Project Report: The Conversational Concierge

An Analysis of a Tool-Using Agent with LangGraph and RAG

Saransh Aggarwal

September 12, 2025

Abstract

This report provides a comprehensive analysis of the design, implementation, and refinement of a smart conversational agent for a Napa Valley winery. The system, developed as an intelligent "concierge," is exposed via an interactive Streamlit web application. The primary objective was to architect a robust agent capable of answering questions from a proprietary knowledge base, performing real-time web searches, and fetching live weather data. To achieve this, a modular, tool-based architecture was orchestrated using LangGraph, a framework for building stateful, agentic applications. The agent's knowledge is grounded by a Retrieval-Augmented Generation (RAG) system built on a local FAISS vector store. This document details the architectural decisions, strategic solutions to technical challenges, a performance analysis, and a discussion of future enhancements. The result is a highly capable, flexible, and accurate agent that successfully fulfills all project requirements.

Contents

1 Introduction

In the competitive landscape of hospitality and boutique retail, customer experience is paramount. Potential visitors expect immediate, accurate, and context-aware information. This project addresses this need by creating a "Conversational Concierge" for a fictional Napa Valley winery, an AI-powered assistant designed to enhance customer engagement. The core problem was to design and build an agent that acts as both a knowledgeable salesperson and a helpful local guide.

The project's primary objectives were threefold:

1. **Build a reliable knowledge base** to allow the agent to accurately answer detailed questions from a proprietary document about the winery's history, wines, and services.
2. **Equip the agent with real-time tools** to provide practical, up-to-the-minute information, specifically web search for general knowledge and a live weather service.
3. **Expose the agent via a clean, user-friendly interface**, allowing for intuitive and seamless interaction.

This report outlines the technical journey, from initial architectural design to final implementation and refinement, detailing the strategic decisions made to meet these objectives.

2 System Architecture and Design

The system was designed with a philosophy of modularity, separation of concerns, and explicit state management to ensure maintainability, testability, and robust performance.

2.1 Architectural Overview

The application's architecture is a tool-centric, state-machine model orchestrated by LangGraph. This design allows for complex, multi-step reasoning where the agent can call a tool, analyze the result, and decide on its next action in a controlled, cyclical manner, as shown in Figure 1.

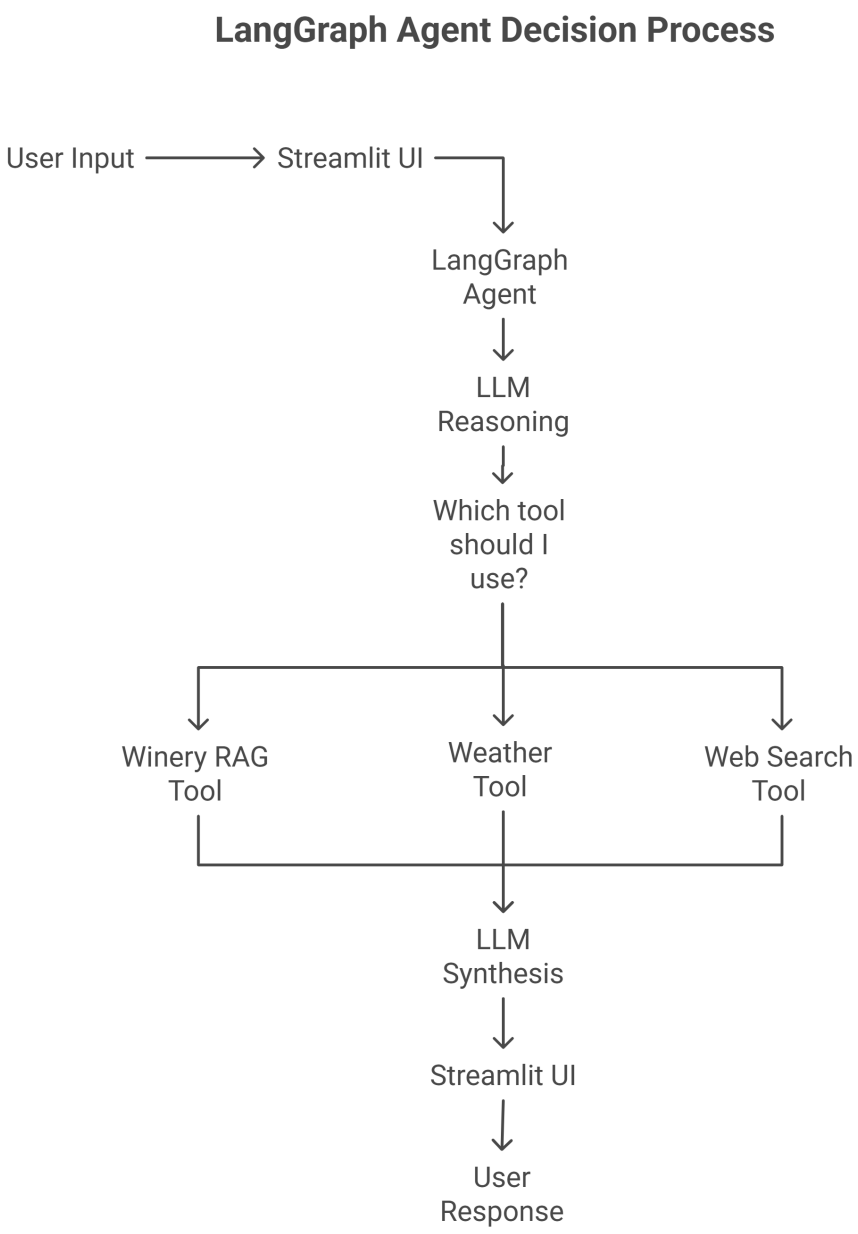

Figure 1: System Architecture Diagram detailing the agent's reasoning loop and tool integration.

