Agentification and Agent Design Patterns - Travel Planner

Matéo Lelong Maxim Bocquillion Samy Yacef

November-December 2025

Contents

1 Introduction

Every year, more than one billion people go on vacation. According to a study conducted by the American company Luth Research for Expedia. Travelers spend an average of five hours online visiting more than 81 pages to plan their trip.

The TravelBro was designed to solve this problem and save valuable time for all travelers. With our solution, the usual five hours spent on planning are reduced to just two minutes: we take care of everything, from selecting destinations to making reservations.

By simplifying the planning process, the TravelBro allows travelers to focus on what matters most: fully enjoying their vacation.

The goals of this project are to provide a simple user interface that suggests a complete vacation plan including flights, hotels, and activities on-site, whether traveling alone, with friends, or with family!

Underneath this proposition lies an agentic system capable of extracting key details from a user prompt, such as the number of travelers, departure and return dates, budget, and destination. Leveraging this information, the agent interfaces with various external APIs to source flights, accommodations, and activities.

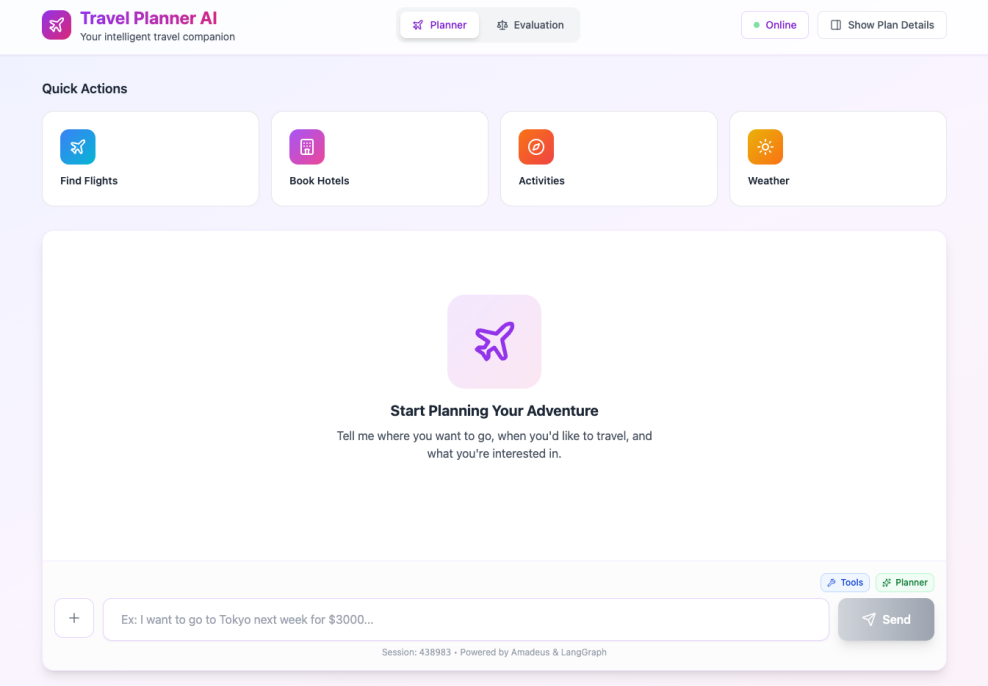

Figure 1: The TravelBro Frontend

2 Architecture

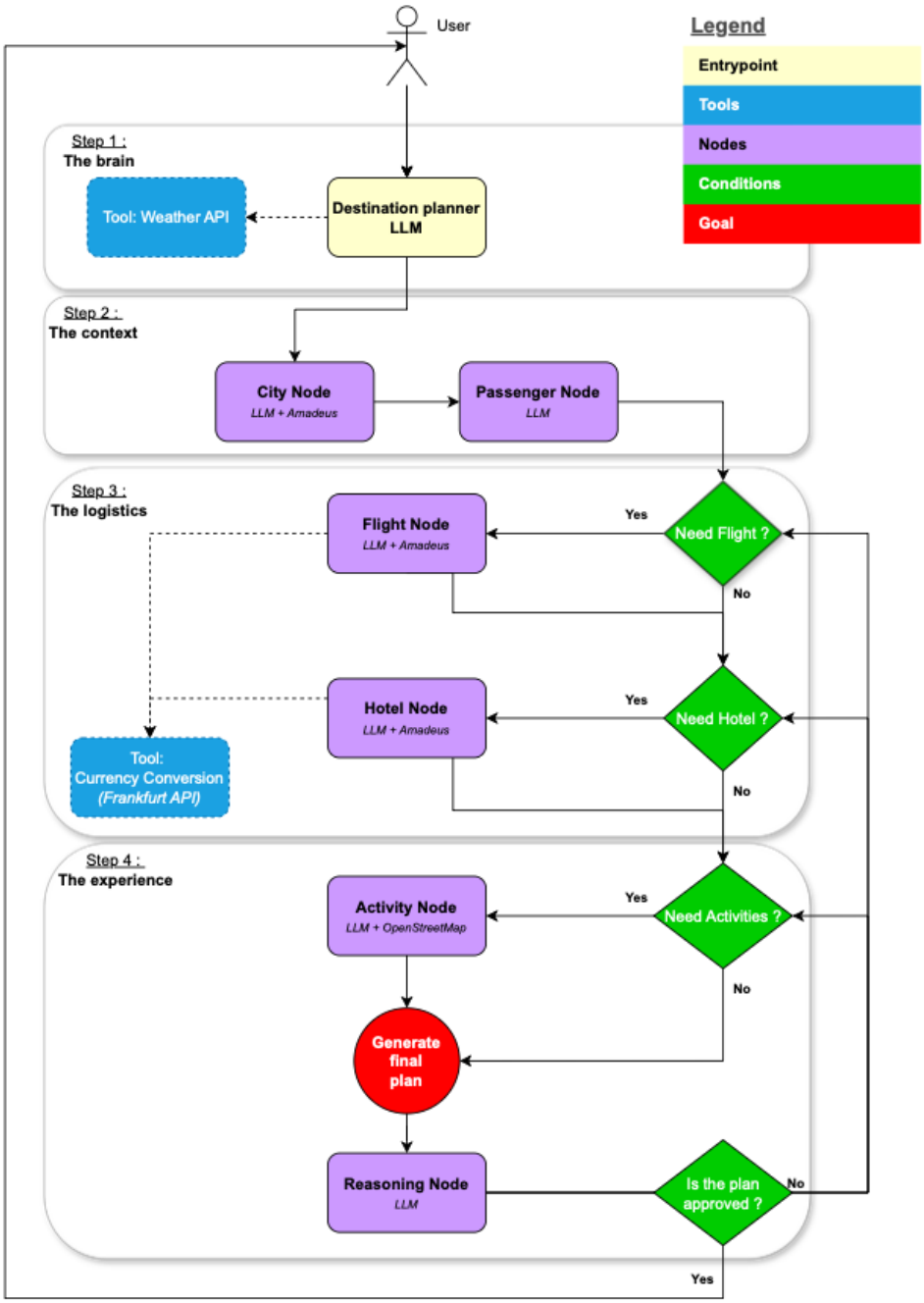

Figure 2: Agent-based architecture diagram

As shown in the architecture diagram, the system consists of several specialized nodes that work together to build a travel itinerary:

- **Planner Node**: This is the entry point of the logic. It analyzes the user's raw request to extract structured data such as the destination, travel dates, and budget. If information is missing, it triggers a request for user input.
- **City Resolver Node**: This node maps location names (e.g., "New York") to the

required IATA airport or city codes (e.g., "JFK", "NYC") using a search tool, with LLM fallback if needed.

- **Passenger Node**: This node extracts and validates traveler details, including passenger counts (adults, children, infants) and travel class.
- **Flight Node**: This node searches for flight offers using IATA codes, dates, and passenger details, then filters results by budget and preferences, converting currencies as needed.
- **Hotel Node**: Similar to the flight node, this searches for accommodations at the destination. It evaluates offers based on price, location coordinates, and contact information, ensuring the selected option fits within the remaining budget.
- **Activity Node**: This node fetches local tours and attractions via the Amadeus API based on the destination's latitude and longitude, further enriching the itinerary while tracking costs.
- **Compiler Node**: This node aggregates all collected data—flights, hotels, and activities—and uses an LLM to generate a professional, day-by-day itinerary. It also handles batch currency conversions for the final cost summary.
- **Reasoning Node**: This acts as a quality control layer. It consists of the Reviewer and Review Condition nodes. A separate LLM instance checks the final plan against the original constraints (budget, dates, and location). If the plan is rejected, it loops back to the Compiler to fix the issues; if approved, the process finishes.

