# AI Scheduling Agent - Technical Approach

**Candidate Name:** Pardhesh Maddala

## Architecture Overview: Agent design and workflow

The system is designed as a decoupled, two-tier architecture.

• **Backend Agentic Workflow:** The core logic is implemented as a collection of specialized Python modules, referred to as "agents," located in the src/agents/ directory. Each agent is responsible for a specific task (e.g., PatientLookupAgent, AppointmentAgent, ReminderAgent).

• **Conversational UI Layer:** User interactions are managed through a conversational chatbot built with Streamlit (app.py). This UI layer orchestrates the entire booking process.

The complete workflow proceeds as follows:

**1.** **Greeting & Information Extraction:** The assistant greets the user and employs the PatientInfoAgent (powered by a Llama model via LangChain with Groq).

**2.** **Patient Lookup & Branching:** The PatientLookupAgent searches the patient database for an existing record. If the patient is new, the system collects insurance details; returning patients proceed directly to scheduling.

**3.** **Scheduling & Confirmation:** The AppointmentSchedulerAgent identifies an available slot based on business rules (60 minutes for new patients, 30 minutes for returning).

**4.** **Final Decision & Execution:** The user is prompted to either confirm or cancel the appointment. Their decision triggers the ConfirmationAgent, which logs the booking, updates the database, and sends an SMS via Twilio.

**5.** **Reminder Agent:** A standalone script that executes every six hours. It monitors patient\_status.xlsx to send appropriate reminders based on each patient’s DOA.

## Framework Choice: LangGraph/LangChain vs ADK with justification

This project adopts a hybrid approach combining LangChain, LangGraph, and Streamlit.

• LangChain was used for the PatientInfoAgent to structure prompts, integrate with the Groq LLM, and generate structured JSON outputs for downstream agents.

• LangGraph flow (patient\_flow\_graph.py) was developed for CLI-based execution, while the main conversational flow was built directly in Streamlit.

• Streamlit’s session\_state provided granular control for turn-based interactions, enabling the chatbot to pause, prompt, and await user input.

• This design ensures flexibility and a natural user experience, avoiding the rigidity of fully sequential frameworks like ADK.

## Integration Strategy: How you handled data sources

The system is designed to simulate a scheduling system similar to Calendly, leveraging Excel and CSV files to emulate a real-world clinical environment. The following data sources used:

• Patient Database (synthetic\_patients.csv): Simulates an Electronic Medical Record (EMR).

• Doctor Schedules (doctor\_schedule.xlsx): Serves as the shared calendar. The AppointmentSchedulerAgent checks availability and updates as Booked by{patient\_name}.

• Admin Reporting (patient\_status.xlsx): Functions as a persistent admin dashboard. The ConfirmationAgent records the final status of all completed bookings in this file.

• External APIs (Twilio & SendGrid): Twilio handles transactional SMS (confirmations and cancellations), while SendGrid delivers emails containing the intake form as a PDF .

## Challenges & Solutions: Key technical decisions made

**1.** **Branching Logic for Patient Types**

**Challenge:** The system needed to handle different conversational flows and business rules for new and returning patients (e.g., insurance details required only for new patients, longer appointment slots).

**Solution:** Implemented a robust state machine within app.py. The PatientLookupAgent output (status: 'new' or 'returning') is stored in session\_state and used to guide the conversation along the correct path, ensuring accurate data collection.

**2**. **Correct and Consistent Data Handling**

**Challenge:** With multiple data sources and agents passing information, risks included data loss like, patient details for returning patients and crashes due to inconsistent columns.

**Solution:** Enhanced agents to be data-resilient. For example, the ConfirmationAgent retrieves patient details from state regardless of how they were passed. Flexible column resolvers were implemented to handle variations names (e.g., 'Doctor' vs. 'Doctor Name').

**3. Setting Up the Reminder System**

**Challenge:** Unlike the booking flow, reminders are asynchronous and time-based, requiring a different architecture. Integrating into the main app would reduce modularity.

**Solution:** Developed reminder\_agent.py as an independent script running on a schedule (via APScheduler). It periodically reads patient\_status.xlsx to identify and send due reminders. This separation ensures a clean, scalable architecture by decoupling the interactive booking flow from asynchronous notification processes.