WakeWander: Travel Planning Assistant using LangGraph and Plan-and-Execute

Suraj Prasai
Department of Computer Science
Wake Forest University
Winston-Salem, North Carolina
Email: prass25@wfu.edu

***Abstract*—Large Language Models (LLMs) are good at understanding and generating text, but they often fail when tasks need multiple steps, memory between steps, error handling, and user input during execution. This project introduces WakeWander, a travel planning agent built on the Plan-and-Execute architecture using LangGraph to manage multi-step workflows. Unlike ReAct, which switches between thinking and acting at every step, Plan-and-Execute builds a full plan first and then runs each step in order. This gives more predictable timing and makes it easier to track progress. We solved three main technical problems: keeping state consistent across workflow nodes, getting reliable JSON output from the LLM, and adding smooth human-in-the-loop interrupts. Our tests show response times of 8-12 seconds with no state corruption, which confirms that Plan-and-Execute works well for structured planning tasks. The system cuts travel planning time from hours to minutes while still letting users make key decisions.**

I. Introduction

Large Language Models have made it possible to build smart agents that can reason, plan, and carry out complex tasks. But using LLMs for real applications that need multi-step planning is still hard. Normal LLM interactions are stateless, the model handles each request on its own without remembering what happened before. When tasks need step-by-step decisions, managing dependencies, and recovering from errors, simple prompt changes are not enough.

Travel planning is a good example of this problem. Building a complete trip itinerary means making many connected choices: figuring out what the user wants, looking up possible destinations, checking the budget, picking hotels, and scheduling daily activities. Each step depends on the one before it, and the whole process has to adjust when users add missing details or change their minds during planning.

Current LLM agent designs fall into three main types. The ReAct (Reason + Act) pattern [1] mixes reasoning and action together, which allows flexible responses but causes unpredictable run times and can get stuck in loops. The Reflection pattern creates outputs and then reviews them over and over, which improves quality but adds delay. The Plan-and-Execute pattern [2] makes a full plan first and then runs steps one by one, giving predictable behavior but less flexibility.

This project presents WakeWander, a travel planning agent that uses the Plan-and-Execute architecture with LangGraph, a framework for building stateful, multi-step LLM applications. Our main contributions are:

- A working Plan-and-Execute system for travel planning with custom workflow nodes
- Solutions to three key technical problems: state consistency, reliable LLM output, and human-in-the-loop integration
- Test results showing predictable performance with 8-12 second response times
- Open-source code that others can adapt for similar planning tasks

II. Related Work

A. LLM Agent Architectures

Several design patterns have appeared for building LLM agents that handle multi-step reasoning.

ReAct. Yao et al. [1] proposed the ReAct approach, which combines reasoning and acting in language models. At each step, the agent thinks, takes an action, and checks the result before moving on. This method adapts well to unexpected situations, but the cost and time are hard to predict since we cannot know how many think-act cycles will happen. When we tested ReAct for travel planning, response times varied between 20-40 seconds, while Plan-and-Execute took only 8-12 seconds.

Plan-and-Execute. Wang et al. [2] showed that separating high-level planning from step-by-step execution makes zero-shot reasoning much better. By creating a full task dependency graph at the start, this approach reduces the mental load on the LLM when running each step. *WakeWander* builds on this idea and turns it into a complete workflow system with state memory and points where users can step in.

Multi-Agent Systems. SagaLLM [3] added context management, validation, and transaction guarantees for multi-agent LLM planning. Their work focuses on coordinating multiple agents, while our single-agent design with workflow nodes handles similar state consistency problems within one execution context.

B. Workflow Orchestration Frameworks

Agentic Frameworks (AutoGen & CrewAI). Microsoft's **AutoGen** [4] uses conversational patterns between agents, where state is kept as shared dialogue history. **CrewAI** [5] wraps orchestration into ”roles” and ”crews” to make things simpler. But these frameworks often hide the actual state structure. For production systems that need strict type safety and clean state updates, the conversation-based approach can cause state corruption when agents make up data formats.

LangGraph. LangGraph [7] takes a different approach by showing the control flow as a directed acyclic graph (DAG) with clear state definitions. It treats the LLM as a state transition engine rather than a chat partner. This lower-level design let us build custom interrupts and proper state management in *WakeWander*.

