ChatGPT

AI Chatbot Web Application Project Plans (Pharma/Medical & Small Business)

Below are five comprehensive project plans for **mock web applications** that showcase JavaScript development and AI chatbot integration (using **Ollama** for local LLMs) in a developer portfolio. Each project is targeted to either a **pharma/medical** use case or a **small business** scenario, with a unique branding identity. The plans cover the full-stack implementation—from frontend and backend through to automation workflows and deployment—assuming an advanced five-person dev team (including two AI engineers). The applications are not production-grade in terms of compliance, but they simulate realistic behavior for demonstration. Each project plan includes:

- **Use Case & Overview:** The specific problem or scenario addressed (pharma/medical or small business focus).
- **Branding & Design:** Unique project name, color palette, font choices, and UI/UX design direction.
- **Technology Stack:** Frontend (React/Next.js), backend (Node.js/Express with Docker Compose), PostgreSQL database, Ollama chatbot integration via API, **n8n** automation via webhooks, and deployment details (Render.com configuration).
- **Architecture Overview:** Breakdown of frontend components, backend services (including an MCP logic layer), integration points with Ollama and n8n, and a high-level PostgreSQL schema outline.
- **Development Steps:** Step-by-step guide from project initialization (`git init`) to setting up the repository structure, CI/CD pipeline, secrets management on Render, Docker Compose configuration, and repository conventions.
- **Workflow Description:** How users interact with the chatbot UI, how the backend processes requests, when/how the Ollama LLM and n8n flows are triggered, with example automation flows and sample bot prompts.

Each project plan is detailed (~4+ pages each) to serve as a portfolio piece demonstrating proficiency in full-stack development and AI integration.

Project 1: MediConnect AI – Patient Support Chatbot (Medical)

Use Case & Target Audience: MediConnect AI is a medical assistant chatbot designed for a small clinic or telehealth provider. It assists patients in answering general health questions, triaging symptoms, and helping schedule appointments or follow-ups. The use case focuses on providing quick, AI-driven guidance to patients (pharma/medical domain) while streamlining clinic operations. For example, a patient can ask about common cold symptoms and receive advice, or request an appointment which the chatbot helps facilitate. This improves patient engagement and reduces routine call volume for the clinic. (Note: for portfolio purposes, it simulates healthcare advice with disclaimers, without needing full HIPAA compliance.)

Branding & Design: The project's branding is friendly and reassuring, reflecting healthcare professionalism:
- **Project Name:** *MediConnect AI* (tagline: "Your Personal Health Assistant"). The name emphasizes connecting patients with medical guidance.

- **Color Palette:** Cool and calming tones (e.g. **#3A7BD5** a trust-inspiring blue, **#29B6AF** a teal for health, with neutral white backgrounds). These colors convey trust, calmness, and clarity, aligning with medical themes.
- **Fonts:** A clean sans-serif font like *Roboto* or *Open Sans* for readability, paired with a slightly softer sans-serif for headings (like *Ubuntu* or *Helvetica Neue*) to appear modern yet approachable.
- **Design Direction:** The UI is clean and user-friendly. It features a chat interface on the clinic's website/app with rounded chat bubbles and medical iconography (e.g. a stethoscope icon for the assistant). The design follows accessibility guidelines (high contrast text for readability and simple language). The overall look is **professional but empathetic** – patients should feel at ease using the chatbot. Key pages include the chat screen and possibly a patient info form or a symptom checker form that can be optionally filled to assist the chatbot. The chatbot's avatar might be a friendly medical assistant icon (like a chat bubble with a cross symbol).

Technology Stack: MediConnect AI uses a modern JavaScript full-stack with a focus on reliability and integration capabilities:
- **Frontend:** Built with **Next.js (React framework)** for its ease of creating interactive UIs and built-in API routes. Next.js is used to create a single-page chat application with server-side rendering for fast initial loads. React handles dynamic chat updates in real-time (possibly using a state management library like Zustand or Redux for chat state). UI components could be built from scratch or with a library like **Chakra UI** for accessible components (styled according to the branding).
- **Backend:** A **Node.js** backend using **Express** (or Next.js API routes) handles API requests from the frontend. The backend exposes RESTful endpoints (or GraphQL, but REST is sufficient) for sending user messages and retrieving chatbot responses. The backend also contains the **MCP logic layer** – the orchestrating service that manages context and calls out to external services. (MCP, or *Model Context Protocol*, is a standardized layer for connecting AI models with tools [1]. In this project, the MCP logic layer will manage the context given to the LLM and coordinate tasks like database access or triggering n8n workflows.) This separation ensures that the Express routes remain thin controllers, and the MCP service encapsulates the intelligence of when to query the LLM vs. trigger automation.
- **AI Integration (Ollama):** The chatbot intelligence is powered by an **Ollama** local LLM model (e.g. an instance of Llama 2 or a medically fine-tuned model) running on the backend server or a dedicated AI worker. **Ollama** is an open-source tool for running LLMs locally [2]. The backend uses the Ollama API (via the `ollama` JavaScript client library) to send user prompts and receive AI-generated responses. (Ollama's JS client allows the Node app to interact with the local LLM server seamlessly [3].) The model can be configured with a system prompt that introduces it as a helpful but not definitive medical assistant, including a disclaimer to seek a doctor for serious concerns.
- **Automation (n8n):** Some user requests trigger **n8n** workflows through webhooks. **n8n** (running externally, e.g. a cloud or separate docker service) is a workflow automation tool that can perform actions like sending emails or scheduling tasks without coding. MediConnect AI uses n8n for tasks such as appointment scheduling: when the chatbot gathers necessary info from a patient (preferred date/time, symptoms), the backend calls an n8n webhook URL to create an appointment entry or send a notification to the clinic staff. The n8n flows could also handle sending follow-up emails (e.g. a summary of advice or a confirmation of appointment). *Webhooks* allow our app to trigger n8n flows via HTTP requests [4], making integration straightforward.
- **Database: PostgreSQL** serves as the database via a Docker container (for dev) and a managed instance on Render (for production). The DB stores persistent data: e.g. user profiles (if patients can log in), chat transcripts or key points from conversations, a table of available appointment slots or doctors, and logs of any actions (for auditing what advice was given). Storing conversation history can help provide context in

subsequent interactions (and can be used by the MCP logic layer to give the LLM some memory of past chats, within reason). The Node backend uses an ORM like **Prisma** or **Sequelize** for type-safe DB interactions, and to manage migrations for evolving the schema.
- **Deployment: Docker & Render.com** for deployment. Both the Next.js frontend and Node/Express API can be containerized. For simplicity, in production they may run as one unified service (Next.js app with API routes) or as separate services (one for frontend, one for backend API). For clarity, this plan treats them as one combined full-stack service deployed on **Render**. Render is a cloud platform that can deploy web services directly from a GitHub repo, and also provides managed databases. We will use **Docker Compose** for local development (bringing up the web app, database, and possibly an n8n container if needed). Docker Compose helps simulate the multi-service environment easily in dev/test [5]. On Render, the Node/Next service will connect to a managed Postgres instance; environment variables on Render will store the DB connection string and any API keys or secrets. (Render's environment variable configuration ensures we don't hardcode secrets like DB passwords in code [6] [7].) The Ollama LLM can be deployed either on the same server (if using a beefy instance with the model pre-loaded) or kept as a local-only feature for demo due to resource constraints – for a portfolio, it's acceptable if the AI features run locally only during presentations. The n8n instance may not be hosted on Render (since n8n might be hosted externally or on a separate VM), but the Node backend just needs the ability to call the n8n webhook URL.

Architecture Overview: The architecture follows a modular, service-oriented design, ensuring clear separation of concerns between the user interface, server logic, AI integration, and automation workflows:

- **Frontend (Next.js) Architecture:** The Next.js app consists of pages and components that implement the chat UI and any supporting pages (like a homepage or info pages for the clinic). A primary component is the **Chat Interface** component (e.g. `ChatWindow` component) which includes:
- A message list display (showing user messages and bot replies in sequence, with timestamps and possibly labels like "Patient" and "Assistant").
- An input box and send button for the user to type questions or information.
- Optional quick-select buttons or suggested questions (like "Speak to a human" or common FAQs) for convenience.
- The chat component manages state for the conversation. Upon user sending a message, it optimistically appends the user message to the list and calls the backend API for a response.
- The UI may show a loading indicator while waiting for the AI response. When the response arrives, it is appended to the conversation.
- Additional components: a header with the clinic name and chatbot avatar, and possibly a footer or help section with a disclaimer ("MediConnect AI is not a licensed physician and is for informational purposes [1] ").
- Next.js API routes (e.g. `/api/chat`) could directly proxy to the backend logic, or the frontend could call a separate Express endpoint. In this architecture, we consider that Next's API route will forward the request to the Express/MCP logic or directly call the Ollama client and handle logic if simpler.

- **State management:** The chat state (messages, user info) can be kept in React context or a global store. We also use Next.js's built-in fetching capabilities to call our API route for new messages. This keeps the front-end relatively thin; it doesn't contain AI logic or data beyond what to display.

- **Backend Services & MCP Logic:** The backend is logically divided into:

- **Express API Layer:** Defines endpoints such as `POST /api/chatMessage` (to handle a new incoming message from user), `GET /api/appointments` (to fetch available slots or user's appointments), etc. This layer handles authentication (if any), basic validation of input, and then passes requests to the appropriate service in the logic layer.
- **MCP Logic Layer (Service Layer):** Here is the "brain" of the app: a controller that decides how to handle a user query. "MCP" stands for a protocol to standardize context for LLMs, but in our design it also implies a **central orchestrator**. For a given user message, the logic may do one or more of:
  - Check the content or intent of the message (possibly using keywords or even a lightweight NLP classifier). For instance, if the user asks a purely informational question ("What are symptoms of flu?"), the logic will primarily query the LLM. If the user asks for an appointment ("I need to see a doctor next week"), the logic will trigger the scheduling workflow.
  - **Calling the LLM (Ollama):** For informational or conversational responses, the logic prepares a prompt. It can include context such as the conversation history (limited to last few interactions) and relevant data from the database if needed. It then calls the Ollama API using the Node `ollama` client. The LLM (say a local Llama-2-based model) generates a response stream. The Node API can stream the response back to the frontend (for a realistic typing effect), or aggregate it and send once complete. (Ollama's client handles communication and manages long responses without timing out [3].) The result is then returned to the Express layer which sends it to the user.
  - **Triggering n8n via Webhooks:** For certain intents (like scheduling or sending some info externally), the logic constructs a JSON payload and sends an HTTP POST to a pre-configured n8n webhook URL. For example, a workflow "CreateAppointment" could be set up in n8n with a Webhook trigger node. The Node logic will call that URL with patient-provided details (name, contact, requested time). As the Medium notes, in n8n "webhooks let you start a workflow from outside by sending a request to a special link, and your workflow will run" [4]. The n8n workflow might then add an entry to a Google Calendar or send a confirmation email via an Email node, and possibly return a response. Our Node backend, after calling the webhook, can wait for a quick response or simply inform the user that "Your appointment request has been sent!" depending on the design (if n8n is set to sync respond or not). Another example: if a user asks for "Connect me with a human", the logic might trigger an n8n flow that pings a human agent (via SMS or email) to contact the patient.
  - **Database Access:** The logic layer also interacts with Postgres via an ORM. For example, before calling the LLM, it might retrieve the patient's profile or recent health history from the DB (if that context should influence the answer). Or if the user asks "what was the advice you gave me last time?", it can fetch previous conversation logs from a `conversations` table to provide context. In the scheduling scenario, the logic might check an `appointments` table or a `doctors_availability` table to see open slots, and provide those to the user or to n8n.
- **Security & Validation:** Although this is a demo app, basic checks are in place. Input from the frontend is validated (to avoid very long prompts or malicious content). The backend can implement rate limiting or authentication if needed (e.g., require patients to log in via a simple token or email verification for personalized info). All secrets (like DB credentials, n8n webhook URLs if they contain keys, etc.) are stored in environment variables (which Render securely manages [7]).

- **Scalability Considerations:** While one Node service can handle both API and MCP logic, this architecture could be scaled by separating the AI handling into a microservice. For instance, an **AI Service** that purely handles calls to Ollama (could even be a serverless function or another Node process) and returns results to the main API. Given a 5-person team including AI engineers, one could be tasked with optimizing this AI service (e.g., model selection, prompt engineering, perhaps fine-tuning the model or using MCP protocol implementations to integrate new tools). But initially, a single backend service is enough for the prototype.

- **Integration Points:** The key integration points in this architecture are:

- **Ollama LLM Integration:** The Node backend communicates with Ollama's local server (running on port 11434 by default [8]). The integration uses the `ollama` NPM package, which "allows you to use JavaScript to interact with the Ollama server API" [3]. The model (e.g. `llama2-med`, hypothetically) must be downloaded and served. The backend ensures the Ollama service is running (e.g. by a startup script `ollama serve` or checking the service health).
- **n8n Workflow Integration:** The backend holds the URLs for relevant webhooks. Each workflow in n8n (like `wf_create_appointment`) once deployed will provide a URL of the form `https://n8n.example.com/webhook/****`. The Node service uses a standard HTTP client (like axios or fetch) to POST to those URLs with JSON data. Optionally, the n8n workflow could respond with some data (like confirmation details) which our service can pass back to the user. The communication is secured by using n8n's built-in API key or basic auth on the webhook if necessary (for demo we might not secure it heavily, but we could include a secret in the URL path).

- **PostgreSQL Database:** The Node service connects to Postgres via connection string (e.g., `postgres://user:pass@host:5432/dbname`). In local dev, Docker Compose sets up a `postgres` service with a known password, which is fine for dev. In production (Render), a managed Postgres instance is provisioned and we use the connection string provided by Render in the `DATABASE_URL` environment variable. The DB schema (outlined below) is migrated on deployment (using Prisma Migrate or a migration script).

