Medcor Chatbot

[Technical Architecture]

1. Introduction

The Medcor AI Chat Widget is a powerful, plug-and-play solution that adds an avatar-driven, voice-enabled, real-time chatbot to any website. It's built with React and integrates with services like HeyGen for avatars, ElevenLabs for voice, Deepgram for speech-to-text, and DeepAR for face filters.

2. Features

Here's what the Medcor AI Chat Widget offers:

- **Easy Embedding**: Integrates into your website with a single script tag.
- **Real-time Avatar**: Features live avatar video streaming via HeyGen.
- **Voice Interaction**: Supports both voice-to-text and text-to-voice functionalities.
- **AI Chatbot**: Powered by the Medcor API for intelligent conversations.
- **Appointment Booking**: Allows users to book appointments.
- **Doctor Portfolio**: Provides access to doctor profiles.
- **Face Editor**: Includes AR-based enhancements for a bit of fun.

3. Getting Started

Embedding the Widget

To add the widget to your website, simply include the following script tag:

```
<script
src="https://app.medcor.ai/static/chat_widget/dist/chatWidget.min.js?uuid=YOUR_U
UID"></script>
```

Make sure to replace YOUR_UUID with your unique identifier.

How It Works

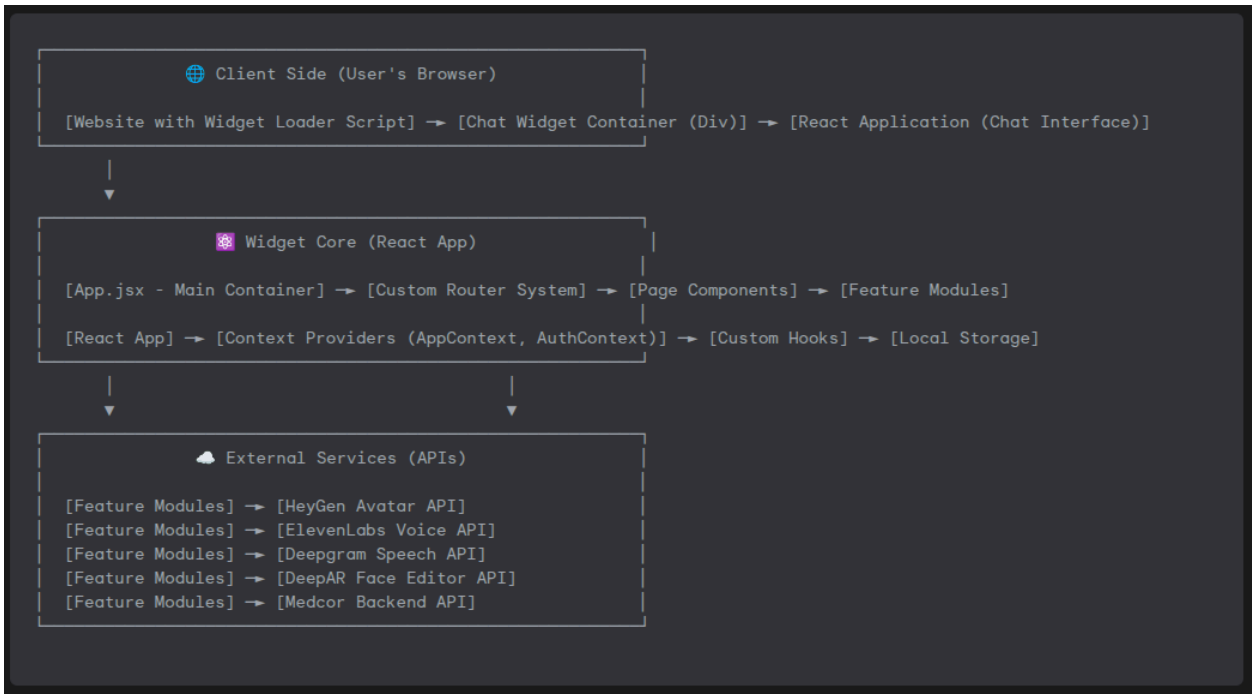
1. The script extracts the uuid from the script URL and stores it in localStorage.
2. It then initializes the avatar connection.
3. A is dynamically created on your page to host the widget.
4. The React application loads within this , rendering the chat interface.

