# Chatbot Project – Detailed Technical Documentation

## 1. Purpose of This Document

This document provides a narrative, code‑free explanation of how the real‑time, WebSocket‑based chatbot was constructed, why each technology was chosen, and how the core logic works end‑to‑end. It is intended for developers, DevOps engineers, and stakeholders who want to understand the design rationale without diving into source code.

## 2. Problem the Project Solves

Educational institutions often need an interactive system capable of instantly retrieving student information such as attendance, GPA, or internship status while also answering general knowledge or aptitude questions. Traditional REST APIs require polling and provide little conversational context. This project introduces a conversational interface that offers real‑time responses over WebSockets, marrying an AI agent with live database access.

## 3. High‑Level Architecture Explained

The architecture is deliberately layered to separate concerns and facilitate horizontal scaling:

• \*\*Daphne (ASGI Server)\*\*: Accepts HTTP and WebSocket traffic and hands it over to Django Channels. Daphne was chosen because it natively supports the ASGI specification, allowing asynchronous handling out of the box.  
• \*\*Channels Routing Layer\*\*: Directs WebSocket requests to the appropriate consumer class. It provides group broadcasting, enabling multiple clients to subscribe to the same chat room.  
• \*\*ChatConsumer\*\*: A subclass of `AsyncWebsocketConsumer` that maintains the client connection, forwards incoming messages to the AI layer, and streams answers back. The choice of a consumer rather than a standard Django view is essential for bidirectional, long‑lived WebSocket sessions.  
• \*\*LangChain AgentExecutor\*\*: Acts like a miniature orchestrator. It reads the user utterance, decides whether it needs to call an external tool, and merges the tool result with LLM reasoning. LangChain was selected to avoid writing a custom planner from scratch.  
• \*\*Custom Tools\*\*: Thin wrappers that expose database queries (e.g., fetching student records) as callable functions the agent can trigger. This keeps SQL and ORM logic out of the LLM prompt yet gives the model structured access to data.  
• \*\*PostgreSQL\*\*: Stores authoritative student data. The relational model simplifies joins among attendance, grades, internships, and performance metrics.  
• \*\*OpenAI API\*\*: Provides generative reasoning to interpret user intent, format responses, and perform small‑talk or GK answers when no tool is required.