- **PostgreSQL Schema Outline:** Key tables for MediConnect AI might include:

- `users` – if users create accounts or if we store basic patient info. Fields: `user_id` (PK), `name`, `email`, `phone`, etc., plus maybe `medical_record_id` if linking to an external system. (For demo, user accounts might be optional; the chatbot could be used anonymously with limited features.)
- `conversations` – to log chat sessions. Fields: `conversation_id` (PK), `user_id` (foreign key to users, nullable if anonymous), `start_time`, `end_time`. Could have a `transcript` or we might store messages in a separate table.
- `messages` – if storing each message: `message_id`, `conversation_id` (FK), `sender` ("user" or "assistant"), `content`, `timestamp`. Storing chat history can help train future models or analyze usage.
- `appointments` – stores appointment requests/confirmed appointments. Fields: `appointment_id`, `user_id` (FK), `requested_time`, `confirmed_time` (could be null if not confirmed yet), `status` ("requested", "confirmed", "completed"), and maybe `notes` (e.g., symptoms described).
- `staff` or `doctors` – list of doctors or practitioners, if needed to schedule appointments with specific people.
- Possibly `faq` or `knowledge_base` – if we want the chatbot to have specific curated info (like clinic hours, common illness info) to augment the LLM. This could be used by the MCP logic layer to fetch

quick answers instead of always relying on the LLM (a simple form of retrieval augmented generation, but within scope, maybe store a few Q&A pairs).

The above schema can be adjusted as needed; for demonstration, even a subset (conversations and appointments) would suffice. Ensuring foreign key relations and using indexes on `user_id` fields will keep queries fast. The ORM or raw SQL can be used to query these tables as needed when the chatbot logic runs.

Development Steps & Setup Instructions: The development of MediConnect AI follows best practices, from repository initialization to deployment. Below is a step-by-step guide for the development workflow:

1. **Project Initialization (`git init`):** Start by creating a new Git repository (e.g., `mediconnect-ai-chatbot`). Initialize Node.js and Next.js projects:
2. Run `npx create-next-app@latest mediconnect-ai --typescript` to scaffold a Next.js application with TypeScript support. This sets up the frontend (and we will integrate our backend logic into it). Alternatively, if separating front and back, run `create-next-app` for frontend and a separate `npm init` + Express setup for backend.
3. Initialize a Node.js Express app inside the repository for the backend. This could be in a subfolder `/api` or as part of Next.js API routes. For clarity in this plan, assume a separate Express app in `server/` directory.
4. Use a monorepo approach (one Git repo, perhaps using yarn workspaces or npm workspaces) to hold both `web` (Next.js app) and `server` (Express API). This allows separate deployment if needed but easy code sharing (for example, shared types or utility functions).

5. Commit the initial scaffold code to Git. Establish a remote repo on GitHub (since Render integrates with GitHub for deployments).

6. **Repo Structure & File Conventions:** Organize the repository clearly:

7. Root directory contains a README.md (for documentation), a `docker-compose.yml` (for local development), and perhaps an `.n8n/` directory to store any n8n workflow JSON exports or credentials (not mandatory, only if we want to version control workflows designs).
8. `web/` (Next.js app) directory contains the usual Next.js structure: `pages/` (or `app/` if using Next 13+ App Router) for pages, `components/` for UI components (including `ChatWindow.tsx`, `MessageBubble.tsx`, etc.), `styles/` for global styles or theme definitions (could use CSS modules or Chakra theme config), and possibly `utils/` for any helpers.
9. `server/` directory (Express backend) contains an `index.ts` (entry point), `routes/` for route definitions, `services/` for MCP logic, `models/` for database ORM models or Prisma schema (e.g., `schema.prisma` if using Prisma), and `controllers/` if following MVC separation (controllers call services).
10. Ensure naming conventions are clear: use PascalCase for React components, camelCase for variables, and consistent file naming (e.g. all Next.js pages in lowercase or something).
11. Add a `.env.example` file at root to document required environment variables (like `DATABASE_URL`, `N8N_WEBHOOK_URL_APPT`, etc.), but not commit actual secrets.
12. Setup `.gitignore` to ignore `node_modules/`, `.next/` build folder, `.env` files, and possibly `docker-volume/` if any.

13. Write a **README** early that describes the project and how to set up and run it (this will be expanded as development progresses to serve as documentation in the portfolio).

14. **Setting Up Database (Postgres) Locally:** Use Docker Compose to run Postgres for local development:

15. Write a `docker-compose.yml` with services for `db` (Postgres) and maybe `n8n` (for a local n8n instance) as well as the app itself. For example, use the official Postgres image with environment variables for `POSTGRES_USER`, `POSTGRES_PASSWORD`, `POSTGRES_DB` (use dev credentials). Map port 5432 so the dev DB is accessible.
16. If including n8n in the dev environment, use the `n8nio/n8n` image, mapping port 5678 and a volume for persistence (so that flows you create persist). This is optional; the team might also use a cloud n8n or just manually trigger flows.
17. Docker Compose can also define a service for our Express server and Next app, but during active development, developers might run Next.js with `npm run dev` locally outside Docker for faster reload. The compose is mainly to easily spin up dependencies (DB, n8n, maybe Ollama if there's a container for it).

18. Confirm the Node app can connect to Postgres (using e.g. host `db` and the credentials as per compose). By using Docker Compose, the dev environment is easy to start and stop, and consistent for all team members [5].

19. **Implementing the Backend Logic:** Start coding the Express server:

20. Define routes in Express, e.g., `POST /chat` which expects JSON `{ userId, message }`. This route will call a controller function (in `controllers/chatController.ts`) that invokes the MCP logic service.
21. Implement the MCP logic in a service module (say `services/chatService.ts`). This will contain the logic described earlier: check message intent or content, call LLM or n8n as needed, and build a response.
22. Integrate the **Ollama API**: Install the `ollama` npm package. Configure an Ollama client or simply use the package's exported functions. For instance, if using the client, one might do:

```
import ollama from 'ollama';
const response = await ollama.generate({ prompt: promptText, model:
'llama2-med' });
```

Ensure the Ollama server is running. Possibly create a small utility to check the server health (ping `http://localhost:11434`) and maybe start it (if on the same machine, though in production on Render, running Ollama might not be feasible – the team can decide to run LLM on a separate server if needed).
23. Add streaming support: The Ollama client supports streaming responses. We can choose to accumulate and then send, or stream directly to client. As a first iteration, accumulate the full response text.
24. Integrate **n8n webhook calls**: For example, if in `chatService` we detect `intent === 'schedule_appointment'`, then prepare the data (could be as gathered from a structured

conversation or simple regex parse of date/time in user message) and `await fetch(N8N_APPT_WEBHOOK_URL, { method: 'POST', body: JSON.stringify(data) });`. Use environment variable for `N8N_APPT_WEBHOOK_URL` so that the actual URL (which may contain an internal API key or unique path) isn't hardcoded. The response from n8n (if any) can be awaited. If n8n immediately returns a result (like confirmation), include that in the chatbot's reply (e.g., "Your appointment is booked for 10/12 at 3PM"). If n8n just responds with 200 and does work in background (more likely for asynchronous flows), then the chatbot can reply with a generic "Your request has been received, our staff will confirm shortly."

25. Database operations: Use the ORM to save the message to `messages` table, and to retrieve any needed info. E.g., if user asks "Do I have an appointment tomorrow?", the service can query the `appointments` table for that user and date, and respond accordingly (perhaps directly without AI, or by feeding that info into the prompt: "The user has an appointment tomorrow at 5 PM, incorporate that into your answer").
26. Include error handling: if the Ollama call fails or times out, catch it and respond with a friendly error message ("Sorry, I'm having trouble accessing that information right now."). Likewise for n8n calls.

27. Logging: instrument the server with simple logs (or use a library like Winston) to log key events (requests, AI responses, errors) for debugging. This is useful for a portfolio to demonstrate awareness of monitoring.

28. **Implementing the Frontend:** Build the Next.js pages and components:

29. Create a Next.js page for the chatbot, e.g. `pages/index.tsx` could render the main chat interface directly (if this is primarily a one-page app). Alternatively, if the site has a marketing page and then a chat page, structure accordingly.
30. Develop the ChatWindow component: uses React state to store an array of messages. Initially, it may fetch a welcome message from the server or just display a static welcome. Users input text, which triggers an API call. Use Next's API route `/api/chat` to forward to the Express logic (if Express is separate, the API route might just proxy or you configure Next to talk to the backend via an environment variable like NEXT_PUBLIC_API_URL).
31. Ensure the UI updates optimistically: add the user message to the list immediately. For the assistant's response, you could either poll an endpoint for streaming or simply wait for the fetch to resolve with the final answer and then update state. A loading spinner or "typing..." indicator can be shown in the meantime.
32. Style the components according to branding: e.g., message bubbles: user messages might align right with a distinct color (perhaps the teal), bot messages align left with a grey background and an icon. Use CSS (possibly module CSS or a CSS-in-JS solution) to implement these styles. If using Chakra UI or similar, define a custom theme with the brand colors.
33. Add any needed forms or modals. For instance, if the conversation gets to scheduling, maybe the bot or UI could show a date-picker component for the user to choose a date/time instead of free text (optional, but a nice enhancement demonstrating UI skills). That selection could still be sent to backend as a structured message.

34. Test the front-end thoroughly by running `npm run dev` and interacting with the chat UI, ensuring messages cycle properly.

35. **Continuous Integration & Deployment Setup:**

36. **Version Control:** Throughout development, commit regularly with clear messages (e.g., "Implement basic chat UI", "Add Ollama integration", "Setup n8n webhook for appointments"). A recommended schedule might be daily commits or each significant feature, and pushing to GitHub. Use branches for major features (e.g., `feature/ollama-integration`) and then merge to main. For a small team, a simple git flow with code reviews (PRs) can be practiced.
37. **CI Pipeline:** Set up GitHub Actions or another CI service to run tests and maybe build the Docker image on each push. A simple action can lint the code (ESLint) and run `npm run build` for both front and back to catch errors. This ensures that broken code isn't deployed. For now, since this is a portfolio piece, CI is nice-to-have; at minimum, ensure the project builds and all tests pass before deployment.
38. **Docker Configuration:** Write a `Dockerfile` for the web app. If combining front and back, the Dockerfile might use a multi-stage build: first install dependencies and build the Next.js app (if using `next build`), then start the Node server. Alternatively, create separate Dockerfiles: one for Next.js (which can also serve the app via Node) and one for any separate Express service. For simplicity: a single Dockerfile using Node 18+, copying the code, installing deps, and running `next start` (which will handle both frontend and API routes). Expose port 3000. Ensure to `COPY . .` all necessary files except those ignored. Use `.dockerignore` to avoid copying `node_modules` and other unnecessary files.
39. Test Docker build locally: `docker build . -t mediconnect-ai` and `docker run -p 3000:3000 mediconnect-ai` to verify it starts up and can serve requests (connected to a running Postgres).
40. **Render Deployment:** Log in to Render, create a new **Web Service**. Link it to the GitHub repo. Choose the environment (likely Node). If using Docker, provide the Dockerfile path and build command (Render can auto-detect if Dockerfile is present). Otherwise, specify build command `npm install && npm run build` and start command `npm run start`.
    - Add environment variables in Render dashboard: e.g., `DATABASE_URL` (Render will provide this if using their Postgres add-on), `PORT` (Render uses 10000 by default but Next listens on 3000; make sure to use the env `PORT` in the code or package.json start), `N8N_APPT_WEBHOOK_URL`, and any others like `OLLAMA_SERVER_URL` if needed.
    - Provision a **PostgreSQL** database on Render (there's an option to add a database). After creation, set the `DATABASE_URL` in the web service accordingly [9] [10]. Apply any initial schema (could use an SQL migration or let the app create tables on first run if using an ORM sync).
    - (Optional) Deploy n8n: If wanting n8n on Render, you could deploy it as a Docker service (Render supports Docker services). But it might be easier to use n8n cloud or a separate service due to the stateful nature and memory needs of n8n. In any case, ensure the webhooks in the Node app point to wherever n8n is running (with a public URL).
41. Once environment is set, trigger a deployment. Render will pull the repo, build the Docker image, and launch the service. Monitor logs on Render for any issues (especially around connecting to the database or needing to run migrations).
42. Set up a **domain or URL**: Render provides a default subdomain, which can be used to demo the app (e.g., https://mediconnect-ai.onrender.com). For portfolio, that link could be shared.

43. **Secrets Management:** As noted, all secrets (like any API keys, though here mostly DB password which is in the connection string, and perhaps an n8n basic auth token if used) are kept out of the code. Render's environment variable settings make it easy to manage these without exposing in the

repo [7]. During local dev, use a `.env` file with similar variables; never commit it. This demonstrates good practice in credential management.

44. **Documentation & Diagrams:** Ensure the project is well-documented:

45. The **README.md** should contain an overview of the project, setup instructions (for local dev and how to run Docker, etc.), and usage instructions (how to interact with the chatbot, what sample questions to try). Include architecture diagrams or flowcharts if possible (for the portfolio, perhaps a simple diagram of the architecture: showing user -> Next.js UI -> Node backend -> (LLM, DB, n8n) might be drawn using a tool and included).
46. Create a **Technical Specifications** document (could be in the repo as well or in the README) detailing the architecture (much like this overview). This shows potential employers the thought process and design skills.
47. Document the **n8n workflows**: for example, export the workflow JSON or take a screenshot of the n8n workflow diagram (which shows nodes connected) and include it in the docs. This helps illustrate the automation visually.
48. If any environment setup is tricky (like installing Ollama for local use), document those steps too (e.g., "Install Ollama from ollama.com, run `ollama serve` and download model X before running the app").

49. Write a short **user guide**: what can the chatbot do (list of example queries it can handle like "I have a headache, what should I do?" or "Schedule me for a check-up next Monday") so that in a demo the functionality can be showcased easily.

50. **Testing & Iteration:** During development, test each piece:

51. Write unit tests for the MCP logic if possible (e.g., using Jest to test that given a certain input, it calls the LLM or calls the n8n webhook appropriately, perhaps by mocking those external calls).
52. Manually test the end-to-end flow: run the whole app locally (Next dev server, Express server, Postgres, and maybe a local n8n). Simulate user queries that cover each path (info question -> get LLM answer, scheduling request -> triggers n8n and get confirmation).
53. Ensure the AI responses are appropriate length and tone for medical context. The prompt can be adjusted (AI engineers on the team can refine the system prompt to enforce a concise, safe advice style).
54. Gather feedback from team members and iterate on both the conversation flow (maybe adding quick-reply buttons for common actions) and the technical implementation (e.g., improving error handling or scaling configuration).

