set up, we import the **modular agents**, each handling a distinct part of the travel planning process. This design allows each agent to focus on a single responsibility — such as generating the itinerary, suggesting activities, or forecasting weather — and makes the system easier to extend or debug.

```
from agents import generate_itinerary, recommend_activities, fetch_useful_links
```

2.1 🗺️ Itinerary Generation Agent

This agent is responsible for creating a day-by-day travel itinerary based on the user's preferences — like destination, budget, month, duration, and trip type.

```
from langchain_core.messages import HumanMessage
from langchain_community.chat_models import ChatOllama
import json

def generate_itinerary(state):
    llm = ChatOllama(model="llama3.2", base_url="http://localhost:11434")
    prompt = f"""
    Using the following preferences, create a detailed itinerary:
    {json.dumps(state['preferences'], indent=2)}

    Include sections for each day, dining options, and downtime.
    """
    try:
        result = llm.invoke([HumanMessage(content=prompt)]).content
        return {"itinerary": result.strip()}
    except Exception as e:
        return {"itinerary": "", "warning": str(e)}
```

It transforms structured preferences into a readable and actionable multi-day plan using LLaMA 3.x running locally via Ollama.

2.2 🎯 Activity Recommendation Agent

This agent augments the base itinerary by suggesting unique, location-specific activities. It blends user preferences and generated itinerary to recommend culturally relevant or offbeat experiences.

```
from langchain_core.messages import HumanMessage
from langchain_community.chat_models import ChatOllama
import json

def recommend_activities(state):
    llm = ChatOllama(model="llama3.2", base_url="http://localhost:11434")
    prompt = f"""
    Based on the following preferences and itinerary, suggest unique local acti
    Preferences: {json.dumps(state['preferences'], indent=2)}
    Itinerary: {state['itinerary']}

    Provide suggestions in bullet points for each day if possible.
    """
    try:
        result = llm.invoke([HumanMessage(content=prompt)]).content
        return {"activity_suggestions": result.strip()}
    except Exception as e:
        return {"activity_suggestions": "", "warning": str(e)}
```

2.3 🔗 Useful Links Agent

Instead of reinventing the wheel, this agent fetches up-to-date blogs, travel advisories, and guides using the Serper API. It enhances user trust by linking to credible web content.

```
from langchain_community.utilities import GoogleSerperAPIWrapper

def fetch_useful_links(state):
    search = GoogleSerperAPIWrapper()
    destination = state['preferences'].get('destination', '')
    month = state['preferences'].get('month', '')
    query = f"Travel tips and guides for {destination} in {month}"
    try:
        search_results = search.results(query)
        organic_results = search_results.get("organic", [])
```

```
            links = [
                {"title": result.get("title", "No title"), "link": result.get("link
                for result in organic_results[:5]
            ]
            return {"useful_links": links}
        except Exception as e:
            return {"useful_links": [], "warning": f"Failed to fetch links: {str(e)
```

It scrapes the top 5 relevant results and returns them as clickable titles for further reading.

2.4 🌦️ Weather Forecast Agent

This agent forecasts the climate for the chosen destination and travel month. It gives a natural-language summary with tips — like packing umbrellas or sunscreen — all generated from a prompt.

```
from langchain_core.messages import HumanMessage
from langchain_community.chat_models import ChatOllama

def weather_forecaster(state):
    llm = ChatOllama(model="llama3.2", base_url="http://localhost:11434")
    prompt = f"""
    Based on the destination and month, provide a detailed weather forecast inc
    Destination: {state['preferences'].get('destination', '')}
    Month: {state['preferences'].get('month', '')}
    """
    try:
        result = llm.invoke([HumanMessage(content=prompt)]).content
        return {"weather_forecast": result.strip()}
    except Exception as e:
        return {"weather_forecast": "", "warning": str(e)}
```

It ensures travelers are prepared for realistic weather conditions.