4. Architecture Overview

Understanding the widget's architecture is key to working with it effectively.

4.1. High-Level System Architecture

This diagram shows the overall system components and their interactions:

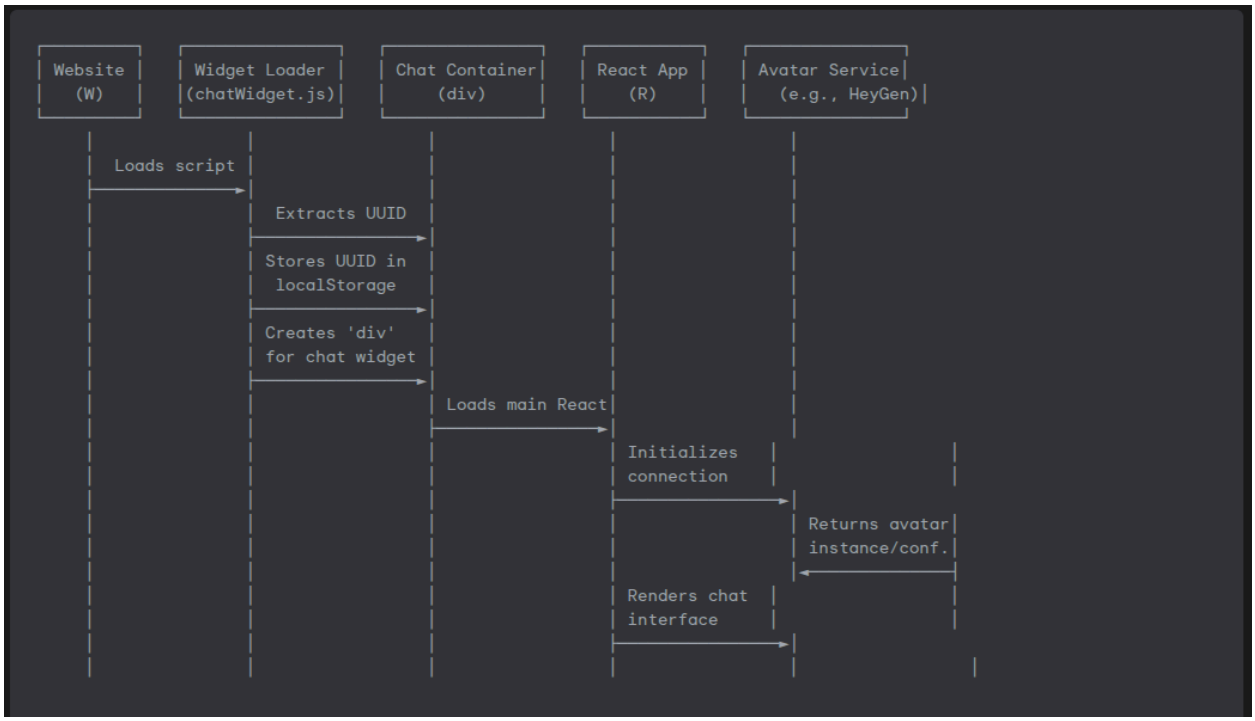


Explanation:

- **Client Side**: The user's website loads the widget loader script. This script creates a container and loads the React application.
- **Widget Core**: The React app, managed by App.jsx, uses a custom router to display different pages and feature modules.
- **External Services**: The widget interacts with various third-party APIs for its core functionalities like avatar streaming (HeyGen), voice synthesis (ElevenLabs), speech recognition (Deepgram), face filters (DeepAR), and backend operations (Medcor API).
- **State Management**: React Context and custom hooks manage the application's state, while localStorage persists some data like the UUID.

