🧠 Objective

The primary objective of this project was to design and develop a **conversational sales chatbot** for an e-commerce platform. The chatbot enables users to interact naturally—searching, filtering, and purchasing products like books—through a **chat-based interface**.

🧪 Methodology

A **full-stack solution** was implemented using:

- **Frontend**: React (JavaScript)
- **Backend**: Flask (Python)
- **Database**: SQLite with mock data

Key techniques:

- **User Input Parsing**: Regular expressions and keyword extraction.
- **State Handling**: Managed using React's `useState` / `useEffect`.
- **Communication**: RESTful APIs powered by Axios.

The UI supports responsive design, login/logout flows, cart management, and order history.

✅ Result

The chatbot allows users to:

- Search books by genre, title, or price range.
- Manage cart items (add, remove, update).
- Place orders via conversational commands.

Sample supported queries:

- **"Show me thrillers under 500"**
- **"Add The Hobbit, quantity 2"**

These were interpreted accurately, demonstrating strong **real-world applicability**.

🎓 Key Learnings

- How to embed **NLP-like logic** in e-commerce chat workflows.
- Managing **stateful interactions** between chatbot and UI.
- Building smooth **API communication** between frontend and backend.
- Handling edge cases like **context loss** and vague commands through modularity and graceful error handling.

🏗️ 1. Architecture Overview

This project is a **chatbot-based e-commerce prototype** built with a **full stack architecture** using **React** for the frontend and **Flask** for the backend.

🔧 Key Components

- **Frontend**: Built using **React.js** for a responsive, component-driven interface.
- **Backend**: Powered by **Flask**, handling API endpoints and chatbot logic.
- **Database**: Uses **SQLite with mock data** for the prototype; designed to support easy migration to **MongoDB** or **PostgreSQL**.
- **State Management**: Managed using React's built-in `useState` and `useEffect` hooks.
- **Communication**: Frontend and backend communicate through **RESTful APIs** using **Axios**.

🛠️ 2. Tools & Technologies

Technology	Role	Why It Was Chosen
React.js	Frontend UI	Fast, modular, and ideal for building dynamic interfaces like a chatbot.

Flask	Backend Server	Lightweight and Python-based — great for quickly integrating chatbot logic.
Axios	API Calls	A simple and reliable HTTP client for React apps.
Python	Backend Language	Easy syntax, rich ecosystem, and perfect for scripting chat behavior.
Mock Data	Product & Chat Simulation	Speeds up development by eliminating the need for a live database in early stages.

📚 3. Mock Data Generation

To simulate a real bookstore experience, mock book data is generated and inserted into the SQLite database using Python and the **Faker** library. Here's how the process works:

1. **Setup**

   The script establishes a connection to the database and prepares for insertion.

2. **Generate Random Books**

   A loop creates fake book entries with random titles, authors, genres, prices, stock levels, and ISBNs.

3. **Insert Into Database**

   Each book is inserted into the `products` table. Duplicate ISBNs are skipped to maintain uniqueness.

4. **Genre Coverage**

   One additional book for each major genre (e.g., Fiction, Thriller, History) is added to ensure broad coverage.

5. **Final Step**

   All changes are committed and the connection is safely closed.

⚠️ 4. Key Challenges & How They Were Tackled

🧩 Challenge 1: Designing Chat Message Flow

- **Issue**: Early on, it was unclear **how to parse user messages** — whether to rely on keyword matching, predefined intents, or integrate an NLP library.
- **Why it mattered**: A rigid approach (e.g., simple `if-else` or keyword checks) made the chatbot brittle and unable to handle variations in user queries.
- **Solution**:
  - Started with simple keyword-based routing.
  - Then refactored into a **modular message processing function** with basic intent recognition logic (e.g., detecting "search," "buy," "add to cart," "checkout").
  - Future iterations could plug in NLP libraries like spaCy or transformers for better understanding.

🧩 Challenge 2: Avoiding Repetition in Chat Responses

- **Issue**: The chatbot often replied with robotic or repetitive messages like "I don't understand" or "Here are the books."
- **Solution**:
  - Created a **response pool** with varied phrases for the same intent.
  - Used simple logic to **randomize responses** while still keeping tone consistent.

🧩 Challenge 3: Delays Causing UI Freezes

- **Issue**: Simulated delays (to mimic chatbot "thinking") caused the UI to feel unresponsive.
- **Solution**: Used `setTimeout()` in React to delay chatbot replies **without blocking** UI rendering:

```
setTimeout(() => {
  setMessages([...messages, response]);
}, 500);
```

🧩 Challenge 4: Scaling Mock Data

- **Issue**: Managing mock data in static Python files wasn’t scalable as product data grew.
- **Solution**: Switched to storing mock data in a SQLite database and implemented frontend-side filtering/pagination. For larger-scale needs, transitioning to **MongoDB** is planned.