2.5 🧳 Packing List Agent

Based on the destination, duration, weather, and holiday type, this agent curates a comprehensive packing checklist. The list is seasonally and contextually aware.

```
from langchain_core.messages import HumanMessage
from langchain_community.chat_models import ChatOllama

def packing_list_generator(state):
    llm = ChatOllama(model="llama3.2", base_url="http://localhost:11434")
    prompt = f"""
    Generate a comprehensive packing list for a {state['preferences'].get('holi
    Include essentials based on expected weather and trip type.
    """
    try:
        result = llm.invoke([HumanMessage(content=prompt)]).content
        return {"packing_list": result.strip()}
    except Exception as e:
        return {"packing_list": "", "warning": str(e)}
```

2.6 🍜 Food & Culture Recommender Agent

This agent blends food exploration and cultural immersion. It suggests must-try local dishes, restaurants, and gives etiquette tips so travelers avoid faux pas and enjoy local hospitality.

```
from langchain_core.messages import HumanMessage
from langchain_community.chat_models import ChatOllama

def food_culture_recommender(state):
    llm = ChatOllama(model="llama3.2", base_url="http://localhost:11434")
    prompt = f"""
    For a trip to {state['preferences'].get('destination', '')} with a {state['
    1. Suggest popular local dishes and recommended dining options.
    2. Provide important cultural norms, etiquette tips, and things travelers s
    Format the response with clear sections for 'Food & Dining' and 'Culture &
    """
    try:
        result = llm.invoke([HumanMessage(content=prompt)]).content
        return {"food_culture_info": result.strip()}
    except Exception as e:
        return {"food_culture_info": "", "warning": str(e)}
```

2.7 💬 Conversational Chat Agent

Finally, this agent powers the interactive Q&A chat on the interface. Users can ask follow-up questions, request clarifications, or tweak their plans — and get humanlike answers instantly.

```python
from langchain_core.messages import HumanMessage
from langchain_community.chat_models import ChatOllama
import json

def chat_node(state):
    llm = ChatOllama(model="llama3.2", base_url="http://localhost:11434")
    prompt = f"""
    Context:
    Preferences: {json.dumps(state['preferences'], indent=2)}
    Itinerary: {state['itinerary']}

    User Question:
    {state['user_question']}

    Respond conversationally with insights or suggestions : keep your response
    {{ "chat_response": "Your response here" }}
    """
    try:
        result = llm.invoke([HumanMessage(content=prompt)]).content
        try:
            parsed = json.loads(result.strip())
            response = parsed.get("chat_response", result.strip())
        except json.JSONDecodeError:
            response = result.strip()
        chat_entry = {"question": state['user_question'], "response": response}
        chat_history = state.get('chat_history', []) + [chat_entry]
        return {"chat_response": response, "chat_history": chat_history}
    except Exception as e:
        return {"chat_response": "", "warning": str(e)}
```

It stores the chat history and responds based on full context (user preferences + itinerary), making the conversation feel cohesive and intelligent.

. . .

3. Wiring the Agents Together with LangGraph

To coordinate the flow between our modular agents, we use **LangGraph**, which lets us define a graph of steps (nodes) with clear entry/exit points. Each node corresponds to an agent we've already built.

LangGraph handles state transitions under the hood, ensuring each function updates and passes state forward cleanly.

3.1 Defining the Global State

Before wiring the agents, we define a shared `GraphState` to track all intermediate outputs from each agent — from itinerary to weather forecast to chat history.

```
from typing import TypedDict, Annotated
class GraphState(TypedDict):
    preferences_text: str
    preferences: dict
    itinerary: str
    activity_suggestions: str
    useful_links: list[dict]
    weather_forecast: str
    packing_list: str
    food_culture_info: str
    chat_history: Annotated[list[dict], "List of Q&A"]
    user_question: str
    chat_response: str
```

This allows all agents to read from and write to a shared, evolving state dictionary.