C. Human-in-the-Loop AI Systems

Interactive machine learning has long valued human oversight [6]. In LLM agents, human-in-the-loop serves several purposes: filling in missing information, checking important decisions, and keeping users in control instead of automating everything. Our system uses LangGraph's interrupt feature to pause at two key points: when asking follow-up questions and when showing destination choices for the user to pick.

III. Method

A. System Architecture Overview

WakeWander implements a directed graph that orchestrates the travel planning workflow. The system accepts natural language travel requests, extracts structured preferences, gathers missing information through conversation, researches destinations, and generates complete day-by-day itineraries.

The technology stack comprises: Gemini 2.5 Flash for LLM inference, LangGraph for workflow orchestration, FastAPI with Server-Sent Events for real-time streaming, Supabase (PostgreSQL) for data persistence, and Next.js for the frontend interface.

B. State Definition and Management

Our system uses a `AgentState` TypedDict at its core, which maintains all information across workflow nodes. The state includes user preferences, research data, budget allocations, and the final itinerary.

We use dataclasses for structured data objects (`TravelPreferences`, `BudgetAllocation`, `LocationData`) to ensure type safety and provide convenient serialization methods. The `Annotated` type with the `add` operator enables automatic accumulation of messages and errors across nodes without overwriting previous entries.

```
class AgentState(TypedDict):
    user_request: str
    preferences: Optional[TravelPreferences]
    missing_info: List[str]
    research_data: List[LocationData]
    budget_allocation: Optional[BudgetAllocation]
    selected_location: Optional[LocationData]
    daily_plans: List[Dict[str, Any]]
    itinerary: Optional[Dict[str, Any]]
    status: str
    messages: Annotated[List[Dict], add]
    errors: Annotated[List[str], add]
```

Listing 1. Core state definition

State Consistency Challenge. LangGraph automatically merges state updates from each node. If a node returns `{‘‘preferences’’: new_prefs}`, it updates only that field while preserving others. However, partial updates could overwrite fields if nodes return inconsistent structures. Our solution enforces immutable updates: each node returns only the fields it modifies, and we validate updates through Pydantic-based dataclasses before applying them.

C. Graph Construction and Routing

The workflow graph is constructed using LangGraph's `StateGraph` class. Each node is a function that receives the current state and returns a partial state update.

```
workflow = StateGraph(AgentState)

# Add nodes
workflow.add_node("general_handler", general_handler
    )
workflow.add_node("analyze_input", analyze_input)
workflow.add_node("identify_missing",
    identify_missing)
workflow.add_node("ask_question", ask_question)
workflow.add_node("research", research)
workflow.add_node("analyze", analyze)
workflow.add_node("season_recs", season_recs)
workflow.add_node("plan_days", plan_days)
workflow.add_node("finalize", finalize)

# Conditional edge example
def route_after_identify_missing(state):
    if state.get('missing_info'):
        return "ask_question"
    if state.get('status') == 'researching':
        return "research"
    return END

workflow.add_conditional_edges(
    "identify_missing",
    route_after_identify_missing,
    {"ask_question": "ask_question",
     "research": "research",
     END: END}
)
```

Listing 2. Graph construction with conditional routing

Routing functions examine the current state to determine the next node. This enables dynamic workflow adaptation, if required information is missing, the graph routes to the question asking node and once completed, it proceeds to research.

D. LLM Output Reliability

LLMs do not guarantee structured output formats. Even with explicit JSON instructions in prompts, models occasionally add explanatory text, markdown code blocks, or malformed syntax. We implement robust JSON extraction:

```
def extract_json(content):
    # Handle list responses from Gemini
    if isinstance(content, list):
        content = ''.join([
            item.get('text', '')
            if isinstance(item, dict)
            else str(item)
            for item in content
        ])
    # Remove markdown code blocks
    content = content.strip()
    if content.startswith('```'):
        content = re.sub(r'^```(?:json)?\s*', '',
            content)
        content = re.sub(r'\s*```$', '', content)

    # Find JSON object in text
    json_match = re.search(
        r'\{[^{}]*(?:\{[^{}]*\}[^{}]*)*\}',
        content, re.DOTALL
    )
    if json_match:
        return json.loads(json_match.group())
    return json.loads(content)
```

Listing 3. Robust JSON extraction from LLM responses

Additionally, all prompts include explicit instructions (*"Return ONLY valid JSON, no markdown code blocks"*) and provide example outputs to guide the model toward the expected format.