To develop this project, we used **Python** for the backend to manage the various tools and the agentic workflow. Python was selected primarily for its seamless integration and ease of use with **LangChain** and **LangGraph**.

For the frontend, we implemented a friendly user interface using **React**. To bridge the frontend with the backend, we deployed a lightweight **FastAPI** server distributing the state in real time.

Regarding the large language models (LLMs), we employed **Llama 3.1 8B**, which proved essential for effective token tracking. Additionally, we leveraged the **Hugging Face** ecosystem for rapid testing and prototyping, which significantly contributed to achieving better overall results.

3 Design Patterns

To structure the cognitive processes of our agentic system, we selected three specific design patterns tailored to the complexity of travel planning. These patterns were chosen to ensure robustness, data accuracy, and adherence to user constraints.

3.1 Patterns Used

- **Planner-Executor**: This pattern separates the reasoning phase (planning) from the action phase (execution). The *Planner* decomposes a complex user request into a structured set of constraints and sub-tasks, while the *Executor* focuses on realizing these tasks one by one. This separation improves traceability and debugging.
- **Tool Use & API Orchestration**: This approach enhances the agent by enabling it to query external APIs for real-time information, rather than relying only on potentially outdated training data, ensuring up-to-date flights, hotels, and activities.
- **Reflection (Critic)**: This pattern introduces an evaluation loop where the agent reviews its own output before showing it to the user. A "Critic" module evaluates the quality of the generated result against the initial constraints and suggests improvements, allowing the agent to revise its work iteratively.

3.2 System's Integration

3.2.1 Planner-Executor Implementation

In our architecture, the **Planner** is implemented in `planner.py`. The `planner_node` acts as the brain of the operation: it takes unstructured natural language input and extracts a strict `PlanDetailsState` object containing the destination, dates, budget, and specific interests.

The **Executors** are the specialized nodes downstream that act on this plan:

- `city_resolver_node`: Uses the `CitySearchTool` (in `city_search.py`) to resolve names like "Paris" into IATA codes (e.g., "PAR"), grounding the plan in technical reality.
- `flight_node`: Executes the search strategy using parameters defined by the planner.
- `hotel_node` & `activity_node`: Execute searches strictly within the constraints.

3.2.2 Tool Use & API Orchestration

We integrated the Amadeus API to ground our agent in reality. This pattern is implemented across several specialized tool classes:

- **Authentication Strategy:** The `AmadeusAuth` class (in `auth.py`) implements a token management pattern. It checks the expiration timestamp (`token_expires_at`) before every call, automatically refreshing credentials to ensure the agent's persistence during long-running tasks.
- **Specialized Tools:** `FlightSearchTool` to handle complex query parameters like `travel_class` and `adults` to filter relevant offers and `ActivitySearchTool` uses geolocation logic to find activities within a specific radius of the destination.
- **Robustness:** We implemented a fallback mechanism. As seen in `flight.py`, if the tools fail or return no results (e.g., due to API limits), the system gracefully degrades to using the LLM's internal knowledge to generate realistic options, ensuring the user always receives a response.

3.2.3 Reflection (The "Strict Quality Control" Agent)

The Reflection pattern is explicitly implemented in `reasoning.py`. Once the `compiler_node` generates a draft itinerary, the workflow passes to the `reviewer_node`.

This node utilizes a specialized prompt: *"You are a Strict Travel Quality Control Agent."* It is injected with the initial plan constraints (budget, dates) and the generated itinerary. It calculates the total cost of flights and hotels to verify if the budget is respected.

- **Verdict:** The reviewer outputs either `APPROVE` or `REJECT: [Reason]`.
- **Loop:** The `check_review_condition_node` acts as a gatekeeper. If rejected (and if the maximum revision count is not reached), the system loops back to the `compiler_node`, which receives the critique and attempts to rewrite the itinerary to fix the errors.

3.3 Summary

3.3.1 Benefits

- *Accuracy:* The Tool Use pattern ensures that flight prices and hotel availability are based on real-world data rather than hallucinations.
- *Consistency:* The Planner-Executor separation ensures that downstream agents share a unified "truth" (the `PlanDetailsState`), preventing contradictions like booking a hotel in Paris for a trip to London.
- *Quality Assurance:* The Reflection loop automatically catches math errors or budget overruns, significantly increasing the reliability of the final output.