By following these steps, the development team will produce a robust, portfolio-ready project. The end result is an application where a user (patient) can converse with a medical chatbot that not only provides helpful answers via an LLM, but also takes actions like scheduling appointments through workflow automation. This project demonstrates full-stack integration: a polished frontend, a secure and well-structured backend with AI capabilities, integration of third-party automation (n8n), and cloud deployment.

Workflow Description (User Journey): Below is how a typical interaction flows through the system, tying together all components of MediConnect AI:

- *User Initiates Chat:* Jane, a patient, opens the MediConnect AI chat on the clinic's website. She's greeted by a message: "Hello, I'm MediConnect, your virtual health assistant. How can I help you today?" (This could be a static welcome from the frontend or fetched from the server.)
- *User Question:* Jane types, "I have had a sore throat and sneezing for two days, what should I do?" and hits send.
- *Frontend to Backend:* The Next.js frontend captures the message and sends an AJAX POST request to the `/api/chat` endpoint (through an API route or directly to the Express service). The request payload includes Jane's user ID (if logged in, or a session ID if anonymous) and her message text. The UI shows Jane's message in the chat log immediately.
- *Backend Processing (MCP Logic):* The Express route for `/chat` receives the request. It creates a message entry in the database (`messages` table) for record-keeping. It then passes the content to the ChatService (MCP logic).
- The logic identifies this is a medical advice query (perhaps checking keywords like "sore throat" which is not a direct action like scheduling).
- It formulates a prompt for the LLM. For example, a system prompt might be: "You are a helpful medical assistant for a clinic. The patient describes symptoms. Provide concise advice and possible next steps. If urgent, advise to seek immediate care. Always add a friendly tone and remind them you are not a doctor." Then user prompt: "Patient: I have a sore throat and sneezing for two days."
- The service calls the Ollama API to generate a response. The local LLM (say a variant of Llama-2 with medical tuning) processes this input. The Node backend may stream the output.
- *AI Response:* The LLM returns a response, e.g., "Assistant: It sounds like you have symptoms of a common cold or possibly seasonal allergies. I recommend getting rest, staying hydrated, and using throat lozenges for your sore throat. If you develop a high fever or if symptoms last more than a week, consider seeing a doctor. Hope you feel better! (Disclaimer: I'm an AI assistant, not a doctor.)"
- The backend receives this full text. It may append a standard disclaimer if not already included. The response is saved to the `messages` table linked to this conversation.
- The Express route sends the response back as JSON to the frontend.
- *Frontend Update:* The Next.js frontend receives the JSON and appends the assistant's answer into the chat window. Jane sees the answer appear. She appreciates the advice. The chat UI perhaps highlights or stylizes the disclaimer portion differently (italic text).
- *Follow-up Action:* The assistant's advice triggers Jane to ask another question: "Thanks. Actually, can I schedule a visit just to be sure it's nothing serious?" She sends this message.
- *Scheduling Intent Recognized:* Backend gets this message. The MCP logic detects keywords "schedule a visit" – this maps to an **appointment scheduling** intent.
- The service might ask a clarifying question through the bot: "Sure. What days work best for you this week?" (This could be another AI response or a canned response from logic). Say Jane responds with a date/time.
- Now the service has enough info (desired timeframe). It prepares an appointment request object (Jane's user ID, preferred date next week, reason "sore throat check-up").
- Instead of querying the LLM for an answer, the logic triggers the **n8n webhook** for scheduling. The Node backend posts to the n8n workflow URL with that data.
- *n8n Workflow Execution:* In n8n, a workflow "Schedule Appointment" is triggered by the webhook. The workflow might do the following:
- A **Webhook node** receives the data (date, user info).

- A **Function node** (or series of nodes) checks the clinic's calendar (maybe by API or database; for simplicity, maybe n8n itself could query the same Postgres or an in-memory schedule).
- It finds an available slot with Dr. Smith on that date at 4 PM.
- A **Database node** (or HTTP Request node to our backend) could create a new row in the `appointments` table for that slot (alternatively, the backend could have waited and done this, but demonstrating n8n writing to DB shows its power. However, direct DB access by n8n would require a DB credential; for a demo, it might be easier for n8n to call a secure backend endpoint to create the appointment).
- A **Send Email node** sends Jane a confirmation email (since her email was provided or on file) saying "Your appointment is confirmed for [date/time] with Dr. Smith."
- The n8n workflow finishes and can respond back to the webhook call with a result (perhaps the confirmed slot or a success message).
- *Backend Continues:* The Node backend, which awaited the webhook call, receives a response from n8n (if configured to wait). The response might contain confirmation details (date/time). The backend then formulates a final chat message to Jane: "Your appointment has been scheduled for March 10 at 4:00 PM with Dr. Smith. You'll receive a confirmation email shortly. Is there anything else I can help you with today?"
- If the n8n call did not return in-line (maybe we set it to fire-and-forget to not keep the user waiting too long), the backend can instead immediately reply: "I've forwarded your request to our scheduling system, and you will receive confirmation soon." In either case, Jane gets feedback.
- The appointment details are now stored in the database via the workflow or via the backend. The team can later verify in the DB that it was saved.
- *Conversation Continues or Ends:* Jane says "No, that's all. Thank you!", the bot responds politely, and the conversation can be ended. The session could remain open for new questions later, with context retained in the DB or temporarily in memory.
- *Demonstration and Logging:* Throughout this interaction, logs have been recording each step on the backend (for the developers to show in the portfolio if needed). For instance, logs show "User message received", "LLM response generated in X ms", "n8n webhook triggered for appointment", etc. These can be shown to illustrate the behind-the-scenes operations.

This user journey highlights how **MediConnect AI** seamlessly integrates conversational AI with actionable workflow automation. It leverages the power of a local LLM (via **Ollama**) to understand and respond to user needs, and the flexibility of **n8n** to perform operations in the real world (like scheduling). The combination is orchestrated by the Node.js backend (MCP logic layer), which ensures the right tool is used for the right task. For the developer portfolio, this project demonstrates: - Competence in setting up a full-stack JS application (Next.js + Node/Express + Postgres). - Ability to integrate and deploy a local AI model (Ollama) and handle its I/O in an app. - Use of automation tools (n8n) through webhooks for extending functionality without reinventing the wheel. - Good software engineering practices (containerization with Docker, using environment variables for secrets [6], deploying to cloud, and writing thorough documentation).

Each of these aspects would be highlighted in the portfolio presentation of MediConnect AI, making it a standout project for both technical depth and real-world relevance.

Project 2: BizAssist Dashboard – Small Business Operations Chatbot

Use Case & Target Audience: BizAssist Dashboard is a web application designed for a **small business owner** or office manager, providing an AI-driven assistant to help manage day-to-day operations. The use case is centered on a small business (e.g., a local retail shop or an online store) that needs quick access to business info and simple task automation through a conversational interface. For example, the owner can ask the chatbot things like "What were last month's sales?" or "Add a task to call the supplier tomorrow." The assistant can fetch data (sales figures from a database) or create tasks/alerts (via automation flows). This scenario focuses on **operational efficiency** – giving a busy small business person an easy way to interact with their data and tools via chat instead of clicking through multiple systems. It's like a smart dashboard that you can talk to, hence "BizAssist Dashboard." It demonstrates AI integration for business data querying and workflow automation for business tasks.

Branding & Design: The branding for BizAssist is professional yet approachable, fitting small business aesthetics with a tech-savvy twist: - **Project Name:** *BizAssist Dashboard* (tagline: "Your Business Assistant, At Your Command"). The name conveys assistance in business tasks. We might shorten it to *BizAssist AI* informally. - **Color Palette:** A blend of **navy blue** (e.g., #264653) and **vibrant orange** (e.g., #F4A261) with neutral greys/white. Navy blue suggests professionalism and stability, while orange adds energy and creativity – reflecting a small business vibe that is dynamic but trustworthy. These also happen to be friendly to both corporate and creative settings. - **Fonts:** Use a modern sans-serif like *Segoe UI* or *Inter* for body text for clarity, and a slightly more distinctive font for headings like *Poppins* or *Montserrat* to give a tech-modern feel. The text should be easily readable on screens and convey a sense of efficiency. - **Design Direction:** The interface is a **dashboard-style** web app with a prominent chat panel. Alongside the chatbot panel, there may be quick-view widgets for key business metrics (e.g., sales today, pending orders, tasks) to complement the conversational interface. The chat UI itself will have a clean design with perhaps a minimalistic avatar (maybe a simple chat bubble icon with a briefcase to hint at business assistant). The overall design should balance data visualization and chat. If the window is large enough, the left side could have navigation or cards, and the right side is the chat. On mobile, it might just be the chat interface. We ensure responsiveness. The style uses slight shadows, cards, and the chosen color highlights (orange for interactive elements like send button or highlights in graphs, blue for headers or chat bubbles from the assistant).

Technology Stack: BizAssist Dashboard's tech stack leverages full-stack JavaScript and integration tools to access data and tasks: - **Frontend:** Built with **React** (using Next.js for ease of deployment and SSR). The front-end is a single-page application feel, with Next.js routing if multiple pages (like a settings page). It heavily uses dynamic data – possibly integrating a chart library (like Chart.js or Recharts) to display any numeric answers in a visual form (for example, if the user asks sales trends, the assistant could respond with text and the UI could also render a chart). The UI may also integrate a component library such as **Material-UI (MUI)** for a professional dashboard look (MUI provides ready-made components like cards, data grids, etc., which can fit the admin-dashboard style). - **Backend: Node.js + Express** forms the backend API as well as the core logic layer. In this project, the backend not only handles chat but also provides endpoints for business data. The architecture might have a unified Next.js server (with API routes) or a separate Express service. For clarity, let's assume a separate Express API that Next calls for both chat and data fetching. The backend also includes an **MCP logic layer** to orchestrate AI and automation similar to Project 1: it interprets user requests and decides whether to answer via the LLM or perform a database query or trigger a workflow. The presence of potentially sensitive business data also means the backend must include an authentication layer (maybe a simple login for the owner) – perhaps using Next Auth or

JWT, but for portfolio demonstration, it could be a basic login form and session. - **Database: PostgreSQL** is used to store business data and application data. For demonstration, we can simulate typical small business data: e.g., a table `sales` (records of transactions), `products` (inventory or items), `tasks` (to-do or reminders), and `users` (if multi-user, e.g., the owner plus employees). The database can be pre-populated with dummy data to allow the chatbot to answer questions like "What are the top 5 products by sales this week?" by actually querying this data. We might also have a `conversations` table (like the first project) if we want to log interactions. - **AI (Ollama LLM Integration):** For questions that require interpretation or more conversational answers (or when data needs summarization), the **Ollama** local LLM is used. For instance, if the user asks "How did the store perform this quarter compared to last quarter?", the system might run raw SQL to get the numbers, then use the LLM to generate a concise summary or explanation. The Node backend thus has two modes: direct query vs. query + LLM. The same Ollama integration method is used (via the `ollama` npm client calling the local server). The chosen model could be a general model (Llama 2 13B perhaps) since this is not domain-specific like medical, or even a smaller model for speed. The prompt for business queries might instruct the AI to be a data analyst style assistant: e.g., "You are a business assistant with access to data. When possible, provide concise answers. If numbers are provided from the database, incorporate them in the answer clearly." The LLM ensures the responses are in natural language and can also handle open-ended questions (like advice or "what can I do to improve sales?" — it might give generic suggestions). - **Automation (n8n) and External Integrations:** BizAssist uses **n8n** to automate certain small business tasks external to the app's core. Examples: - If the user says "Remind me to call the supplier next Monday at 9am," the backend identifies a reminder/task creation intent. It could simply save a task in the DB and rely on the app to show it. But to demonstrate integration, we use n8n to handle reminders: the backend calls an n8n webhook for "create calendar reminder," passing details. n8n workflow then could integrate with, say, Google Calendar or send an email/SMS reminder at the specified time. Or n8n could simply echo back a confirmation. This offloads scheduling logic to n8n's cron capabilities. - If the user requests "Email the sales report to me," the backend could fetch the data and then call an n8n workflow that has an **Email node** (or uses Gmail/SendGrid integration) to send out an email with an attachment (the report, maybe as CSV or PDF). The Node service might generate the report content or instruct n8n to do it (depending on complexity). - If integrated with external APIs (for demo, maybe not actually calling them but simulating), n8n can be the go-between. E.g., "track shipment for order 123" could trigger an n8n flow to call a shipping API and return info. - In summary, n8n acts as the **operations hub** for tasks that go beyond read-only queries: scheduling, emailing, notifications, or updating external systems. The Node backend uses webhooks to start these flows, similar to Project 1's pattern. - **Deployment:** Deployed on **Render.com** as well. We'll have possibly two services: one for the web app (Next.js/Node server) and one for the Postgres DB (managed). For n8n, if needed, possibly another service or an external instance. Docker is used for consistency in dev and deploy. Docker Compose will include Node and DB for local dev (and optionally n8n). During development, Docker helps ensure that the Postgres and any required services are easily available. As before, environment variables store secrets and endpoints (like DB URL, n8n webhook URLs, maybe an API key if any external service is used; these are configured on Render's dashboard in production).

Architecture Overview: - **Frontend (React/Next.js):** The UI is a dashboard with a chat interface: - The main page (e.g., `/dashboard`) might have a two-column layout: left side with "insight cards" and right side with the chatbot panel. The insight cards can display data fetched via API (e.g., total sales today, number of orders pending, etc.). These give at-a-glance info and also serve as prompts for the user to dig deeper via chat if needed. - The chat component works similarly to Project 1: a message list and an input box. Additionally, above the chat or in a menu, there might be some preset example queries (like a dropdown or button list: "View Sales Report", "List Overdue Tasks") that when clicked, feed a query into the chat. This is to guide demonstration. - We incorporate authentication: perhaps a simple login page (`/login`) and then the dashboard requires login (Next.js can handle routing or an Auth provider, but in simplest form we check a session cookie). This is mainly to simulate that business data is protected. - UI components will include charts or tables for data. For instance, if the user asks for sales trends, the assistant's answer could include an array of numbers or link to data that the frontend knows how to plot. We can either: - Generate a chart server-side and send as an image (complex, skip for demo), or - Have the backend send raw data along with the textual answer, and the frontend has logic: if response JSON contains a data array, it renders a chart component with that data. This shows off some creative front-end work. - The styling uses Material-UI components: an `<AppBar>` for the top with the business name/logo, a side `<Drawer>` for navigation if needed (like links to different sections), and cards for metrics. The chat uses a Paper or Card component for the chat area. Colors (navy, orange) are applied via a custom MUI theme.