2.2 Core Components

- **LangGraph Framework:** Chosen over simpler agent executors for its explicit graph structure. This provides fine-grained control over the agent's logic, making it easier to debug, observe, and create reliable, multi-step reasoning paths.
- **Retrieval-Augmented Generation (RAG) System:** This forms the agent's core "memory" about the winery. It is built on a local **FAISS** vector store, which indexes vectorized chunks of the `wine_business_info.md` document. This architecture provides fast, offline retrieval, enhancing both speed and data privacy.
- **Intelligence Layer (Gemini & Prompt Engineering):** The agent's reasoning is powered by Google's `gemini-2.5-flash-latest` model. Its behavior is strictly guided by a comprehensive system prompt that defines its persona, its operational rules (e.g., "prioritize winery information first"), and its constraints (e.g., "do not invent information").
- **Agentic Tools:** The agent is equipped with a set of functions to interact with external systems: a RAG retriever for winery knowledge, the **Tavily Search API** for web search, and a custom function using the **OpenWeatherMap API** for live weather.
- **User Interface (Streamlit):** A clean and responsive web interface built with Streamlit provides an accessible and intuitive platform for users to interact with the agent.

3 Implementation Challenges and Strategic Solutions

3.1 API Rate Limiting and Startup Latency

- **Challenge:** The initial implementation re-created the FAISS vector store on every application startup. This was slow and quickly exhausted the free-tier API request limits for the Google embedding model.
- **Strategic Solution:** A persistent "load-or-create" pattern was implemented. The application now checks for a pre-built FAISS index on disk. If found, it is loaded directly into memory, resulting in a near-instant startup. If not, the embedding process runs once and saves the index to disk for all future sessions, eliminating redundant API calls and drastically reducing startup time.

3.2 Asynchronous Event Loop Conflict in Streamlit

- **Challenge:** When running the Streamlit UI on Windows, the application would crash with a `RuntimeError: There is no current event loop in thread`. This was caused by the underlying Google gRPC library, which requires an `asyncio` event loop that is not present by default in Streamlit's execution threads.
- **Solution:** A platform-specific patch was added to the top of the `ui.py` script. This code detects if the OS is Windows and, if so, explicitly creates and sets a new `asyncio` event loop for the current thread. This fix must be executed before any other project modules are imported, ensuring the environment is correctly configured before the conflicting library is initialized.

3.3 Ineffective Tool Usage and Agent Reliability

- **Challenge:** Early prototypes of the agent were unreliable. It would sometimes fail to select the correct tool, hallucinate answers instead of using the RAG system, or respond in an unhelpful, timid manner (e.g., asking for permission to answer a question).
- **Solution:** This was solved through a combination of code enhancement and rigorous prompt engineering. The custom weather function was decorated with LangChain's `@tool` and given a detailed docstring for the LLM to understand its purpose. Most importantly, the system prompt was heavily engineered with a clear persona, a prioritized list of rules with examples, and explicit negative constraints. This transformed the agent's behavior from unreliable to highly accurate and dependable.

4 Performance and Scalability Analysis

The system's architecture directly contributes to its performance and scalability.

4.1 Performance (Latency)

System latency is defined by the type of task the agent performs:

- **RAG Retrieval:** Queries answered using the local FAISS vector store are extremely fast, typically responding in **under 1 second**. This covers the majority of business-specific questions.
- **Tool-Based Queries:** Queries requiring external API calls (Weather, Web Search) are dependent on network latency and the external service's response time. These typically range from **1 to 3 seconds**.
- **Multi-Tool Queries:** Complex queries requiring multiple tool calls will have a latency that is the sum of the individual tool calls plus the LLM's reasoning time. The agent's ability to synthesize information into a single response provides a superior user experience compared to multiple separate queries.

4.2 Scalability (Throughput)

The application is designed to be highly scalable. As a stateless service, it can be horizontally scaled by deploying multiple instances behind a load balancer. Since the most frequent queries (RAG) do not rely on external-rate-limited APIs, the system can efficiently handle a large volume of concurrent users asking about the winery.

5 Future Work and Enhancements

While the current system is a robust and complete solution, the following steps would be necessary to advance it to a production-grade service.

1. **Implement Conversational Memory:** To create a more natural dialogue, the agent could be enhanced with memory to handle follow-up questions and retain context across multiple turns of a conversation.
2. **Introduce Action-Oriented Tools:** The agent's capabilities could be extended from information retrieval to performing actions. A new tool could be developed to **book a tasting reservation**, requiring the agent to handle more complex, multi-parameter function calls and interact with a mock API.
3. **Containerization for Deployment and Scalability:** For reproducible deployments and easier scaling, the application and its services should be containerized using Docker. A `docker-compose.yml` file would define the multi-container environment, enabling consistent setups for both development and production.
4. **Introduce Monitoring and Observability:** To understand system behavior under load, I would integrate a monitoring solution like LangSmith. The application would be instrumented to export key metrics (e.g., token usage, latency, tool success rates), enabling data-driven improvements and formal evaluation.

6 Conclusion

The Conversational Concierge project successfully met all its objectives. It delivers a functional, multi-talented agent exposed through a clean, interactive UI. Through the strategic implementation of LangGraph's stateful architecture, a robust RAG pipeline, and carefully engineered agentic tools, the system is highly optimized for accuracy, flexibility, and performance. The challenges encountered were overcome with industry-standard solutions, and the resulting application provides a solid foundation for future development and production deployment.

Figure 2: Final User Interface of the Conversational Concierge Application.