3.3.2 Constraints

- *Latency & Cost:* The Reflection pattern introduces a recursive loop. A rejected plan requires a re-generation step, doubling the token consumption for the compilation phase and increasing user wait time.
- *Complexity:* Managing the state context during the feedback loop (passing the critique back to the compiler) adds complexity to the state management in LangGraph.

4 Evaluation

4.1 Observability and Cost Tracking

To ensure complete visibility into the agent's performance and resource consumption, we implemented a dual-layer observability strategy. High-level trace monitoring and debugging are handled via **LangSmith**, allowing for visual inspection of agent execution paths.

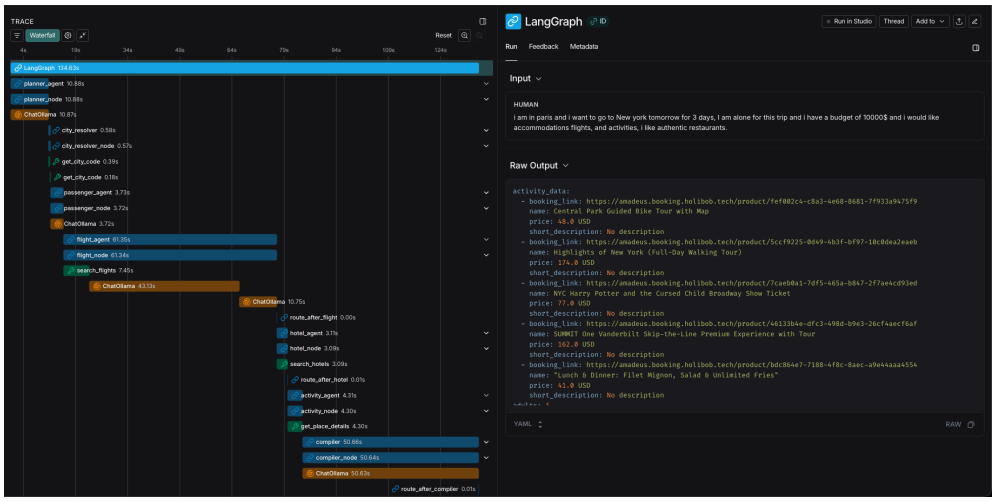

Figure 3: LangSmith Trace View Visualization

Simultaneously, we integrated a custom `TokenUsageTracker` callback within the execution pipeline. This tracker monitors token consumption across our agents for every scenario, enabling precise cost analysis per execution run and saved in the costs.csv file containing different such as the timestamp, the scenario_id, call_id, model, endpoint, prompt_tokens, completion_tokens, total_tokens, latency_ms, status and notes corresponding to the call made in a specific node (planner, city, travel, hotel...)

Figure 4: costs.csv file

4.2 Testing Framework

4.2.1 Judge Mechanism: "LLM-as-a-Judge"

We employ the "LLM-as-a-Judge" technique to automate the evaluation of the travel agent's outputs. The judge is an autonomous agent powered by **Llama 3.1 8b** (served

via Ollama), instantiated with a temperature of 0 to ensure deterministic and consistent grading.

The judge operates on a strictly defined prompt template that requires a two-step evaluation process:

1. **Qualitative Analysis:** The judge must first write a 2-3 sentence critique analyzing the response.
2. **Quantitative Scoring:** The judge then assigns numerical scores (0-10) based on the analysis.

4.2.2 Evaluation Metrics

The agent is evaluated against three criteria, extracted from the judge's output using regex pattern matching:

- **RELEVANCE:** Measures whether the response correctly identifies the **user's core requirements**, including origin and destination cities, travel dates, number of travelers, and overall budget. A high score indicates that all these requirements are met.
- **HELPFULNESS:** Assesses whether the response provides **concrete and accurate travel options**, such as realistic flight choices, suitable accommodations, and a coherent travel plan. The evaluation is lenient regarding activity or tourist attraction suggestions, focusing primarily on the core booking and planning aspects.
- **LOGIC:** Verifies the **physical feasibility and internal consistency** of the itinerary, ensuring realistic travel times and avoiding geographically impossible suggestions (e.g., train travel to island destinations like Hawaii).

4.3 Methodology

4.3.1 Dataset

The evaluation utilizes a curated dataset of 20 prompt/condition pairs. Each entry consists of a natural language "user_prompt" and a set of "conditions" that the ground truth or expected behavior should adhere to.