- **Backend Logic & Components:** The backend has dual responsibility: answering chat queries and providing API endpoints for direct data (like the cards on the dashboard).
- **Express API Endpoints:** Besides the chat endpoint (`POST /chat`), we have GET endpoints like `/api/metrics/today-sales`, `/api/tasks`, etc., which the frontend might call on page load to populate the dashboard cards. These endpoints query the database and return JSON.
    - Example: `GET /api/metrics/today-sales` returns `{ date: 2025-05-23, totalSales: 1234.56 }`.
    - These allow the app to function in a classic way, but the chat can also retrieve this info through the same or similar functions called via the chat logic.
- **MCP Chat Logic:** The `chatService` (MCP layer) interprets user messages to either:
    - Query the database directly for factual answers. e.g., User: "What were my sales last month?" -> The service recognizes the pattern, performs a SQL query to sum last month's sales, and then **instead of asking the LLM blindly**, it might incorporate the result in a prompt or might simply have a template: "Your sales for April 2025 were $X." If the user specifically says "show me the sales by week for last month", the logic could fetch that data and either return a table or feed it to the LLM to phrase nicely.
    - Use LLM for open-ended or summary questions: e.g., "How can I improve sales?" – here the system might not have a single factual answer. It could either answer from a knowledge base (if one was provided) or directly use the LLM's knowledge (with a general model, it can give generic business advice). In that case, the prompt might include some context like the business type if known ("We are a retail shop specializing in gadgets") and then ask the LLM for suggestions.
    - Mixed approach: e.g., "Compare this year's revenue to last year's." The logic can fetch numbers from DB (say YTD revenue for 2024 vs 2025), then prompt LLM: "We had $X in 2024 and $Y in 2025 up to this date. Give a brief comparison and insight." The LLM then creates a short analysis.
    - **Tool usage concept:** Essentially, the backend acts as a LangChain-like agent: it can either answer via tool (DB query) or via LLM or both. The team's AI engineers could create a set of regex or AI classification to detect when to use which. (This is a form of implementing an MCP pattern manually by connecting model and tools.)
- **n8n Automation Integration:** The backend triggers n8n flows for tasks which are actions:
    - All "reminders" or "notifications" requests go to an n8n workflow that could handle scheduling or external communication. The Node service might immediately acknowledge the request and say "Okay, I've set a reminder."

- For the "email report" scenario: Node could gather the report data (or instruct n8n to gather it if n8n has DB access). But likely Node will query DB, format data (maybe as CSV or HTML) and send that in the webhook payload to n8n, which then uses an Email node to send it out. The workflow might respond "Email sent." which Node passes back to user.
- If we had inventory and user says "order more stock for product X", that could trigger a simulated purchase order workflow (for demo, maybe just log or pretend to call a supplier API). n8n could be used here to illustrate connecting to an external service (like sending an HTTP request to a supplier endpoint or even just creating a Trello card or Slack message).

- **Database Schema Highlights:** Some key tables:

  - `users` – likely one user (the owner) for login. Contains `id`, `username`, `password` (hashed) or we skip secure auth for demo, but assume.
  - `sales` – columns: `id`, `date`, `amount`, `product_id`, `quantity`, etc. Enough to derive totals by date/product.
  - `products` – `id`, `name`, `price`, `stock`, etc. (Could allow queries like "How many XYZ do we have in stock?" – logic would fetch from here.)
  - `tasks` – `id`, `description`, `due_date`, `completed` flag. For reminders/to-dos. If user says "what are my tasks", the system can return tasks. If user says "mark task 5 as done", we update DB, etc.
  - Possibly `expenses` or other financials if we want to extend queries. But not necessary for initial scope.
  - `messages` or `conversations` to log chat if needed.
  - The data in these tables can be dummy but realistic (some random sales records for the past year, etc.), so that queries produce meaningful outputs.

- **Security & Access:** Because this deals with business data, it's logical to implement at least a basic login. So:

- Use Next.js API or Express to handle login (e.g., a POST /login that checks a hardcoded user or an entry in `users` table). On success, create a session cookie (could use a library like express-session or NextAuth if going fancy, but a simple JWT stored in localStorage for demo is fine).
- The chat requests and data APIs then require that the user is authenticated (check a token or session). For demo, this can be simulated by requiring a session ID param or checking a flag in the frontend before allowing chat.

- This demonstrates understanding of user management, albeit lightly.

- **DevOps & Deployment:** Similar approach as project 1:

- Use Docker Compose in dev to bring up `postgres` and optionally an `n8n` container plus the `app` (Next.js + Express). The compose file can have:

```
services:
  db:
    image: postgres:15
    environment:
```

```
      POSTGRES_USER: dev
      POSTGRES_PASSWORD: dev
      POSTGRES_DB: bizassist
    ports: ["5432:5432"]
  n8n:
    image: n8nio/n8n:latest
    ports: ["5678:5678"]
    volumes: ["./n8n_data:/home/node/.n8n"]
  app:
    build: .
    ports: ["3000:3000"]
    environment:
      - DATABASE_URL=postgres://dev:dev@db:5432/bizassist
      - N8N_WEBHOOK_URL_REMINDER= ... etc
    depends_on: [db]
```

This is illustrative. With this, a developer can do `docker-compose up` and get everything running. (During active dev, one might still run Next dev server outside docker for hot reload, but compose ensures consistency.) By using Docker, the environment is portable. A note: for the LLM (Ollama), since it's local, developers would need to have it running on their host (or we add it as another service if there's an image, though Ollama is typically an app. Perhaps skip adding to compose; just document that AI won't function fully unless Ollama is running on `localhost:11434` on the host).

- CI/CD: use GitHub, commit often, possibly set up tests. The test strategy could include:
  - Unit tests for the intent recognition logic (ensuring "sales" queries go to DB, etc.).
  - Maybe a test stub for the database (or using a test DB).
  - Integration test for an API route if time permits.
- Render deployment:
  - Create a web service on Render for the app, attach Postgres as before. Use environment group to manage variables across similar projects (maybe the portfolio author can reuse environment settings if many projects on Render).
  - After deployment, test the live site by logging in and asking a query to ensure DB connectivity and that environment variables (like connection string) are correctly set.
  - Take note that if the app uses server-side generation for some pages, it might need the DB connection at build time or runtime accordingly. Typically, Next API routes handle it at runtime, which is fine.
  - If multiple services (if we separated front and back), we would deploy them separately, but likely we keep one service for simplicity.
  - The usual environment variables: `DATABASE_URL` for DB, `JWT_SECRET` or similar if using JWT, `N8N_*_WEBHOOK_URL` for webhooks, and maybe `OLLAMA_API_URL` if the Ollama service is remote (but if not using remote, then this might not be used on Render, as LLM might not be deployed due to resource).
  - Document clearly in README if the deployed version has limitations (for example, "LLM responses are stubbed on Render due to no GPU – using a placeholder response if needed").

Development Instructions: (From git init to deployment) 1. **Setup Repo & Basic App:** Initialize as before with `create-next-app` (or manually setting up Next and Express). Possibly use a template for a dashboard to speed up UI development (Next.js with Material-UI starter, etc.). Commit initial code. 2. **Add**

Express API in Next.js: Utilize Next.js API routes or a custom server to incorporate Express. Another approach: Next.js can proxy to an external API. If more straightforward, stand up an Express server on a separate port for API and use Next for the frontend only. Either way, set up the ability to handle REST calls in development. 3. **Database Schema & ORM:** Define the Postgres schema (maybe use Prisma to define models for `sales, products, tasks`). Run `prisma migrate dev` to create the SQLite or local Postgres if installed. Or write raw SQL to create tables. Populate with sample data (could write a seed script or manual SQL inserts). Ensure the data covers various scenarios for Q&A. 4. **Implement Data Endpoints:** Write Express routes and SQL/ORM queries for retrieving metrics (for dashboard cards) and listing tasks, etc. Test these endpoints (with a tool like Postman or curl) to ensure they return correct data. 5. **Implement Chat Logic Step-by-Step:** - Start with a simple echo or rule-based response to ensure the chat pipeline works (client to server and back). - Incorporate a basic intent detection. Perhaps maintain a list of trigger keywords or phrases to detect what the user wants. For example: - If message contains "sales" and maybe a time reference ("today", "this month"), label it as a sales query. - If contains "task" or "remind" or "call" -> likely a task/reminder. - If contains "how to" or "improve" with no direct data reference -> a general advice query (use LLM). - If cannot detect, default to LLM for a generic answer or a fallback "I'm not sure I can help with that." - For each intent, implement the handler. Sales queries: parse time range (you can use a library like chrono-node to parse dates in text or ask the user for clarification if needed). Then query the DB for that range, get sum or list. Return an answer or feed it to LLM. - Task creation queries: parse the task description and due date. For now, could require a pattern like "remind me to X on [date]" – parse the date, and X is description. If date not given, assume next day or ask back. - Implement actual calls: e.g., to create a task, either save in DB (for displaying in UI) and/or call n8n to set an external reminder. It might do both: DB for internal tracking, n8n for external action (like email reminder). - Use LLM via Ollama for at least a couple of scenarios: summarizing data or giving advice. For instance, once you get monthly sales numbers, pass them to LLM: "We had $10k in Jan and $12k in Feb. Give a short comparison." The LLM might say "February sales increased by 20% compared to January, a positive sign of growth." This makes the chatbot seem smarter. - Connect to Ollama (like in Project 1). Ensure `ollama serve` is running locally. Possibly, for dev, if not having a heavy model, use a smaller one to keep responses quick. 6. **Frontend Integration:** Develop React components that correspond to new features: - A login form if needed. Once logged in (simulate by setting a cookie or local state), show the dashboard. - The dashboard page uses `getServerSideProps` or similar to fetch initial data (or calls the metrics API on mount via useEffect). - The chat component is similar to before but now resides in a dashboard context. It sends requests to `/api/chat` with maybe an auth token. - The component that receives responses should handle cases where the response includes structured data. For example, if the response JSON from server has `chartData` field, render a chart. The server can decide to include that for known queries. E.g., for "show me sales each month this year", the server might return `{ text: "Here are the monthly sales...", data: [ {month:1, sales:1000}, {...} ] }`. The client sees `data` and then uses a Chart component to display it. This demonstrates a richer integration than text-only. - Implement a section for tasks: maybe below the chat or on the side, a list of tasks from the DB (via API). So when a new reminder is added, the UI can update the list (either via another API call or push update if using websockets for real-time, but that may be overkill; polling or just refresh on each chat submission might suffice). - Ensure the UI gracefully handles waiting for responses. Possibly use a loading spinner in the chat submit button or a "Assistant is typing..." indicator. - Styling: refine the CSS/ theme to the chosen colors. Possibly include the business logo (could be a placeholder logo "BizAssist Inc."). 7. **Testing:** As before, test each feature: - Does logging in gate the content properly? - Try various queries and see if the correct path is taken (maybe instrument the server to log which intent it went with). - Test n8n flows independently by sending requests to the webhook URLs with sample data (simulate what the app would send). - If possible, demonstrate that even if AI fails, the app doesn't crash (like if Ollama isn't running, perhaps have a fallback: respond "AI service unavailable, please try again later." but since it's a demo, we might ensure it's running). 8. **Documentation:** Write project-specific docs focusing on the business context: - Explain how the dummy data can be modified or extended. - Document how to set up n8n (for example, "We used n8n cloud for ease; just update the webhook URLs accordingly."). - Possibly include a short video or GIF in the repo showing the dashboard in action (optional fancy addition for portfolio). - Provide usage examples in the README (e.g., list sample questions to ask BizAssist). - Outline which external APIs could be integrated in a real scenario (e.g., QuickBooks for financial data, or Slack for notifications) to show you understand potential extensions, even if not implemented fully.

Workflow Examples: To illustrate BizAssist's functionality, here are a couple of example interactions and automations: - **Example 1: Data Query (Sales)**
User: "What were my total sales last week?"
Bot: (Backend identifies "sales last week" intent)
- The backend calculates last week's date range, queries the `sales` table for that period, sums the amounts. Suppose it finds $5,000.
- It then responds, possibly directly: "Last week (May 12-18), your total sales were **$5,000**."
- (If the user is asking through chat, the answer comes through the chatbot. If the user had clicked a "Weekly Sales" button, the same info might show on a card.) - The data could also be sent as `data: [{date: '2025-05-12', total: ...}, ...]` if they wanted to show a breakdown, but since question was total, a single figure suffices. **User follow-up:** "Great, how does that compare to the week before?"
Bot: (Now the logic will fetch the prior week's sales, say $4,000, and then likely use the LLM to generate a comparison)
- It might formulate a prompt: "We had $4,000 in sales during May 5-11 and $5,000 during May 12-18."
- LLM could respond with: "You saw an increase of $1,000 in sales week-over-week, which is a 25% improvement. The last week's sales were higher than the week before."
- The bot sends this text. The UI could optionally highlight the numbers.
- This showcases combining data with AI for analysis.

- **Example 2: Task/Reminder Creation**
  User: "Remind me to submit the quarterly tax forms next Friday at 10am."
  Bot: (The backend sees "remind" -> intent to create task/reminder)
- It parses out "next Friday 10am" (could use a date parsing library or a quick custom parser). Determine that date (if today is May 23 and next Friday is May 31, 2025, 10:00).
- Save a new entry in `tasks` table: description "Submit quarterly tax forms", due_time = 2025-05-31 10:00, completed = false.
- Call n8n webhook `CreateReminder`: payload includes the date/time and description and user's email perhaps.
- n8n workflow might schedule an action. For instance, n8n could have a **Wait node** that waits until that date/time then an Email node to email the user, or it could create a Google Calendar event via API node. In any case, something will happen at that time (for demo purposes, might just log, but we assume it will send an email).
- Immediately, the chatbot responds: "Sure, I've set a reminder to 'submit the quarterly tax forms' on Fri, May 31 at 10:00 AM. You'll get a reminder email then."
- On the UI, if there's a task list, the new task appears (pulled from DB or the bot's response triggers the UI to refresh the tasks list via API).