3.2 Registering Agents as Nodes

We now add each of the imported agent functions to the LangGraph workflow as nodes:

```
from langgraph.graph import StateGraph, END
workflow = StateGraph(GraphState)
# Register agents as nodes
workflow.add_node("generate_itinerary", generate_itinerary)
workflow.add_node("recommend_activities", recommend_activities)
workflow.add_node("fetch_useful_links", fetch_useful_links)
workflow.add_node("weather_forecaster", weather_forecaster)
workflow.add_node("packing_list_generator", packing_list_generator)
workflow.add_node("food_culture_recommender", food_culture_recommender)
workflow.add_node("chat", chat_agent)
```

Each node name is a string identifier used to reference the function in the graph.

3.3 Defining the Flow

We now specify the **entry point** and connect each node to an endpoint (`END`). For simplicity, we run one node per user action, but you can define complex paths if needed.

```
workflow.set_entry_point("generate_itinerary")
# Define terminal nodes for single-pass flow
workflow.add_edge("generate_itinerary", END)
workflow.add_edge("recommend_activities", END)
workflow.add_edge("fetch_useful_links", END)
workflow.add_edge("weather_forecaster", END)
workflow.add_edge("packing_list_generator", END)
workflow.add_edge("food_culture_recommender", END)
workflow.add_edge("chat", END)
```

Finally, compile the graph:

```
graph = workflow.compile()
```

This gives us a callable object that takes in `GraphState` and returns an updated version after the selected node runs.

3. 4 Example Usage

To invoke any agent as part of the graph:

```
result = graph.invoke(st.session_state.state)
```

You can also invoke nodes directly (if needed) with:

```
workflow.get_node("recommend_activities")(st.session_state.state)
```

This is particularly useful for button-driven interactions in the Streamlit frontend — which we'll cover next.

. . .

4. Building the Streamlit UI

The Streamlit interface ties everything together, letting users enter travel preferences, invoke individual agents, and view results — all with a clean, interactive layout.

4.1 Session State Initialization

We initialize a comprehensive session state dictionary to hold all agent outputs and inputs persistently across user interactions:

```
if "state" not in st.session_state:
    st.session_state.state = {
        "preferences_text": "",
        "preferences": {},
        "itinerary": "",
        "activity_suggestions": "",
        "useful_links": [],
        "weather_forecast": "",
        "packing_list": "",
        "food_culture_info": "",
        "chat_history": [],
        "user_question": "",
        "chat_response": ""
    }
```

This ensures every part of our system — from itinerary to chat history — stays synchronized.

4.2 User Input Form

We present a form for users to input travel details like destination, month, duration, holiday type, and budget:

```
with st.form("travel_form"):
    col1, col2 = st.columns(2)
    with col1:
        destination = st.text_input("Destination")
        month = st.selectbox("Month of Travel", [...])
        duration = st.slider("Number of Days", 1, 30, 7)
        num_people = st.selectbox("Number of People", [...])
    with col2:
        holiday_type = st.selectbox("Holiday Type", [...])
        budget_type = st.selectbox("Budget Type", [...])
        comments = st.text_area("Additional Comments")
    submit_btn = st.form_submit_button("Generate Itinerary")
```

Upon submission, we update the session state with these preferences and trigger the itinerary generation agent.

4.3 Agent Action Buttons

Once an itinerary exists, we display a row of buttons for users to fetch activity suggestions, useful links, weather forecasts, packing lists, and food/culture info — each button triggers the respective agent and updates the UI dynamically:

```
col_btn1, col_btn2, col_btn3, col_btn4, col_btn5 = st.columns(5)
with col_btn1:
    if st.button("Get Activity Suggestions"):
        result = recommend_activities(st.session_state.state)
        st.session_state.state.update(result)
with col_btn2:
    if st.button("Get Useful Links"):
        result = fetch_useful_links(st.session_state.state)
        st.session_state.state.update(result)
# ... similarly for weather, packing list, food & culture
```

This modular approach lets users explore different aspects of their trip interactively.

. . .