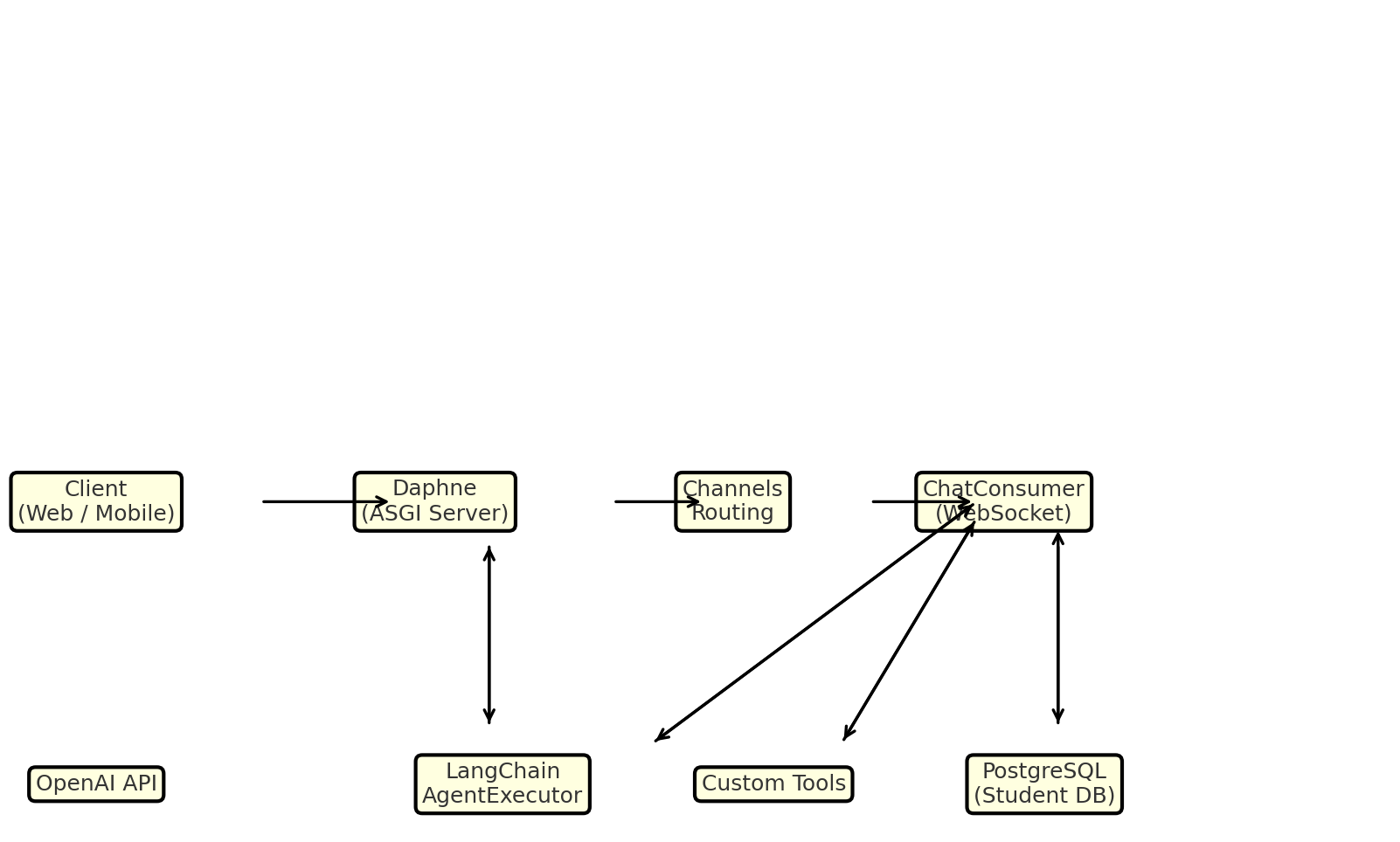


Figure 1 – Logical architecture and data flow.

## 4. Step‑by‑Step Request Flow

1. \*\*Client sends JSON message\*\* over `/ws/chat/` containing the user’s natural‑language question.  
2. \*\*ChatConsumer catches the message\*\*, stores it in an in‑memory history, and forwards the content to the LangChain AgentExecutor.  
3. \*\*AgentExecutor consults its tools list\*\* and the user prompt. It determines whether direct knowledge suffices or a live database lookup is necessary.  
4. \*\*If a lookup is needed\*\*, the relevant custom tool performs an asynchronous Django ORM query (e.g., count total students) and returns a structured Python object.  
5. \*\*LLM receives both the original question and the tool result\*\*, formats a succinct JSON answer, and hands it back to the consumer.  
6. \*\*ChatConsumer emits the response\*\* via the WebSocket group so every participant in the chat room instantly sees the update.

## 5. Rationale Behind Each Custom Tool

• \*\*get\_student\_details\*\* – Consolidates all personal and academic fields a serializer can produce. Centralising the details prevents repetitive ORM calls.  
• \*\*get\_student\_records\*\* – Returns a raw model instance for internal logic when only the primary key is required (e.g., to chain further queries).  
• \*\*count\_total\_records\*\* – Offers a single switchboard for aggregate counts instead of multiple one‑off count queries.  
• \*\*failed\_students / topper\_students\_list\*\* – Pre‑filters students by performance criteria, making the LLM’s life easier because it receives ready‑made lists.  
• \*\*get\_student\_session\*\* – Normalises access to four separate related tables (attendance, grades, internships, performance) under one semantic umbrella called ‘session’.

## 6. Handling Real‑World Pitfalls

During integration, two argument‑mismatch errors surfaced:  
• \*\*topper\_students\_list\*\* was inadvertently registered through a lambda, causing LangChain to inject an extra parameter. Removing the lambda stopped the unwanted argument.  
• \*\*get\_student\_session\*\* originally exposed only one parameter but the tool definition expected two. Aligning the signature with `(student\_name, session)` resolved the runtime crash.

## 7. Deployment & Operations

• \*\*Horizontal Scaling\*\* – Because Daphne and Channels are stateless, you can add replicas behind a load balancer, using Redis as the channel layer for fan‑out.  
• \*\*CI/CD\*\* – A GitHub Actions workflow builds the Docker image, runs migrations, triggers a blue‑green rollout on AWS ECS, and performs a post‑deployment smoke test on `/ws/chat/`.  
• \*\*Observability\*\* – Structured logs emit request IDs from the consumer to each database query, so Datadog can trace end‑to‑end latency.

## 8. Conclusion

By combining WebSockets, Django Channels, and LangChain, the system delivers instant, context‑aware answers while preserving the robustness of a typed relational database. The tool pattern isolates business data from conversational reasoning, drastically reducing bug surface area and making future feature additions (e.g., new session types or analytics) straightforward.