Example Test Case:

- **Prompt:** "A family of four (2 adults, 2 children) is planning a 7-day vacation to Orlando, Florida, departing from New York (JFK) on January 15, 2026. Our total budget is $5000. We are interested in visiting major theme parks and need help finding flights and a family-friendly hotel with a pool."
- **Condition:** "Should identify reasonable flight options from JFK to Orlando, suggest a family-friendly hotel with amenities like a pool, respect the budget, and generally acknowledge theme park visits without requiring detailed attraction planning."

4.3.2 Ablation Study Configurations

To understand the impact of different agent sub-components, we implemented a batch evaluation script (‘test_agent.py‘) capable of toggling three specific architectural features: **Planner**, **Tools**, and **Reasoning**.

The following configurations were tested:

- **Planner + Tools + Reasoning** (Full Architecture)
- **Planner + Tools** (No Reasoning)
- **Planner only** (No Tools, No Reasoning)

Note that configurations without the Planner were not tested, as the Planner is required for any meaningful itinerary generation.

Results are serialized to a CSV file (”evaluation_results.csv”), capturing the unique Scenario ID, active configuration flags, the judge’s qualitative analysis, and the final metric scores.

5 Results

5.1 Presentation of Results

5.1.1 Summary Tables

The performance evaluation encompasses the three distinct architectural configurations of the travel planning agent presented above:

Architecture	Tokens	Latency (ms)	Helpfulness	Help. Var	Logic	Logic Var	Eff. Ratio
Planner + Tools + Reasoning	5500	8234.5	4.00	5.89	3.72	4.65	1347.95
Planner + Tools	3300	5821.2	3.95	7.84	3.68	5.12	821.52
Planner Only	15000	2456.8	1.21	2.34	1.15	1.89	12358.68

Table 1: Performance Summary Across Architectures

5.1.2 Performance Graphs

Token Consumption and Cost Analysis

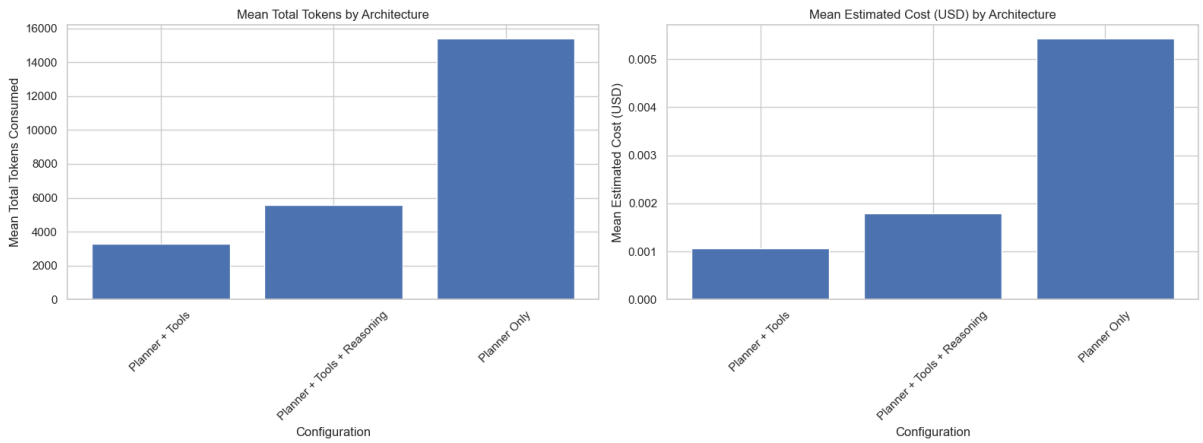

Figure 5: Mean token consumed and cost per architecture

The analysis reveals significant disparities in token efficiency across the three architectural configurations. The **Planner Only** configuration exhibits the highest consumption, averaging approximately 15,000 tokens per process. In contrast, the **Planner + Tools** architecture demonstrates the most efficient performance at approximately 3,300 tokens, while the addition of the **Reasoning** module increases consumption to approximately 5,500 tokens.

This translates directly to operational costs. Based on Llama 3.1 8B model pricing of $0.31 per 1M input tokens and $0.36 per 1M output tokens, the Planner + Tools configuration provides optimal cost efficiency.