4.4 Displaying Agent Outputs

We show each agent's output inside expandable panels for a tidy, organized view:

```
if st.session_state.state.get("activity_suggestions"):
    with st.expander("🎯 Activity Suggestions"):
        st.markdown(st.session_state.state["activity_suggestions"])
if st.session_state.state.get("useful_links"):
    with st.expander("🔗 Useful Links"):
        for link in st.session_state.state["useful_links"]:
            st.markdown(f"- [{link['title']}]({link['link']})")
# ... similar blocks for weather, packing list, food & culture info
```

This allows users to dive deeper into specific info as they prefer.

4.5 Interactive Chat

To round off the experience, we include a chat interface powered by the `chat_agent`, allowing users to ask questions about their itinerary and get contextual, conversational responses:

```
st.markdown("### Chat About Your Itinerary")
for chat in st.session_state.state["chat_history"]:
    with st.chat_message("user"):
        st.markdown(chat["question"])
    with st.chat_message("assistant"):
        st.markdown(chat["response"])
if user_input := st.chat_input("Ask something about your itinerary"):
    st.session_state.state["user_question"] = user_input
    result = chat_agent(st.session_state.state)
    st.session_state.state.update(result)
    st.experimental_rerun()
```

This seamless chat experience makes the travel planner feel interactive and personal.

4.6 PDF Export

Finally, we offer an export button for users to download their itinerary as a PDF document:

```
if st.button("Export as PDF"):
    pdf_path = export_to_pdf(st.session_state.state["itinerary"])
    if pdf_path:
        with open(pdf_path, "rb") as f:
            st.download_button("Download Itinerary PDF", f, file_name="itinerar
```

. . .

This UI setup keeps the user journey smooth and intuitive, while effectively integrating your multi-agent backend powered by LangGraph and the llama3.x model.

. . .

Conclusion

What makes this project strong is how it breaks down a complex problem like travel planning into smaller, focused agents — each handling a specific task like creating the itinerary, suggesting activities, fetching live web data, or providing cultural tips. This modular design not only keeps the code clean and easier to maintain, but also lets you swap or improve individual agents without overhauling the entire system. Plus, using LangGraph to wire everything together gives you a powerful way to manage the workflow and data flow between agents smoothly.

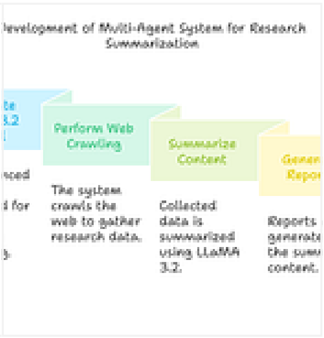
On top of that, the Streamlit interface makes the whole thing approachable and interactive, so users get real-time feedback and can explore every aspect of their trip with just a few clicks. This combination of modular AI agents, a flexible orchestration layer, and a user-friendly UI is what really sets this code apart — it's practical, scalable, and ready for real-world use. If you're interested in building AI-powered applications, this kind of multi-agent setup is a great pattern to learn and adapt.

Explore More:

Multi-Agent System for Research Summarization and Reporting with Crew AI

Leveraging Crew AI and Gradio for Efficient Multi-Agent Research Summarization and Automated Reporting

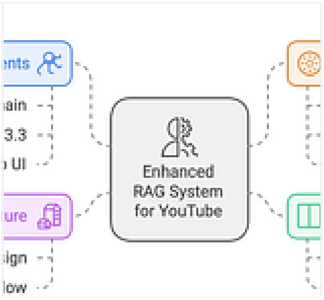
ai.gopubby.com



Building an Enhanced RAG System to Summarize and Converse with YouTube Videos

Leveraging LangChain, Ollama Llama 3.2, and Gradio UI to create an advanced RAG system for summarizing and interacting...

ai.gopubby.com



Connect with me on Linkedin for more updates and professional insights.

https://www.linkedin.com/in/vikrambhat249/

Github Repo: https://github.com/vikrambhat2/MultiAgents-with-Langgraph-TravelItineraryPlanner

Generative Ai Tools Programming AI Data Science Agents

Follow

Published in Towards AI

90K followers · Last published just now

Making AI accessible to 100K+ learners. Find the most practical, hands-on and comprehensive AI Engineering and AI for Work certifications at academy.towardsai.net - we have pathways for any experience level. Monthly cohorts still open — use COHORT10 for 10% off!