- When the actual time comes, n8n will send the email (even if the user isn't chatting at that moment). This demonstrates the asynchronous automation ability. (In a live demo, you can simulate the time or adjust the reminder time to a few minutes ahead to show it working.)

- **Example 3: External Info Request** (if implemented)
  User: "Where is order 12345 right now?" (assuming orders have tracking)
  Bot: The question implies checking shipment tracking. If we prepared a workflow or an API integration:

- Node sees "where is order 12345" -> could call n8n webhook `TrackOrder` with order ID.
- The n8n flow might call a dummy shipping API or just return a made-up status (like "In transit, expected delivery June 1").
- Bot replies: "Order 12345 is currently in transit and expected to be delivered on June 1."
- This example shows how we could plug in other services easily without coding that logic in our backend, thanks to n8n.

Through these workflows, BizAssist Dashboard demonstrates a cohesive system where **AI** provides a conversational interface and analysis, **data integration** provides factual answers from the business's database, and **automation workflows** (n8n) handle actions and external integrations. The architecture scales with the business: more data tables can be added (the chatbot can then be extended to handle them), more workflows can be automated, and the LLM can even be fine-tuned on company data or replaced with an API call to something like OpenAI if needed (the design is flexible to swap the AI provider if desired).

For the developer portfolio, BizAssist shows an understanding of small-business needs and how to address them with technology: - It highlights proficiency in building **dashboard UIs** and combining them with chat interfaces. - It involves writing **complex backend logic** that merges AI and database operations. - It emphasizes the use of **automation tools** to augment the app's capabilities without reinventing the wheel (a practical skill in modern full-stack development). - It also touches on **data analysis and visualization**, which is a plus for demonstrating well-rounded skills.

Project 3: PharmaRep Companion – Pharmaceutical Sales Assistant (Medical/Pharma)

Use Case & Target Audience: PharmaRep Companion is a web application tailored for pharmaceutical sales representatives (sales reps) or medical liaisons working for a pharma company. The use case is in the **pharma/medical domain** but on the industry side rather than patient side. These reps often need quick access to drug information, sales figures for their territory, and support for common questions from doctors. They also have repetitive tasks like logging visit notes or scheduling follow-ups. PharmaRep Companion serves as an AI-powered personal assistant for a rep: it can answer detailed questions about drug products (mechanisms, dosage, clinical trial results), provide sales data on demand, and automate tasks like creating a summary report of a doctor visit or reminding about compliance training. This project demonstrates an integration of domain-specific knowledge (pharma data) with AI and workflow automation, showcasing how an AI chatbot can boost productivity in a regulated industry context.

Branding & Design: The branding here should resonate with pharma professionals: it should appear **trustworthy, efficient, and domain-specific**: - **Project Name:** *PharmaRep Companion* (tagline: "Your AI Assistant for Pharma Success"). The name clearly targets pharma reps and suggests a helpful sidekick. - **Color Palette:** Use colors often found in pharma corporate branding: **Royal blue** (#1A3E8C) and **white** as primary (blue conveys trust, stability, corporate feel), with a secondary accent like **orange or green** (e.g., a vibrant orange #F45D22 or a fresh green #4CAF50) for call-to-action elements to signify innovation and health. These colors maintain a professional vibe suitable for enterprise use. - **Fonts:** Professional sans-serif fonts like *Helvetica Neue* or *IBM Plex Sans*. These give a clean, corporate feel. Possibly use a contrasting font for headings like *Source Sans Pro* to have a slight modern edge. Ensure readability for potentially dense info (like drug details or charts). - **Design Direction:** The UI layout is akin to a productivity tool or CRM: maybe a dashboard interface (since reps might use it on a laptop or tablet). Key design elements: - A navigation menu with sections such as "Knowledge Base", "Q&A Chat", "My Sales", "Tasks/Calendar". - The chatbot is a central feature but integrated with context. For example, if the rep is on the "Knowledge Base" section looking at a drug profile, they can ask a question in the chat about that drug and get an answer. Or in "My Sales", they can query the chatbot for specific numbers. - The chat interface might be on a sidebar or as a floating chat icon (depending on UI decision) – accessible across the app to answer questions at any time. - Use relevant icons: e.g., a pill or molecule icon for drug info, a chart icon for sales data, a calendar icon for tasks. - The look should be **sleek and data-rich**: show tables or stats for sales, have a section to display search results from the knowledge base etc., all in a cohesive style. - Considering pharma might involve a lot of text (drug descriptions, etc.), use collapsible panels or tabs to organize content. The chatbot can also return lengthy information, so the design should accommodate possibly long responses by allowing the chat window to scroll and perhaps copy text.

Technology Stack: Similar full-stack tech with emphasis on data integration and AI: - **Frontend:** A **React/ Next.js** application, likely with a multi-page dashboard structure (similar approach to Project 2 but with pharma context). Next.js helps with routing to different sections and could pre-render certain pages (e.g., static info like drug details page, if we include static content for known products). The UI may incorporate a design system or component library suitable for enterprise apps, such as **Ant Design** or continue with Material-UI. Ant Design has a more enterprise feel and components like tables and collapsible panels which might suit the data-heavy aspects. - **Backend: Node.js/Express** for API and core logic. The backend will integrate with various data sources: - A **drug information database** – this could be a table in Postgres with entries for each product (including description, indications, side effects, etc.), or it could simulate calling an external API (like if a real company might use an internal API for drug data). For the portfolio, storing some sample data in Postgres is simplest. - **Sales data database** – similar to Project 2, maybe tables with sales by region, by product, by quarter, etc. If the rep logs in, the system can filter to their region or clients. - **Document repository** – optionally, reps often have PDFs or slide decks (we might not implement file handling here, but could simulate by having some text from a PDF in the database). - The backend's **MCP logic layer** will be more sophisticated in terms of using the right data: - It will use the LLM (Ollama) for answering free-form questions about products or for generating narratives from data. - It will use direct queries for numeric data (sales metrics). - Possibly incorporate a retrieval mechanism: e.g., using an embedding search for drug info (if not just exact match). However, to keep it simpler, we can just query the DB for relevant info and include it in LLM prompt. - It should also handle user authentication (since this is company data, login is required). We might have a simple login with email & password for the rep, and the backend checks an `users` table. - Also, consider role-based content: a sales rep should only see their data. So queries to sales data are filtered by user's region/territory (assuming the `users` table has a `territory` field). - **Database: PostgreSQL** as usual, with schema for: - `users` (sales reps info, territory, etc.), - `products` (drug info library), - `clients` (doctors/hospitals the rep visits, perhaps), - `sales`

(sales numbers, maybe fields: product_id, client_id, date, quantity, revenue, rep_id linking to user). - `questions` (maybe store common Q&A or previous queries made by the rep, to allow quick recall or to train AI). - Possibly `visits` (if reps log their visits: client, date, notes). - `tasks` or `reminders` similar to earlier projects (like follow-ups). - **AI Integration (Ollama):** The LLM usage here is significant: - We can use an LLM to answer complex product questions. For example, "What are the main side effects of Drug X reported in the Phase 3 trial?" – The system could have the trial results summary in the database and feed that to the LLM to compose an answer, or perhaps the model itself, if large enough, might answer from general knowledge. But since it's local (and likely an open model), better to rely on providing context. - This becomes an example of **RAG (Retrieval-Augmented Generation)**: the Node logic finds relevant info from the `products` table or some documents, then instructs the LLM: "Based on the following info: [insert excerpt], answer the question about side effects..." This ensures accuracy from known data. - For sales questions like "How did product X perform last quarter in my region?", the logic can fetch the number from DB and either directly answer or ask LLM to phrase it nicely and perhaps add context ("It grew 5% from previous quarter"). - The Ollama local model might be the same base model (like Llama2 13B) but possibly loaded with some fine-tuning or just relying on the provided context. We might mention if an open source medical text model is used, but assume base model is fine. - The Node backend again uses the `ollama` NPM client. Given possibly large context (e.g., drug info could be long), the team might consider chunking or summarizing data to fit in prompt limits. - **n8n Automation:** How does n8n fit for a pharma rep assistant? - **Task automation:** e.g., "Schedule a follow-up meeting with Dr. Smith next Wednesday" – triggers n8n to create a calendar event or send an invite email to the doctor (could integrate with Outlook/Gmail). - **Report generation:** "Prepare a weekly report of my activities" – could trigger a workflow that compiles data from DB and emails a report to the manager. However, that might also be done in-app; still, an n8n workflow could format and send it. - **Notifications:** compliance or training reminders delivered via email or Slack if integrated. Possibly the bot could say "Remember to complete your compliance training by Friday" and if asked, "schedule it for me", calls n8n. - **CRM updates:** maybe if rep says "Log that I visited Dr. Green today and discussed Product Y" – the system could save to DB, but to show off integration, maybe an n8n workflow could update an external CRM (Salesforce, for example, if this were real). For demo, maybe just log it in DB and simulate external by an n8n dummy workflow acknowledging it. - So n8n in this project serves to integrate with external corporate systems or to handle scheduling tasks, leaving our core app focused on data and Q&A. - **Deployment:** Will be on Render as well, similar pattern. Possibly because this is an enterprise app, one might think of internal deployment, but for portfolio hosting on Render is fine. Dockerized, environment variables for secrets (like DB URL). If any external APIs (maybe not in demo), their keys would be env secrets as well. - The team ensures that no sensitive data is actually present (it's dummy data), so hosting it is fine. But they can highlight that in a real scenario, further security (SSL, role-based access, maybe self-hosting due to data sensitivity) would be considered.

Architecture & Components: - **Frontend UI Components:** - A main Dashboard page showing key info (e.g., current quarter sales vs target, upcoming tasks). - A navigation menu (could be a sidebar) to go to sections: - **Chat Assistant** (though likely accessible anywhere, perhaps a full-page chat for focus mode). - **Products** – list of products/drugs the rep deals with. Clicking one opens a detail page (with info pulled from DB: description, indications, etc.). On this page, the rep can ask questions specifically about that product, which the chatbot will use the context of that product. - **Clients** – list of doctors/hospitals in the territory. Perhaps can click one to see past interactions or sales to that client. - **Sales Data** – could have some charts or tables (e.g., sales by product, by month). - **Tasks/Calendar** – showing any upcoming reminders or follow-ups the rep set, possibly editable. - (The app might not fully implement all of these, but design as if they exist, to present a broad concept.) - The chatbot interface might be available as a component that overlays on any page (like a chat icon bottom-right that expands to chat, which can interpret context from the current page if needed). Or it can be a fixed panel on the side. - Use of Ant Design components: e.g., `<Layout>` with `<Sider>` for menu, `<Content>` for page content. Use `<Table>` for listing sales or clients. Use `<Collapse>` for sections in product details (like a panel for each data type: description, clinical trial results, etc.). The chatbot could be in a `<Drawer>` that slides out. - The style should remain consistent: the blue/white theme means lots of whitespace and blue headers, with orange or green for highlights (like a “New” badge or important numbers). - Possibly incorporate the company’s logo on top (just an example pharma logo placeholder).

- **Backend & Logic:**
- **Express API:** Many endpoints likely:
    - Auth: `POST /login`, etc.
    - Product data: `GET /api/products`, `GET /api/products/:id` (for details, maybe including related data like trials or FAQs).
    - Sales data: `GET /api/sales?filter=quarter&product=X` etc., or more specific ones like `GET /api/sales/summary` returns aggregated data the dashboard needs.
    - Clients and visits: `GET /api/clients`, `POST /api/visits` for logging a visit note, etc.
    - Tasks: similar to earlier, endpoints to create and list tasks.
    - Chat: `POST /api/chat` – the main one which will handle natural language requests.
- **MCP Logic Layer:** The chat service here might be a bit more advanced:
    - Possibly integrate a framework like LangChain (if one of the AI engineers wants to use it), but we can describe it conceptually without naming that. They might use an approach where the query is classified into categories: “data query”, “knowledge query”, “task command”, etc.
    - If data query (e.g., asking for numbers like sales or some stat) – fetch from DB.
    - If knowledge query (e.g., asking about product info) – fetch from product database or some knowledge base entries.
    - If general query (like asking for advice not directly in data) – use LLM general knowledge.
    - If task command (imperative like "schedule", "remind", "log") – trigger appropriate n8n flow or internal function.
    - To perform this classification, they could either do keyword matching or even use a small ML model or prompt the main LLM to identify the intent first (e.g., first ask LLM "Classify this input into [categories]" then act accordingly – though that adds complexity; likely simpler rules suffice in most cases).
    - **Retrieval Augmentation:** For any knowledge query about products, implement a simple retrieval: search the `products` table for the mentioned drug name (if the question mentions a product) and retrieve relevant fields (like “side_effects” text, “clinical_trial_outcome” text). Provide those to the LLM in the prompt: `"Assistant, use the info: {side_effects: ..., trial_outcome: ...} to answer: What are the side effects of Drug X?"`. This ensures accurate answers, using LLM more to nicely format or synthesize if multiple pieces of info.
    - The system can also have some pre-written Q&A pairs (maybe a table of FAQ for products). If the question matches an FAQ, it could just return that answer (or also feed to LLM to expand).
- **Role of n8n:** As mentioned, integrate tasks:
    - Provide an endpoint in Node for webhooks e.g. `POST /webhook/visitLogged` that n8n might call after doing something, or simpler is Node calls n8n and doesn’t expect a call back but just an action.
    - Workflows examples: create calendar events, send summary email of a visit to the rep’s email or to their manager, etc.

    - For example, after a visit is logged, the rep might say "summarize this conversation and email to my boss" – the LLM could help summarize the conversation note, and n8n could send it via email to the boss. This is multi-step, but doable: Node calls LLM for summary, then calls n8n email workflow with that summary.
    - Another: a nightly n8n workflow could send the rep a briefing every morning (like "Today's plan: you have 2 visits, and here are updates..."). The chat could allow scheduling that (like "Set up daily briefings at 8am" -> triggers an n8n recurring workflow).
- **Postgres Schema Outline (Key Tables):**
    - `products`: fields: id, name, description, indications, contraindications, side_effects, trial_results, etc. (All text fields where we can store paragraphs of info.)
    - `faqs`: if needed, product_id nullable or category, question, answer.
    - `clients`: id, name, specialty (for doctors), location, etc. Possibly link to `sales` or `visits`.
    - `sales`: id, product_id, client_id, rep_id, date, quantity, revenue.
    - `visits`: id, client_id, rep_id, date, notes (text). Maybe a field like `actions` (e.g. "send samples next week").
    - `tasks`: similar to before.
    - The relationships: rep (user) has many clients (that they manage), sales and visits are linked to a rep and client.
    - This data can be filled with sample values: e.g., 5 products with some dummy info (like a cholesterol drug, a diabetes drug, etc.), 10 clients, some sales records, etc., making sure there's enough to make queries interesting.