E. Human-in-the-Loop Integration

Two nodes require user input: `ask_question` for gathering missing preferences and `analyze` for destination selection. We use LangGraph's `interrupt()` function to pause execution:

```
def ask_question(state):
    field = state['missing_info'][0]
    question = questions[field]

    # Pause execution, return control
    user_response = interrupt({
        "type": "question",
        "field": field,
        "question": question
    })

    # Execution resumes here with answer
    prefs_dict = state['preferences'].to_dict()
    prefs_dict[field] = parse_response(
        field, user_response
    )

    return {
        "preferences": TravelPreferences(**
            prefs_dict),
        "missing_info": remaining_fields
    }
```

Listing 4. Human-in-the-loop interrupt pattern

When `interrupt()` is called, LangGraph's checkpointer saves the current state. The calling application receives the interrupt payload and can present it to the user. When the user responds, the application calls `graph.invoke()` with `Command(resume=user_response)` and the same `thread_id`, restoring execution at the exact point after the interrupt.

The graph is compiled with `interrupt_before=[''ask_question'', ''analyze'']` to ensure interrupts occur at node boundaries, enabling clean state checkpointing.

F. Node Implementations

General Handler. Distinguishes travel requests from general conversation using keyword matching. General greetings and messages receive friendly responses and directs the conversation towards trip planning.

Analyze Input. Extracts structured preferences from natural language using an LLM prompt that requests JSON output with fields for destination, duration, budget, season, and interests.

Identify Missing. Checks required fields (season, budget, duration_days) and populates the `missing_info` list for fields with null values.

Research. Queries the LLM for 6-8 destination recommendations matching the user's criteria, or researches a specified destination with alternatives.

Analyze. Calculates budget allocation (40% accommodation, 25% food, 25% activities, 5% transport, 5% contingency), filters locations within budget, and presents options for user selection.

Season Recommendations. Generates season-specific advice including weather expectations, packing suggestions, and seasonal events.

Plan Days. Creates detailed day-by-day itineraries with specific hotels, restaurants, activities, times, and costs.

Finalize. Assembles the complete itinerary with budget summaries and stores it for retrieval.

IV. Experiments

A. Experimental Setup

We evaluated *WakeWander* on a diverse set of 15 travel planning scenarios covering various destinations, budgets ($1,000-$10,000), durations (3-14 days), seasons, and group sizes. The system was deployed with the FastAPI backend communicating with the Gemini 2.5 Flash API.

B. Performance Metrics

Table I summarizes the key performance metrics observed during evaluation.

Response Time. Complete itinerary generation averaged 8-12 seconds, compared to 20-40 seconds observed in the ReAct prototype. The predictable timing validates our choice of Plan-and-Execute.

State Consistency. Across all test runs, zero state corruption incidents occurred. The immutable update pattern and Pydantic validation successfully prevented partial overwrites and type mismatches.

LLM Output Reliability. The robust JSON extraction succeeded in 97.6% of cases on first attempt. The remaining 2.4% were recovered through retry mechanisms, resulting in 100% eventual success.

Human-in-the-Loop. All interrupt-resume cycles completed successfully. We observed seamless transitions when answering questions or selecting destinations, with no perception of workflow discontinuity.

TABLE I
WAKEWANDER PERFORMANCE METRICS

Metric	**Value**
End-to-end response time	8-12 seconds
State corruption rate	0%
JSON parsing success rate	97.6%
Interrupt resume success rate	100%
Average LLM calls per itinerary	6-8

TABLE II
ARCHITECTURE COMPARISON FOR TRAVEL PLANNING

Factor	**ReAct**	**Plan-and-Execute**
Predictability	Low	High
Response time	20-40s	8-12s
Progress visibility	Limited	Clear steps
Flexibility	High	Medium
Loop risk	Present	None

C. Comparison with ReAct

Table II compares Plan-and-Execute with ReAct based on our implementation experiments.