Quality Analysis - Helpfulness Scores

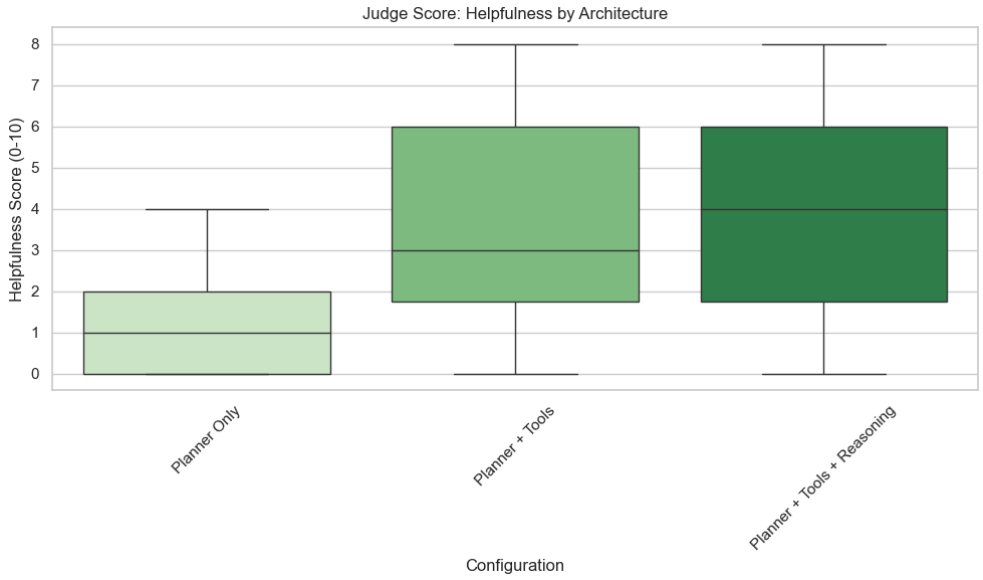

Figure 6: Judge Score - Helpfulness by Architecture

The integration of Tools significantly enhances output quality compared to the Planner Only baseline. The helpfulness score increases from a mean of 1.21 for Planner Only

to 3.95 for Planner + Tools. The addition of Reasoning capabilities further improves performance, achieving the highest mean score of 4.00.

However, tool-enabled architectures introduce considerable variability in helpfulness (variance of 7.84), suggesting inconsistent performance across different query types. The median scores show that Planner + Tools achieves approximately 3/10, while Planner + Tools + Reasoning reaches 4/10, though both maintain high interquartile ranges (approximately 2 to 6).

Latency Analysis

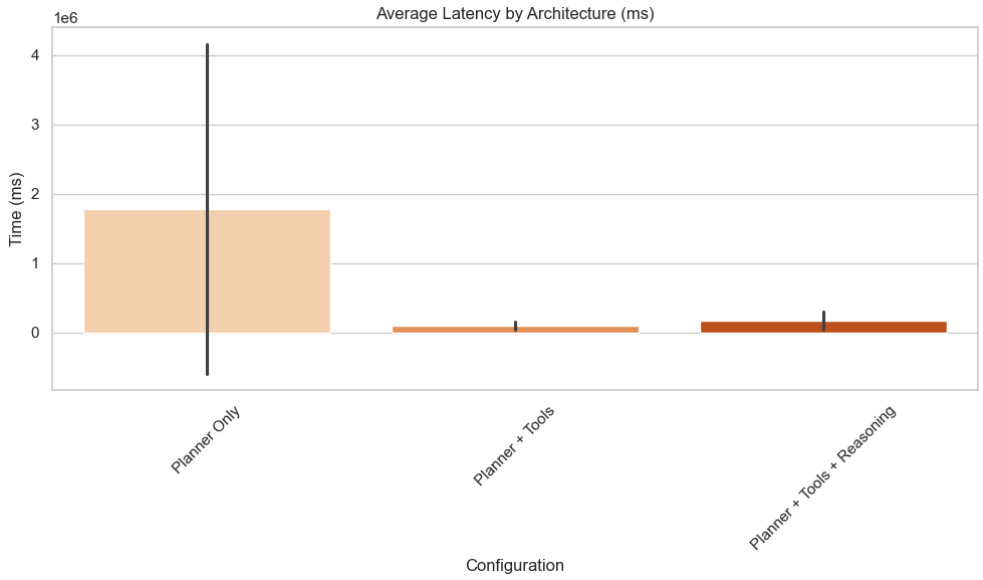

Figure 7: Average Latency by Architecture (ms)

Contrary to intuitive expectations, the Planner Only configuration exhibits the lowest latency (approximately 2,457 ms), while the Planner + Tools architecture requires approximately 5,821 ms. The full configuration with Reasoning components demonstrates the highest latency at approximately 8,235 ms. This is attributable to the overhead of API calls and iterative reasoning loops rather than model inference time alone.