Development Steps (Highlights): - Start a new Next.js project, set up authentication early (could use NextAuth with credentials provider to simplify login flow in Next). - Build page structure using Next's routing (pages for products, clients, etc.). Use dummy data first to shape the UI. - Implement the Express backend with routes for that data. Protect routes with a simple auth check (if using NextAuth, we can call the backend with session info or use API routes with `getSession`). For simplicity in explaining, mention using an auth middleware that checks a token. - Integrate the chat logic similarly: it might be easier to integrate it as part of Next API route or as an internal function, given the app might consider context of what page user is on (but that could be sent as part of the chat request, e.g., chat request payload can include `context: "product: Drug X"` if user is on a product page). - Set up the database with required tables and fill sample data. Possibly write a seed script for the product info (since likely to be static text). - Connect Ollama: maybe in dev, choose a model. If there's a known biomedical model in Ollama's library (not sure if Ollama hosts like PubMedGPT or something), but can mention if needed. If not, use Llama2 but supply our data as context. - Build n8n workflows and note their URLs for integration. Write them in documentation or store minimal info (maybe environment variables like `N8N_CALENDAR_WEBHOOK` etc., like previous projects). - Implement each feature one by one (product Q&A, sales queries, logging visits). - Write tests for at least the critical logic – e.g., test that asking a known question returns an answer containing expected info (given fixed data). - Use Docker Compose for dev with Postgres and maybe n8n as well. Possibly include an adminer or pgAdmin container to easily view the DB while testing (not necessary to mention, but could). - After finishing, deploy to Render, carefully storing env vars for DB, and if using NextAuth, any secret.

Workflow Description (Examples): - **Example 1: Product Question**
A rep, Alice, is preparing to meet a doctor about *Drug Alpha*. She opens PharmaRep Companion and navigates to Drug Alpha's page which lists key details. Alice asks in the chat: "What's the mechanism of action for Drug Alpha?"

- The context (current product = Drug Alpha) is sent with her question to the backend.
- Backend finds the record for Drug Alpha in `products` table, specifically the "mechanism_of_action" field (or it might be part of description).
- It then forms an answer. Possibly it's a straightforward factual answer stored in DB, so the bot could respond directly: "Drug Alpha's mechanism of action is [the stored info]." If the info is in a paragraph, maybe the LLM is used to shorten or clarify it.
- If the rep asks a more complex question like "How does Drug Alpha compare to Drug Beta in terms of efficacy?", then the logic might retrieve info on both products (their efficacy data from trial_results field) and then feed both to LLM: "Compare efficacy of Alpha vs Beta given: Alpha's trial result X, Beta's trial result Y." The LLM generates a concise comparison, which the bot returns. This shows AI reasoning beyond just DB facts. - The answer appears in the chat panel next to the product info. Alice can copy this info or use it in her discussion.

- **Example 2: Sales Data Query**
  After the meeting, Alice wants to check her progress. She types: "Show me my Q2 sales for each product."
- The system recognizes a sales data query. It knows Alice's rep_id and territory. It queries the `sales` table for Q2 (say April-June) of the current year, grouped by product. It finds: Drug Alpha: $500k, Drug Beta: $300k, etc.
- The bot could respond with a text summary: "In Q2, your sales were $500k for Alpha and $300k for Beta (total $800k)."
- Additionally, the response payload might include a small data table. The frontend then displays a bar chart or a table for clarity. This makes the answer more actionable.
- Alice might ask follow-up: "What's my top-selling product this year?" The logic sums YTD sales by product and responds, possibly with the product name and number. If such questions become common, an FAQ or direct metric could be coded (but AI can handle it with the data too).
- She could also ask "How far am I from my annual target?" If the target is stored (could have a field in users or a config), logic calculates gap and LLM or template says: "You have achieved 60% of your annual target of $2M. You need an additional $800k to reach 100%."

- These show integration of personal data with the chatbot.

- **Example 3: Logging an Interaction & Automation**
  After visiting Dr. Smith, Alice wants to log the visit and set a reminder. She tells the bot: "I visited Dr. Smith today and discussed new study results of Drug Beta. Remind me to send him follow-up materials next Thursday."

- The input is a bit complex; possibly the app might break it down or prompt for structure. But assuming the AI can parse or the rep could do in two commands. Let's split it:
  1. **Log visit**: If Alice says "Log visit with Dr. Smith: discussed new study results of Drug Beta." The bot/logic:
  2. Recognizes keyword "log visit" (or the presence of a known client name Dr. Smith).
  3. It creates a `visits` entry: client=Dr. Smith, rep=Alice, date=today, notes="Discussed new study results of Drug Beta."
  4. It responds: "Noted. I've logged your visit with Dr. Smith on [date] about Drug Beta." Perhaps also, if appropriate, update some `last_contact` field for that client.

5. Optionally triggers n8n if say they want an email to themselves or manager about the visit (some companies do that).
6. **Set reminder**: The "Remind me to send him follow-up materials next Thursday" part indicates a task.
7. The logic either from the same message or separate, identifies a reminder task. Next Thursday's date is parsed (maybe June 5, 10:00 default time).
8. It creates a `tasks` entry or schedules directly. And calls an n8n workflow "Schedule Email Reminder":
    - The workflow could wait until next Thursday then send Alice an email: "Send the brochure to Dr. Smith now."
    - Or it could directly schedule an email to Dr. Smith (if we had his email and pre-set materials).
9. The bot replies: "Alright, I will remind you on Thursday, June 5 to send the follow-up materials to Dr. Smith."

- This interaction shows how multiple systems (DB for logging, n8n for future action) work through one conversational interface, which is a powerful demonstration of productivity gain.

- **Example 4: Knowledge Search**
  Alice recalls there was a question about Drug Alpha's side effects in diabetic patients. She asks: "Were there any notable side effects of Drug Alpha in diabetic patients in the trial?"

- This might not be directly in the structured data; maybe the trial_results text mentions something. The logic might do a keyword search in `products.trial_results` for "diabetic". Finds a sentence about slight increase in blood sugar, for example.
- It then uses LLM to formulate an answer: "In the clinical trial for Drug Alpha, diabetic patients experienced a slight increase in blood sugar levels as a side effect, although it was generally well-tolerated overall [11]." (The citation here is conceptual—if we had source, but likely we won't cite in-app. The idea is it gives a precise answer from the text.)
- This demonstrates handling queries that require pulling specific info from a chunk of text.

From these examples, **PharmaRep Companion** showcases: - The ability to incorporate domain-specific data into an AI chatbot's responses. - A mix of direct database querying and AI-generated language for clarity and completeness. - Integration with workflow automation to handle scheduling and external communication (a very relevant feature for busy professionals). - A sophisticated UI that consolidates data and AI in one tool.

From a development perspective, this project plan is heavy on integrating multiple types of data and ensuring accuracy (which is crucial in pharma). While as a demo it wouldn't involve real proprietary data, the structure mimics what a real solution might do. It's a strong portfolio piece to demonstrate: - **Domain Modeling** (creating a data model for pharma info). - **Retrieval-Augmented AI** (ensuring the AI uses real data from the DB). - **Advanced Workflow Integration** (not just trivial tasks, but potentially connecting to calendars, emails, etc. via n8n). - **Enterprise-grade UI/UX** (navigation, multi-page, secure access). - Considerations for **scaling and compliance** can be mentioned: e.g., if this were real, we'd ensure data is secure, possibly encryption, audit logs of AI responses (pharma compliance), etc., though not implemented, but awareness can be noted in documentation.

Project 4: EasyReserve – Restaurant Reservation Chatbot (Small Business)

Use Case & Target Audience: EasyReserve is a web application for a small restaurant (or a chain of small restaurants) that integrates a chatbot into the restaurant's website to handle **customer inquiries and reservations**. The focus is a **small business** in the hospitality sector. Customers visiting the site can use the chatbot to ask questions (like menu queries: "Do you have vegan options?") and to make reservation bookings conversationally ("Book a table for 2 tomorrow at 7pm"). For the restaurant owner, this automates customer service and reservation management, reducing phone calls and providing quick responses 24/7. It's a demonstration of using AI to improve customer experience for a small business.

Branding & Design: The branding should feel aligned with a restaurant vibe, inviting and clear: - **Project Name:** *EasyReserve* (tagline: "Your Dining Concierge"). This name emphasizes simplicity in making reservations. - **Color Palette:** Depends on a hypothetical restaurant style, but to keep it general: **Burgundy red** (#8C2F39) and **golden yellow** (#E2B54B) for a touch of elegance (common in upscale dining), or if we assume a more casual cafe, perhaps **teal** (#2A9D8F) and **coral** (#E76F51) for a friendly feel. Let's go with burgundy and gold for a classic restaurant feel, with white/off-white backgrounds for readability. These colors will appear in the chatbot interface as well (maybe the bot messages in a light gold bubble, user messages in the burgundy tone). - **Fonts:** Use a stylish yet legible font: maybe *Georgia* or *Merriweather* for headings (to give a touch of class or a "menu" vibe), and a clean sans-serif like *Lucida Sans* or *Open Sans* for chat text and body (ease of reading on screens). - **Design Direction:** The website might be a one-page or few-page site with the chatbot as a prominent feature: - Possibly a hero section with nice imagery of the restaurant, and the chatbot icon "Chat with us" available. - The chatbot could also proactively pop up: "Hi! Need help or want to book a table? ". - The chat window styling: using the brand colors, with perhaps a background image or watermark of a dining theme (faint). - The UI for reservation might also include the chatbot guiding through steps: e.g., it might ask date, time, party size one by one if the user didn't provide all info initially. So a combination of free text and structured input (like maybe quick reply buttons for common times, or a date picker integration) could be used to make it smoother. - The design should remain simple for mobile users, since many will use phones. The chatbot window likely covers most of the screen on mobile when active. - Additional pages or sections: maybe a menu page, contact page. The chatbot could answer questions from those (e.g., if a user is on menu page and asks about a dish, it can respond contextually).

Technology Stack: - **Frontend: Next.js (React)** for the website. It serves the static content (like home, menu, contact info) and includes the chatbot component accessible throughout. Next.js can pre-render the static portions for speed. For the chatbot UI, we might use a library or custom build: it's similar to prior projects, a chat component with input and message display. - Could incorporate a date/time picker library for the reservation flow if we want a hybrid approach (the bot could trigger a date picker pop-up when it needs a date). - Use of a CSS framework or styled-components to implement the theme easily. Might also consider an existing chat widget library for speed (but since it's a portfolio, building it shows skill). - **Backend: Node.js/Express** for handling chat requests and reservation logic. - The backend will integrate with a reservations system – likely just a Postgres database table for reservations. If the restaurant uses something like OpenTable or Google Calendar we could simulate or integrate via n8n, but we'll assume an internal DB holds reservations. - The **MCP logic layer** handles user messages. Many queries could be answered from pre-defined info (like hours, menu, location) rather than bothering the LLM, but we have the LLM to handle arbitrary phrasing. - Specifically: - If question is about menu or hours: those answers could be stored in a knowledge base (like small JSON or DB table) and returned directly, or given to LLM to phrase nicely. - If it's a reservation request: the logic needs to gather details (date, time, number of people, name, contact). This could be multi-turn: the chatbot can ask for missing info. - Once details are confirmed, the backend will create a reservation entry in the database and likely also trigger a confirmation email or message via n8n (e.g., email or SMS). - If user asks something off-topic, the LLM can attempt to answer (like general conversation or compliments). - Possibly incorporate sentiment or satisfaction detection (not required but could be an idea). - The backend may utilize the LLM for natural language understanding of the request (like parsing "tomorrow at 7" to a date). However, using a date parsing library is more deterministic. A hybrid: parse what you can with code, use LLM if something is complex. - Also, since it's customer-facing, the LLM responses should be well-controlled (maybe a smaller model or heavily instructed to be concise and polite). - **Database: PostgreSQL** to store: - `reservations`: id, name, phone, date, time, party_size, special_requests (if any). - `hours` or `info`: could store open hours, etc., or we can just encode that in logic or config. - Possibly `menu_items`: name, description, etc., if we want the chatbot to answer menu questions by pulling from DB. Alternatively, just have a JSON of FAQs for menu items that the LLM can use. But storing in DB allows showing on a menu page too. - `tables` (optional): if we simulate table management (like track how many tables or seats are free), but for demo, we can oversimplify availability (e.g., assume always available until some limit). - **AI (Ollama):** Use LLM for: - Understanding user input in flexible ways. - Answering FAQs in a conversational manner. We might prime it with a system message like "You are a friendly assistant for a restaurant called [Name]. You have the following info: [hours, address, top 3 dishes]. Use it to answer questions. If reservation details are provided, confirm them." - Possibly for casual chat if user engages in small talk ("Do I need to wear a mask?" or "Can I bring a pet?" – these either have known answers or policy). - Likely a smaller model could suffice (for speed), e.g., a 7B model. - The Node backend again calls Ollama's API. For certain straightforward questions, the logic might bypass LLM and use template answers (faster and avoids any hallucination). But it could still route through LLM with the factual context inserted to maintain a single interface for responses (especially if we want the LLM to always produce the final phrasing). - **n8n Integration:** For a restaurant, n8n can be used for: - **Confirmation emails/SMS**: When a reservation is made, trigger an n8n flow to send a confirmation to the user (if email or phone is collected). Twilio or SendGrid nodes can be used here. This offloads sending logic so that we don't have to code SMTP or SMS in our backend. - **Notifications to staff**: e.g., as soon as a reservation is booked, call a webhook that triggers an n8n flow to notify the restaurant (maybe send an email to the restaurant's email or log it in a Google Sheet). - **Daily summary**: Possibly an automated daily 5pm workflow that emails the restaurant owner the list of reservations for the day (the owner might appreciate that). - The chatbot could also allow cancellation or modifications if we implemented that, and n8n could handle sending the update notifications. - Essentially, n8n is used to handle external communications triggered by chat actions. - **Deployment:** As with others, on Render, with Docker. The environment variables would include any API keys used for notifications (like SendGrid API key, Twilio SID/token if used). Keep those in secrets on Render. The rest is straightforward: DB URL, etc.