ReAct excels in exploratory tasks where the solution path is unknown, such as debugging or open-ended research. However, for structured domains like travel planning where the workflow is well-defined, Plan-and-Execute provides superior predictability and efficiency.

V. CONCLUSION

This project presented *WakeWander*, a travel planning agent demonstrating effective application of the Plan-and-Execute architecture for structured multi-step agentic workflows. Our implementation addresses practical challenges in building production LLM agents: maintaining state consistency through immutable updates and type-safe dataclasses, ensuring LLM output reliability through robust JSON extraction, and integrating human-in-the-loop interactions through LangGraph's interrupt mechanism.

The experimental results validate Plan-and-Execute as the appropriate choice for well defined workflows, achieving predictable 8-12 second response times compared to 20-40 seconds with ReAct. The 0% state corruption rate and 100% interrupt success rate demonstrate the robustness of our technical solutions.

Limitations. The current implementation supports single destination trips only. The system relies on LLM generated cost estimates rather than real-time booking APIs, limiting practical accuracy. Additionally, the fixed budget allocation percentages may not suit all travel styles.

Future Work. The future extension for *WakeWander* includes multi-destination support, real-time booking API integration, a vector database for RAG-based recommendations from past itineraries, and A/B testing to compare Plan-and-Execute with ReAct in production. The planning and reasoning pipeline demonstrated here is domain-agnostic and applicable to other structured workflows in finance, healthcare, and automation.

VI. CODE AVAILABILITY

To ensure reproducibility, we have made the entire implementation of our framework publicly available. Our codebase is available at https://github.com/spygaurad/WakeWander. All dependencies and execution instructions can be found in the repository's README file.

ACKNOWLEDGMENT

We acknowledge Anthropic's Claude's assistance in generating frontend structure and debugging, Supabase for free PostgreSQL hosting that enabled rapid prototyping, Gemini API for fast and cheap LLM access and the LangChain team for the LangGraph framework that made this implementation possible.

REFERENCES

[1] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "ReAct: Synergizing reasoning and acting in language models," in *Proc. 11th Int. Conf. Learning Representations (ICLR)*, 2023.

[2] L. Wang, W. Xu, Y. Lan, Z. Hu, Y. Lan, R. K.-W. Lee, and E.-P. Lim, "Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models," in *Proc. 61st Annu. Meeting Assoc. Computational Linguistics*, 2023, pp. 2609–2634.

[3] E. Y. Chang and L. Geng, "SagaLLM: Context management, validation, and transaction guarantees for multi-agent LLM planning," *Proc. VLDB Endow.*, vol. 18, no. 12, pp. 4874–4886, Aug. 2025.

[4] Q. Wu *et al.*, "AutoGen: Enabling next-gen LLM applications," *arXiv preprint arXiv:2308.08155*, 2023.

[5] J. Moura, "CrewAI: Framework for orchestrating role-playing autonomous AI agents," 2024, https://github.com/joaomdmoura/crewAI, accessed Dec. 5, 2024.

[6] S. Amershi, M. Cakmak, W. B. Knox, and T. Kulesza, "Power to the people: The role of humans in interactive machine learning," *AI Magazine*, vol. 35, no. 4, pp. 105–120, 2014.

[7] LangChain, "LangGraph: Build stateful, multi-actor applications with LLMs," 2024. [Online]. Available: https://langchain-ai.github.io/langgraph/, accessed Dec. 5, 2024.

[8] Google, "Gemini API documentation," 2024. [Online]. Available: https://ai.google.dev/gemini-api/docs, accessed Dec. 5, 2024.

[9] J. Wei *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," in *Advances in Neural Information Processing Systems*, vol. 35, 2022, pp. 24824–24837.

[10] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: Language agents with verbal reinforcement learning," in *Advances in Neural Information Processing Systems*, vol. 36, 2023.

[11] J. S. Park *et al.*, "Generative agents: Interactive simulacra of human behavior," in *Proc. 36th Annu. ACM Symp. User Interface Software and Technology*, 2023, pp. 1–22.

[12] T. Brown *et al.*, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, vol. 33, 2020, pp. 1877–1901.