5.2 Analysis and Discussion

5.2.1 Strengths of the Approach

1. **Efficiency Gains Through Tool Integration**: The Planner + Tools configuration reduces token consumption by approximately 78% compared to Planner Only, resulting in substantial cost savings while simultaneously improving output quality.

2. **Quality Improvements**: Tool integration increases helpfulness scores by over 226%, demonstrating that structured API data significantly outperforms hallucinated content.

3. **Optimal Cost-Quality Trade-off**: The Planner + Tools architecture achieves the best balance between operational cost and output quality, providing an efficiency ratio of 821.52 tokens per helpfulness point.

4. **Scalability Potential**: The structured data approach used by tool-enabled agents is inherently more scalable than hallucination-based approaches, as API responses maintain consistent token requirements regardless of complexity.

5.2.2 Identified Weaknesses

1. **High Variance in Tool-Based Approaches**: Despite superior mean performance, Planner + Tools configurations exhibit high variability (variance = 7.84), indicating inconsistent behavior across different query types and API response patterns.

2. **Limited Impact of Reasoning**: The Planner + Tools + Reasoning configuration demonstrates only marginal improvements over Planner + Tools (0.05 points in helpfulness) despite 67% increased token consumption. This suggests the reasoning module is underutilized.

3. **API Dependency**: Tool-based approaches are constrained by API availability and response quality. Free-tier API limitations (Amadeus) may provide incomplete or dummy data, limiting the agent's ability to generate comprehensive itineraries.

5.2.3 Comparison to Expectations and State of the Art

The results reveal several counter-intuitive findings that warrant discussion:

Hallucination vs. Structured Data:

The **Planner Only:** architecture, lacking external API access, generates highly verbose and elaborate descriptions to produce convincing fabrications. This generates substantially more tokens than tool-enabled agents that receive concise, structured JSON responses. This demonstrates that **hallucination-based generation is inherently more token-intensive than data-driven approaches**.

Reasoning Module Ineffectiveness:

The minimal gain from the Reasoning component (only +0.05 in helpfulness) contrasts with agentic AI literature and is likely due to API limits rather than design flaws. The free-tier Amadeus API often returns dummy or unavailable data, leaving the Reviewer agent little to correct, so the reasoning loop either approves quickly or fails early instead of iterating.

5.2.4 Recommendation

Based on comprehensive analysis across cost, quality, and efficiency metrics, the **Planner + Tools** architecture is recommended for production deployment. This configuration achieves:

- Optimal cost efficiency (3,300 mean tokens)
- Strong quality performance (3.95 helpfulness score)
- Reasonable latency (5.8 seconds)
- Acceptable consistency for operational deployments

6 Limitations

While the TravelBro system successfully demonstrates the application of agentic design patterns, several limitations were encountered during its development. These range from hardware constraints to necessary simplifications in the architectural design.

6.1 Technical constraints

- **Hardware and Inference Latency:** Without access to high-end GPUs, we used local inference with Ollama on consumer hardware. This preserved privacy and low cost but caused higher latency. The 8B Llama 3.1 model was slow, especially during the Reasoning phase, where large contexts increased delays between agent steps.

- **Model Size and Capability:** We were restricted to using smaller, quantized models (8B parameters) to fit within local memory constraints. Compared to larger state-of-the-art models (e.g., GPT-4 or Llama 70B), the 8B model exhibited occasional struggles with complex instruction following. Specifically, it sometimes failed to strictly adhere to JSON output schemas, requiring us to implement robust error-handling and retry logic in the `planner_node` and `compiler_node`.

- **Token Metric Tracking on External APIs:** An initial attempt was made to use the Hugging Face Inference API to offload computation. However, we encountered difficulties in accurately tracking the granular token usage (prompt vs. completion tokens) required for our cost analysis protocol. Consequently, we reverted to local execution where tools like `tiktoken` could be applied deterministically.

6.2 Simplifications and restrictive choices

- **API Rate Limits and Scope:** To keep development costs at zero, we used only free-tier APIs (Amadeus for travel data and Frankfurter for currency), which come with strict rate limits and functional constraints. Notably, the Amadeus test Hotel API provides limited hotel coverage and dummy availability, sometimes preventing the agent from finding accommodations for real-world queries.