Architecture Overview: - **User Interaction Flow:** - A user visits the site. The Next.js frontend serves a nice page. The chatbot either auto-greets or waits for user to click it. - User enters queries, which via a client-side script sends to our backend (likely through a Next.js API route or directly to an endpoint). - The backend processes and returns an answer or follow-up question, which the front displays. - If making a reservation, there might be multiple back-and-forth messages to collect all info. - Once confirmed, the chatbot says a confirmation message and triggers behind-the-scenes processes (DB save and external notifications). - **Component Breakdown (Frontend):** - `ChatWidget` component: can be included in all pages. It manages showing/hiding the chat window and holds conversation state. - `ChatWindow`: UI for the conversation (messages list, input box). Also if needed a special UI for selecting dates/times. Maybe implement a simple dropdown of available times if we have logic for availability. - Possibly use context or global state to persist chat state across pages (so if user navigates, they don't lose chat). - Other components: NavBar with restaurant name/logo, pages like `MenuPage` showing items (fetched from DB or static), etc. - CSS styling to match brand, maybe a background image in header to show ambiance. - **Reservation Logic (Backend):** - Manage state of an in-progress reservation per user session. Because chat is stateless HTTP calls, we need to track if a user is in the middle of a reservation. Possibly use a simple in-memory session map (session ID -> reservation being built). Or embed state in conversation by including previous messages. Given this is not too heavy, including the conversation history in each prompt for LLM might suffice to maintain context ("User: I want to book...; Assistant asked for date; User gave date; etc."). - Alternatively, when a user says they want to reserve, Node can create a temp entry in `reservations_temp` or keep it in memory keyed by a session cookie. The next replies fill it out, then finalize. - The LLM can help parse one-message requests like "Book a table for 2 tomorrow at 7pm" directly using e.g. a prompt that extracts details or just let the logic parse using regex for date/time, number. - Use a library like date-fns or moment to interpret "tomorrow 7pm" to an actual timestamp (taking into account the restaurant's timezone). - Check availability logic: could be as simple as check if less than X reservations already at that slot in DB (like if capacity is 10 tables, count existing and if <10, it's available). For demo, one could even skip strict checking. - Once info is ready (name, contact, time, size), the backend writes it to DB and calls any webhooks. - The final chatbot message: "Confirmed! We've booked a table for 2 on May 24 at 7:00 PM under the name John. See you then! (Confirmation sent to your email.)" - **FAQs and Info:** - The logic can have a small repository of Q&A or facts: e.g., store in DB or as constants: open hours, address, contact number, corkage policy, etc. If user asks, fetch and answer. - For menu questions, if someone asks "Do you have vegan options?" we can either have a pre-defined answer "Yes, we have several vegan dishes including X, Y." or if we had a menu table with tags, query those. Let's say we do: a `menu_items` table with a `diet` field, then we can list those. - Using the LLM might allow more fluid answers, e.g. user could ask "What wine would you pair with the steak?" – if we had such info, an advanced thing could be to let LLM answer from general knowledge. But that might be beyond scope and risk wrong answers, so maybe avoid unless we have curated answer. For portfolio, could mention potential to integrate with a recommendation engine or just gracefully answer "Our staff sommelier can help you at the restaurant!" if it's outside known scope.

Development Steps: 1. Scaffold Next.js app and basic pages (home, menu, contact). 2. Implement the ChatWidget with a dummy conversation to test UI. 3. Set up Express backend or Next API route for chat. Decide one approach; possibly Next API is fine as we don't need a separate server. - But for using Docker Compose with a separate Node service, maybe keep it separate Express listening on /api. - Actually, because reservation has multiple steps, having a continuous connection (like WebSocket) could be beneficial. However, it can be done with polling or just sequential calls. Simpler: do one message at a time via HTTP. 4. Database: define and migrate tables for reservations, menu, etc. Write seed data (like 5 sample menu items with dietary tags). 5. Write backend logic: - Create a service for handling a chat message. If there's an ongoing reservation, attach to that context, else treat as new. - Use pattern matching: perhaps define triggers: * If user message contains "book" or "reserve", enter reservation mode. * If contains keywords like "menu", "vegan", then answer from menu data. * If greeting or general, either have canned or let LLM respond friendly. * If unknown, use LLM to attempt an answer or politely say not sure. - Integrate Ollama calls: for natural answers, especially to rephrase or for less structured Qs. - Implement reservation flow: a state machine or sequence: * State 0: not booking -> detect booking intent -> move to State 1 (collect date). * State 1: have date? If not provided initially, ask for date. * State 2: have date, ask for time if not given (or combine with date step). * State 3: ask for party size. * State 4: ask for name (and maybe phone/email). * At each step, validate input (e.g., date is a valid future date, size is a number <= maybe 20). * Use LLM's understanding to parse maybe user's answer. e.g., if asked "For what date?" and user says "Friday next week", could directly parse with date library or possibly use LLM if complex. * Once all info, confirm summary and write to DB. - The above could be done with if/else and a context object stored in memory, or simply by sequential prompting with history. Possibly easier to manage explicitly rather than hoping the LLM tracks it all. - Because it's important to not lose data if user messages come separately, an in-memory session object (like store in a Map with sessionId) is pragmatic. Alternatively, cookie with a sessionId to tie requests. - Implement the known Q&A: if user asks "What are your hours?" fetch from config or DB and answer with LLM or template "We're open Mon-Fri 5-10pm, Sat-Sun 4-11pm," etc. 6. Testing: Simulate conversations: - Ask a menu question, get correct answer. - Do a full reservation conversation, ensure each step works and final entry appears in DB. - Try a combined request "reserve 2 for tomorrow" to see if parsing works (should skip directly to name if we got date/time/size). 7. n8n Workflows: - Create one for "Send confirmation email": trigger via webhook, use SendGrid (need API key) to email the customer. The Node backend calls this after saving reservation. Or could also use Twilio for SMS if phone given. - Create one for "Notify restaurant": maybe just an email to owner or Slack message if we imagine. - Test these by calling the webhooks manually or via code. 8. Deploy: as before, using Render. Possibly we need the environment for email API keys. We also add environment var for base URL of n8n if needed.

Workflow & Example Interaction: - **Example 1: Reservation via Chat**
User: "Hi, can I book a table for 4 people tomorrow night?"
Bot: "Hello! Sure, I can help with that. For **4 people tomorrow night**, I have a couple of questions: What time tomorrow would you like the reservation?" (The bot recognized date=tomorrow, party=4, missing exact time.)
User: "7 pm, please."
Bot: "Great. A table for 4 tomorrow at 7:00 PM. Can I have your name for the reservation?"
User: "It's under Alex."
Bot: "Thanks, Alex. And a contact phone or email?"
User: "123-456-7890."
Bot: "Perfect! I've booked your table for 4 on [Date] at 7:00 PM under the name Alex. You'll get a confirmation at your phone. Looking forward to seeing you! 🍽️"
(Meanwhile, backend wrote the reservation to DB, and triggered n8n. n8n sends an SMS via Twilio to 123-456-7890 saying "Your reservation at [Restaurant Name] is confirmed for [Date] 7:00 PM for 4 people. See you soon!")
- This covers multi-turn flow and external notification.

- **Example 2: FAQ and Info**
  User: "What are your hours today?"
  Bot: The backend sees "hours today" -> it knows today's day of week (say Tuesday) -> looks up hours (like Tue: 5-10pm).
  Bot: "We're open today from 5:00 PM to 10:00 PM. Can I help you with anything else, like booking a table?" (Bot gives info and a friendly prompt to drive action.)
  User: "Also, do you have parking?"
  Bot: If we had that in an FAQ entry: "Yes, we offer free parking in the lot behind our restaurant." (straight from DB or config).
  Possibly the LLM could be used to vary the phrasing. For example, if the info is static, not necessary, but we can feed "parking: yes, free lot" into LLM with a system prompt "User asked about parking, answer politely." So the answer might be "Yes, we do! We have free parking available in the lot right behind our restaurant." This shows handling multiple simple queries seamlessly.

- **Example 3: Menu question**
  User: "Do you have vegan options?"
  Bot: If we've stored some menu info, e.g., a few dishes marked vegan:

- The logic finds 3 items are vegan: "Grilled Vegetable Platter, Tofu Stir Fry, Vegan Brownie".
  Bot: "Absolutely! We have several vegan options, including a Grilled Vegetable Platter, a Tofu Stir Fry, and even a Vegan Brownie for dessert. "
  (It lists them nicely, courtesy of either a template or LLM formatting the list with a friendly tone.)
  User: "That sounds great. What about gluten-free?"
  Bot: Similarly, list or mention gluten-free items.

- **Example 4: Cancellation or Modification (if implemented)**
  If user says "I need to cancel my reservation for Alex tomorrow",

- The bot can find that in DB by name/date, remove it, and confirm.
- Possibly require a phone or confirmation code if multiple Alex (but we can assume unique in context).
- Then trigger n8n to send a cancellation confirmation.
- This shows two-way management, but optional for initial scope.

EasyReserve demonstrates: - **Customer-facing AI integration** (requires careful natural language handling and good UX). - Understanding of multi-turn conversation design and state management. - Use of AI for both understanding and responding cheerfully within a specific domain (restaurant). - Integration with real business processes (reservation booking). - This project is highly visual (immediate user-facing UI) and can be very impressive in a portfolio because one can show a video of booking via chat as a tangible, relatable scenario.

From a developer perspective, one learns about: - Handling partial information and guiding a user (which is a big part of conversational UX). - Combining rule-based logic (for critical transaction details) with AI (for flexibility and language). - Ensuring reliability (if AI mis-parses, have fallbacks or clarifications). - Small business needs like sending confirmation messages and maintaining a bookings database. - Also, dealing with date/time parsing, which is a real-world challenge in such apps.

Project 5: MarketMuse AI – Small Business Marketing Content Generator

Use Case & Target Audience: MarketMuse AI is a web application aimed at **small business owners or marketing teams** that need help generating marketing content and managing their marketing tasks. The focus is to integrate a chatbot (AI assistant) that can brainstorm and create content (like social media posts, product descriptions, blog outlines) and also trigger automation for scheduling or publishing this content. This is a small business use case (marketing domain) showcasing how an AI can serve as a creative partner and an automation trigger in one. For example, a user can ask, “Generate a Facebook post about our new cafe opening,” and the chatbot will create a post and even schedule it via an integration (n8n or otherwise). This project highlights the synergy between creative AI generation and business automation.

Branding & Design: The branding should feel creative, energetic, but still professional enough for business: - **Project Name:** *MarketMuse AI* (tagline: “Your Marketing Idea Machine”). The name suggests creativity (muse) and marketing focus. - **Color Palette:** Vibrant and inspiring colors. Perhaps **purple** (#8E44AD) as a nod to creativity and wisdom, and **orange** (#F39C12) for energy and attention. These two are complementary in vibe (purple = creativity, orange = action). Use white as a background to keep it clean, with these colors for accents like buttons, highlights, and chatbot avatar. - **Fonts:** Use a modern sans-serif that has a friendly tone, such as *Google’s Product Sans* or *Nunito*. Could pair with a handwriting or script font for the logo to represent the muse aspect, but mainly keep UI text in sans-serif for readability. - **Design Direction:** The interface is dashboard-like but with a creative twist: - Possibly a canvas area where generated content is displayed with formatting (like a preview of a social post or blog snippet). - The chatbot interface could be central: maybe the homepage is essentially the chat where you converse about marketing ideas. - Include supportive visuals: maybe icons or illustrations of lightbulbs, quills, or social media icons to signify content creation. - UX should encourage iteration: e.g., after the AI produces content, user can say “make it shorter” or “add a more cheerful tone”. This implies we might have buttons for those adjustments or simply instruct the user to ask in chat. - Also incorporate an output panel or history of generated content, where the user can copy or save it. - Since automation is part, there might be a sidebar with toggles or settings: e.g., connect your social media accounts (not actually implementing OAuth in demo, but we can simulate hooking to an API). - Could have templates: e.g., quick prompts like “Write a promo tweet” that user can click to auto-ask the bot.