Follow

Written by Vikram Bhat

534 followers · 202 following

Data Scientist specializing in LLMs, RAG systems, and ML model deployment. Skilled in Python, R, AWS Sagemaker, optimizing search with vector databases.

Responses (3)

Bgerby

What are your thoughts?

Nicholas Kersting
Jun 28 (edited)

Thank you for being brave enough to tackle this hard problem of Travel Planning! Your method is very systematic, but before I download and install your code I would love to see does it actually work? Do you have a little demo available (if not... more

2 1 reply Reply

Aditya Inamdar
Jun 27

LLaMA 3

It's good one

7 1 reply Reply

R Amin
Jun 26

i found a litte bug in your code. the "workflow.add_edge" in "travel_agent.py" all lead to "end" insted of to the "next agent". hope i could help with that.

beside this i love your work. thank you.

2 1 reply Reply

More from Vikram Bhat and Towards AI

In Towards AI by Vikram Bhat

Hybrid Time-Series Forecasting with LangGraph, Prophet & Large Language Models (LLMs)

Build a Local, Explainable Pipeline with Parallel Models, Automated Evaluation (RMSE/MAE/SMAPE), and Agent Explanations.

Aug 12 68

In Towards AI by Teja Kusireddy

We Spent $47,000 Running AI Agents in Production. Here's What Nobody Tells You About A2A and MCP.

Multi-agent systems are the future. Agent-to-Agent (A2A) communication and Anthropic's Model Context Protocol (MCP) are revolutionary. But...

Oct 16 1.8K 49

In Towards AI by Ashish Abraham

No Libraries, No Shortcuts: LLM from Scratch with PyTorch

The no BS guide to build, train, and fine-tune a Transformer architecture from scratch

Oct 2 1.1K 14

In AI Advances by Vikram Bhat

AI-Powered Resume Matcher: Multi-Agent System with Streamlit and CrewAI

Automate resume-JD matching, enhancement and cover letter generation with a local multi-agent system built using CrewAI, Streamlit and...

Jul 3 194

See all from Vikram Bhat

See all from Towards AI

Recommended from Medium

In Artificial Intelligence in Plain English by Piyush Agnihotri

Building Agentic RAG with LangGraph: Mastering Adaptive RAG for Production

Build intelligent RAG systems that know when to retrieve documents, search the web, or generate responses directly

Jul 20 2.2K 30

In Data Science Collective by Paolo Perrone

Why Most AI Agents Fail in Production (And How to Build Ones That Don't)

I'm a 8+ years Machine Learning Engineer building AI agents in production.

Jun 16 2.1K 37

In MongoDB by MongoDB

Build AI Agents Worth Keeping: The Canvas Framework

This article was written by Mikiko Bazeley.

Oct 8 88 4

In Towards AI by Hamza Boulahia

Agentic Design Patterns with LangGraph

If there’s one thing I’ve learned building AI systems over the last couple of years, it’s this: patterns matter. Whether we’re designing...

Sep 30 355 7

In Level Up Coding by Fareed Khan

Building a Multi-Agent AI System with LangGraph and LangSmith

A step-by-step guide to creating smarter AI with sub-agents

Jun 1 1.6K 24

In Generative AI by Gao Dalie (高達烈)

DeepSeek-OCR + LLama4 + RAG Just Revolutionized Agent OCR Forever

During the weekend, I scrolled through Twitter to see what was happening in the AI community. Once again, DeepSeek has drawn worldwide...

6d ago 193

See more recommendations