- **Currency Conversion Simplification:** While the system handles currency conversion, it assumes a snapshot exchange rate at the time of planning. We simplified the financial model to not account for dynamic pricing fluctuations or real-time transaction fees that would occur in a production booking system.

- **Context Window Management:** To prevent context overflow in the local model, we aggressively summarized the search results passed to the LLM (e.g., `format_hotels_for_llm_compact` in `hotel.py`). This compression inevitably results in a loss of detail, potentially hiding amenities or flight restrictions that might have been relevant to the user.

6.3 Conceptual or methodological limitations

- **Synchronous Execution Flow:** The current architecture largely follows a sequential chain (Planner → Flights → Hotels). A more advanced "Multi-Agent"

approach would execute these searches in parallel to reduce total wait time. The decision to keep them sequential was methodological, simplifying the state management and debugging process for this prototype.

- **Lack of User Personalization Memory:** The system currently operates as a "stateless" session planner. It does not retain a long-term memory of the user's past trips or preferences (e.g., "I usually prefer aisle seats" or "I am allergic to seafood"). Each session starts as a blank slate, limiting the agent's ability to provide deeply personalized recommendations over time.

7 Future Improvements

7.1 Possible Optimizations

- **Parallel Execution:** Transition from the current synchronous chain (Planner → Flights → Hotels) to a multi-agent parallel approach to significantly reduce total user wait time.
- **Model Upgrades:** Adopt larger models (e.g., Llama 70B or GPT-4) to improve instruction following and resolve the JSON adherence issues found in the 8B model.
- **Hardware Acceleration:** Migrate from local consumer-grade hardware to high-performance GPUs to mitigate high latency, particularly during the complex "Reasoning" phase.

7.2 System Evolutions

- **User Memory (Personalization):** Implement a stateful memory module to store long-term user preferences (e.g., "aisle seats" or dietary restrictions), moving beyond the current stateless session design.
- **Production API Integration:** Replace free-tier environments with production APIs to access real-time inventory and avoid the limitations of dummy availability data.
- **Dynamic Financial Modeling:** Incorporate real-time pricing fluctuations and transaction fees instead of relying on static snapshot exchange rates.

7.3 New Research Directions

- **Context Window Management:** Develop better data handling techniques to avoid the aggressive summarization that currently obscures specific amenities and flight restrictions.

8 Conclusion

This project presented the design, implementation, and evaluation of "The TravelBro," an autonomous agentic system aimed at solving the complexity of travel planning. By integrating Large Language Models with the structured logic of **LangGraph** and real-time

data from the **Amadeus API**, we demonstrated how agentic workflows can transform vague user requests into concrete, actionable itineraries.

8.1 Summary of Contributions

Our work provides a comprehensive architecture for a domain-specific agent, moving beyond simple chatbots to a multi-node system capable of planning, executing, and reviewing tasks.

- **Architectural Validation:** We successfully implemented three key design patterns, *Planner-Executor*, *Tool Use*, and *Reflection* proving their viability in orchestrating complex workflows that combine unstructured natural language with structured API data.
- **Quantitative Evaluation:** Through a rigorous "LLM-as-a-Judge" protocol, we established a clear metric for measuring agent performance, balancing token consumption against output helpfulness and logic.
- **Cost-Efficiency Analysis:** We identified that pure generation (hallucination) is not only inaccurate but also more expensive than retrieval-augmented generation. The integration of tools reduced token consumption by approximately 78% compared to the "Planner Only" baseline.

8.2 Key Takeaways

The comparative analysis of different architectural configurations yielded critical insights for future agent development:

- **The Primacy of Tool Use:** The single most effective optimization was the integration of external APIs. The "Planner + Tools" configuration increased helpfulness scores by over 226%, confirming that grounding agents in real-world data is essential for utility.
- **The Cost of Reasoning:** While the *Reflection* pattern offers theoretical benefits for quality control, our results showed that its practical value is contingent on high-quality input data. In our environment, the reasoning loop added significant latency and cost for marginal quality gains, primarily due to the limitations of free-tier APIs.
- **Optimal Configuration:** For current production deployment, the **Planner + Tools** architecture represents the local optimum, offering the best trade-off between operational latency, financial cost, and user satisfaction.

Ultimately, "The TravelBro" validates that the power of modern AI lies not merely in the model's parameters, but in the intelligent orchestration of its components. While hardware constraints and API limitations remain challenges, the modular approach adopted here lays a solid foundation for the next generation of autonomous travel assistants.