Technology Stack: - **Frontend:** Built with **Next.js (React)**. This allows building a multi-section UI if needed: - A main Chat/Content section where the conversation and output happens. - Possibly a section or modal for scheduling (like showing a calendar interface if scheduling posts). - The UI likely needs to show formatting for content (maybe the AI returns some Markdown or simple markup for bold, emojis, etc.). The frontend can render that accordingly. - Use of a rich text editor component (maybe something like Draft.js or TipTap) could be a stretch goal so that user can fine-tune the AI output manually before accepting it. - But for MVP, we can just display text and allow copy. - Possibly integrate a library for date/time picking for scheduling posts. - Also, since content might be multi-line (like a short blog outline with bullet points), ensure the chat bubbles can handle that nicely (maybe larger bubble or a different section). - The style should differentiate AI content vs user instructions clearly (maybe AI content bubble has a colored border or background). - **Backend: Node.js/Express**. The tasks: - Use the LLM to generate content based on user prompts. Possibly use specialized prompts or even fine-tuned models for marketing if available (for now, assume general model is fine). - Possibly implement different modes: the user could specify what content type (“tweet”, “Instagram caption”, “product description”) and the backend formats a prompt accordingly (like having a prompt template per content type). - Manage some state like the last generated content (for editing or reusing). - The backend also should handle integration with external posting/scheduling: - e.g., if user says "schedule this on Facebook tomorrow 10am", the backend would confirm and then trigger an n8n flow that is connected to Facebook’s API or a placeholder that simulates it. For demo, since we cannot actually post to FB without complex OAuth, we might simulate by just storing in a `scheduled_posts` table or sending to an n8n that maybe logs it (or posts to a dummy endpoint). - Similarly for other platforms. Possibly just simulate one platform for simplicity. - Use **Ollama LLM** for content generation. Content can be lengthy, but manageable. The LLM should be given instructions to produce the desired format and tone. For example, if user said "Make a Facebook post about our new cafe", backend might issue to LLM: "You are a marketing copywriter. Write a friendly and engaging Facebook post announcing the new cafe opening. Keep it under 100 words." Then LLM output is the post text. - We might allow follow-up edits: if user says "shorter", we can either track the original prompt and ask LLM to shorten it, or instruct LLM with "shorten the previous text". - So conversation context is important here too; we can include the last response in prompt if user asks to modify it. - Additional tasks: summarizing content, listing ideas, etc. All LLM tasks basically. - **Database: PostgreSQL**: - To store generated content if needed (maybe to have a history of content pieces). - `contents`: id, user_id, prompt, response, timestamp. - `scheduled_posts`: id, content_id (ref to content), scheduled_time, platform, status. - Possibly `templates` or `prompts` if we have some predefined categories the user can choose (like "product launch tweet template" which is essentially a prompt stub). - `users` if there's login. This app might be presented as single-user for portfolio demonstration, or one could implement a simple sign-in (but for demo maybe not needed). - **AI Integration (Ollama):** - The LLM usage is core. Possibly the team might consider using a larger model or even hooking OpenAI API if allowed, because marketing content often benefits from high creativity and fluid language. But since focus is on Ollama: maybe they could use an open model like Llama2 13B which can produce decent text, or an instruct model. Quality might not be GPT-4, but fine for demo of concept. - May need to enforce style guidelines in system prompt if needed (like instruct not to include certain things). - The LLM will produce the raw text content. The backend might do a little post-processing (e.g., add hashtags suggestion if needed, or ensure no forbidden words if we care about brand safety). - Possibly integrate a second AI step: for example, for long blog outlines, maybe after outline is generated, the user can say "expand point 2", then the system might call LLM again to generate a paragraph for point 2. So multiple AI actions in one flow (though not mandatory, just an example of interactive content refining). - **n8n Automation:** The automation in this context is about distribution: - **Scheduling Social Posts:** user could instruct scheduling content to platforms. We can set up n8n workflows for each platform: * For example, a "Post to Twitter" workflow where a webhook triggers it, and it uses Twitter API (we might not have keys but we can simulate with a dummy API or just log the action). * "Post to Facebook" similarly. * Or a generic "schedule email to team" that sends the content to someone for approval. - **Newsletter or Email Drafting:** If user says "draft a newsletter and email it to me", n8n could handle sending the email (with content from AI). - **Multi-platform posting:** maybe if they say "post this everywhere at 5pm", we trigger workflows for multiple platforms. - The Node backend would parse such commands in user input or through UI actions and call appropriate n8n webhooks. - The app could also list scheduled posts from DB and allow canceling them (that would require integration: Node could call an n8n flow to delete a scheduled event, or if n8n isn't storing them and just relying on CRON, maybe Node just stores and at runtime n8n flows query DB). - Simplest: schedule via DB and have n8n periodically check DB for due posts, then post. But that adds complexity. Alternatively, when scheduling, Node calls n8n which uses something like a Wait Until node or schedule trigger to execute at the right time (n8n has a "Schedule Trigger" but that might not be dynamic per user). - Possibly easier: Node sets a delay via setTimeout (but that won't persist if server restarts or if multiple instances). Could rely on a cron in Node or directly use DB with a cron job. - However, since we want to show n8n usage, maybe we say n8n has workflows waiting on specific times by being triggered with "Wait". - For demonstration, we might not fully implement scheduling in realtime but simulate scheduling to show concept: - e.g., user says schedule for tomorrow, we log it and say "Sure, will post tomorrow." In demo, one could manually trigger that piece or just trust. - But the design shows how it *would* be done.

Architecture & Implementation Highlights: - **Frontend Interactions:** - The user enters either an open prompt or uses a UI like selects "platform: Facebook" and "goal: promotion" then enters details. Could implement this for guided prompting (like a form that creates a fancy prompt for LLM). - Then the response appears. The user might have options: "Use this", "Regenerate", "Edit", "Schedule". - If "Use this", maybe just saves it somewhere. - If "Regenerate", calls LLM with same prompt again (maybe user didn't like the result). - If "Edit", allows user to modify text directly in an editor on screen. - If "Schedule", asks for time/date and platform if not already. Then sends to backend to schedule (calls n8n). - These functionalities mean the front-end has to manage some state of the current content piece, and modals for scheduling. - We can keep it simpler in description: maybe just focus on chat-based usage, with user instructing via text including scheduling. - **Backend Intelligence:** - Perhaps define a set of command triggers. E.g.: - If user message starts with "Generate" or "Write" or "Draft", treat it as a content generation request. If they mention a platform, include that in prompt (as style). - If user says "Schedule this for [time]" and we have a last generated content, link them. - We might incorporate user accounts if scheduling should actually send (to authenticate with their accounts) but skip that complexity. Instead, simulate posting by emailing or logging content. - The logic can be somewhat linear because it's mostly one-turn interactions (the user asks for content, gets it). Multi-turn is mostly refinement which can be done by further prompts. - It's different from reservation where multi-turn was to gather different info. Here multi-turn is more iterative refinement which an LLM can handle as conversation: * e.g., user: "Make it funnier." We keep the previous AI output and prompt: "The user wants the previous text to be funnier. Here's the text: ... Make it more humorous." The LLM returns a revised version. This requires storing previous output or including conversation in prompt (which using conversation history can do). - The conversation can be maintained in state or via giving the entire conversation to LLM each time (which could be heavy if output is large; maybe just last piece and instruction). - **Content Templates and DB**: If we had common requests, we could have templates. For example, a user might choose "Twitter" from UI, then our system knows to prompt with "in 280 characters or less" constraint. This knowledge can reside in code or config rather than DB. - Save each result in DB so user can retrieve earlier ideas (though if no login, just store in session memory). - **DevOps considerations:** - Because content generation can be resource heavy, running the LLM on server might need enough memory. For local dev, it's fine. - Use Docker to containerize if we can incorporate a model. But maybe we rely on an external if heavy. - In any case, on Render, if we were truly deploying, we might have to use smaller model or accept it's slower. - But since it's demonstration, maybe not all content generation requests have to be done live if we precomputed something, but ideally show live generation.

Development Steps: 1. Set up Next.js project. Perhaps integrate a UI library like Tailwind or Chakra for quick styling (color theme). 2. Build core UI: chat area, input box, a panel or two for scheduling or output listing. 3. Backend with Express or Next API to handle requests. Possibly separate to allow easier multi-step processing (like if wanting to stream results etc., though streaming isn't necessary). 4. Implement the LLM prompt strategies and integration using `ollama`. 5. Build a few sample prompts/test the generation locally to fine-tune the system prompts for good output. 6. Implement scheduling logic in backend: - Possibly create a dummy integration: For instance, use Node to send an email with the content at schedule time (like using node-schedule to schedule a job if environment persists). - Or utilize n8n: set up a webhook for "schedule post" where Node calls like `POST /webhook/postToFacebook` with content and datetime. The n8n workflow then might schedule itself (maybe by delaying inside the workflow) and then "simulate posting". - Could also just have n8n send an email at the scheduled time to simulate the post going out (like an email to user "Pretending to post: [content]"). - Document that with real API keys it could directly post. 7. Connect the pieces: on front-end, after generation, allow scheduling which triggers the above logic. 8. Test end-to-end: - Ask for a piece of content, get it. - Ask to tweak it, see it updated. - Schedule it (maybe schedule just 1 minute in future for test) and verify n8n or Node actually does what it's supposed to (maybe look at n8n execution or check email). 9. Write documentation, including perhaps examples of prompts and results. Possibly include example outputs in the documentation to illustrate the capability.

Workflow Example: - **Example 1: Generating and scheduling a social media post**
User: "Write an Instagram post about our new fall collection of handmade scarves. Include a couple of hashtags."
Bot: (LLM processes with instruction to be Instagram-appropriate)
Bot: " Exciting news! Our new fall collection of handmade scarves has arrived! Each one is crafted with love

and warm, cozy vibes. Perfect for those chilly nights #FallFashion #HandmadeWithLove"
(The response is shown, maybe with some emoji and hashtags as requested.)
User: "Great, schedule this post on Facebook for tomorrow at 9am."
Bot: "Sure! I'll schedule the post for Facebook tomorrow at 9:00 AM."
- Backend receives the schedule request. It creates an entry or triggers n8n. n8n might confirm via its response which Node passes to user in chat. - If using n8n: Node calls `POST /webhook/schedulePost` with content and time and platform=Facebook. n8n workflow sets up a delayed execution (perhaps using its Wait node until that time, then a dummy post action). Immediately, n8n can return a 200 OK which Node interprets as success. - The chat is updated with the confirmation message. - (The next day at 9am, the n8n flow runs: maybe it sends the content to an email or a log to simulate posting. This won't be seen by user in chat unless we wanted to integrate that, but it's an offline effect.)

- **Example 2: Brainstorming and refining**
  User: "I need ideas for a blog post about the benefits of organic coffee for health."
  Bot: (LLM returns say 3 bullet point ideas or a short outline)
  Bot: "How about a blog post titled *“Brewing Good Health: The Benefits of Organic Coffee”*? It could cover points like:
- Rich in antioxidants, boosting your immune system.
- No harmful pesticides, better for health and environment.
- Supports sustainable farming communities.
  Would you like me to draft a section or expand on any of these points?"
  User: "Expand on the antioxidants part."
  Bot: (LLM takes that bullet and writes a paragraph)
  Bot: "*Rich in Antioxidants:* Organic coffee is packed with antioxidants that fight inflammation and cell damage. These antioxidants, such as polyphenols, can help protect your body from diseases and support overall wellness. By choosing organic, you're ensuring maximum antioxidant content without chemical interference, giving your immune system a natural boost."
  User: "That's great. Now generate a concluding sentence."
  Bot: "In summary, savoring a cup of organic coffee isn’t just a delightful morning ritual – it’s an investment in your health and the well-being of the planet."
- This showcases iterative content creation. The UI would show each of these as messages or maybe allow assembling them.

- Perhaps we might build a feature where user can click "Insert into editor" to compile into one article. But that's a bonus.

- **Example 3: Multi-platform adaptation**
  User: "Create a product description for our new smartwatch. Then give me a shorter version for Twitter."
  Bot:

    - First output: a paragraph describing the smartwatch features, maybe ~100 words.
    - Then sees "shorter version for Twitter" – which might be a follow-up the user expects as a separate output. The bot (with context of previous content) might output: "Twitter version: 'Meet our new SmartX Watch – your fitness coach, personal assistant, and style statement in one. Stay connected and track health, all from your wrist! #SmartXWatch #NewRelease' (119 characters)"

User: "Schedule the tweet for Monday 10am and email the longer description to the sales team."
Bot: "Got it. I'll schedule the tweet for Monday at 10:00 AM and send the full product description to the sales team email."

- Back-end triggers n8n for tweet scheduling and maybe another for sending an email (perhaps using SendGrid to company email with the description text).
- Possibly it asks for confirmation of sales team email address if not on file; but maybe in config we have an email list.

MarketMuse AI as a project underlines: - The creative use of AI for content generation. - Practical integration where that content doesn't just stay in the app, but flows out to channels (via automation). - It demonstrates to portfolio reviewers the candidate's understanding of how AI can streamline marketing, a common need, and how to implement it technically (with emphasis on prompt design and integration with external APIs). - The project also highlights an engaging UI/UX – because dealing with content requires a nice interface for reading and editing text, beyond just plain chat, which adds complexity (like ensuring formatting is preserved, copy to clipboard functionality, etc.).

Each of these five projects – MediConnect AI, BizAssist Dashboard, PharmaRep Companion, EasyReserve, and MarketMuse AI – presents a distinct scenario with unique challenges and showcases a wide range of skills. From handling confidential medical guidance to parsing reservation details, querying databases for analytics, or generating creative content, the common thread is the integration of a **JavaScript full-stack** solution with an **AI chatbot (via Ollama)** and **n8n automation workflows**.

By implementing these projects, a developer not only demonstrates proficiency in modern web development (React, Node, databases, deployment) but also the ability to harness AI and automation to solve real-world problems in both **pharma/medical** and **small business** contexts. Each project is designed to be portfolio-ready: with attractive branding, thorough documentation, and realistic functionality that could be demoed to potential employers or clients to illustrate the developer's capabilities.

Sources: The design and implementation strategies reference best practices in software and AI integration. For example, Docker Compose is used to simplify development with multiple services [5], environment variables and secrets are utilized for configuration [7], and an MCP (context protocol) approach is considered to standardize LLM tool use [1]. The importance of webhooks in connecting the Node backend with n8n is highlighted, as *"In n8n, webhooks let you start a workflow from outside... send a request to a special link, and your workflow will run."* [4]. For AI integration, **Ollama** provides a local LLM server, with an NPM client that *"allows you to use JavaScript to interact with the Ollama server API"* [3], ensuring our Node apps can seamlessly retrieve AI-generated responses. Each project plan applies these technologies in a targeted way, illustrating not just theoretical knowledge but practical application in building modern AI-enhanced web services.

---

[1] [11] n8n with MCP. Small notes for n8n and MCP | by Pelin Balci | Apr, 2025 | Medium
https://medium.com/@balci.pelin/n8n-with-mcp-13e9ba0c10c9

[2] [3] [8] Creating a Recipe Recommendation Chatbot with Ollama and Twilio | Twilio
https://www.twilio.com/en-us/blog/creating-recipe-recommendation-chatbot-ollama-twilio

[4] n8n Webhook. My notes on creating a Webhook in n8n | by Pelin Balci | Apr, 2025 | Medium
https://medium.com/@balci.pelin/n8n-webhook-ec9de8e4200c

[5] docker-compose for nodejs and postgres connection by MathurUtkarsh · Pull Request #321 · docker/awesome-compose · GitHub
https://github.com/docker/awesome-compose/pull/321/files

[6] [9] [10] How to Host Node.js Server and PostgreSQL DB on Render | by Aditya Jaiswal | Medium
https://medium.com/@adityajaiswal094/how-to-host-node-js-server-and-postgresql-db-on-render-fd345e9373e2

[7] Environment Variables and Secrets – Render Docs
https://render.com/docs/configure-environment-variables