4.2. Widget Embedding Architecture

This sequence diagram details how the widget loads onto a webpage:

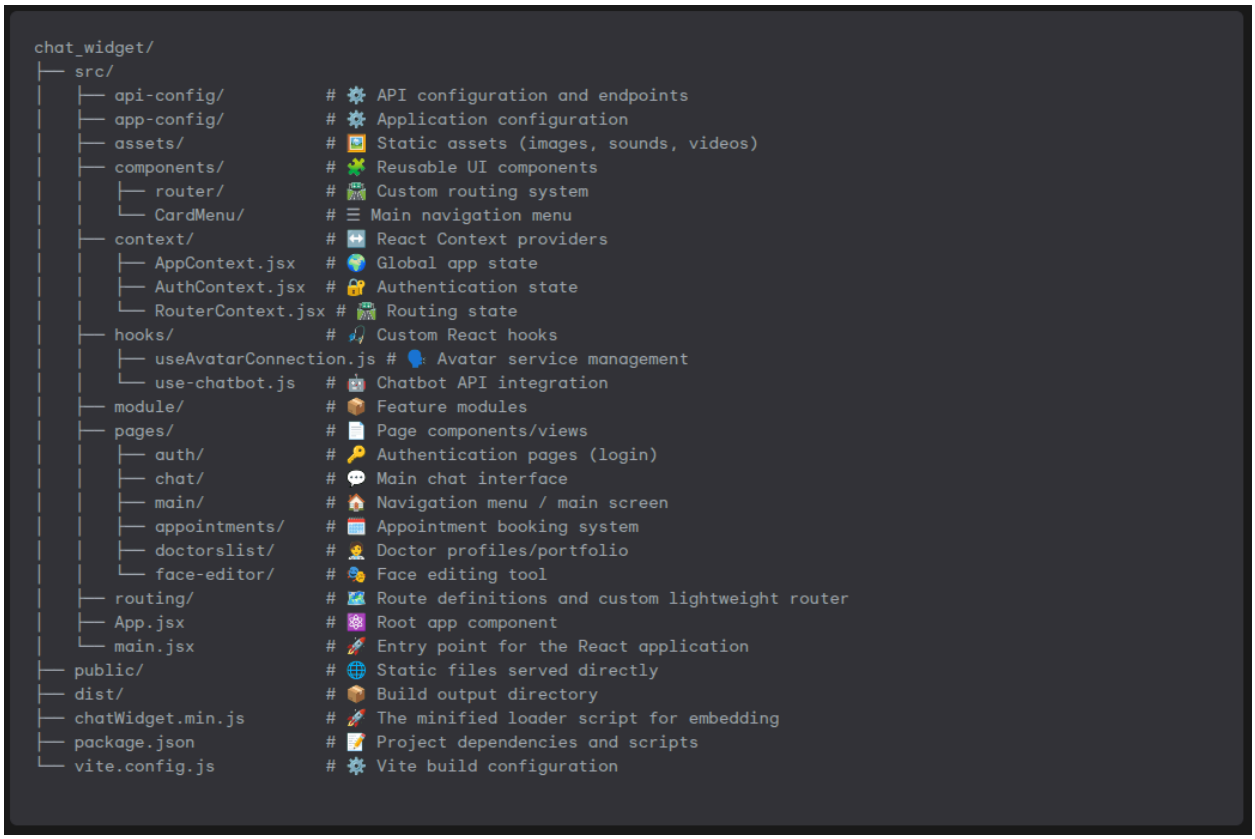


Explanation:
The host Website includes the chatWidget.min.js script, passing a UUID.
The Widget Loader script extracts this UUID, saves it to localStorage, creates a div on the page, and then loads the main React App.
The React App initializes services like the Avatar Service and then renders the chat UI into the created div.

5. Project Structure

The project is organized to keep code manageable and maintainable.

5.1. Directory Architecture



5.2. Component Hierarchy

This diagram illustrates how the main components are nested:

Explanation:
App.jsx is the top-level component, wrapping everything in AppContext.Provider for global state.
It conditionally renders either a Chat Bubble Button (if the chat is closed) or the main Chat Container (if open).
The Chat Container holds the header and AppRoutes, which uses the custom Router to display different Page Components like the Chat Page, Auth Page, etc.
Pages like the Chat Page and Main Menu Page are composed of smaller, specific UI components.

6. Core Components & Flows

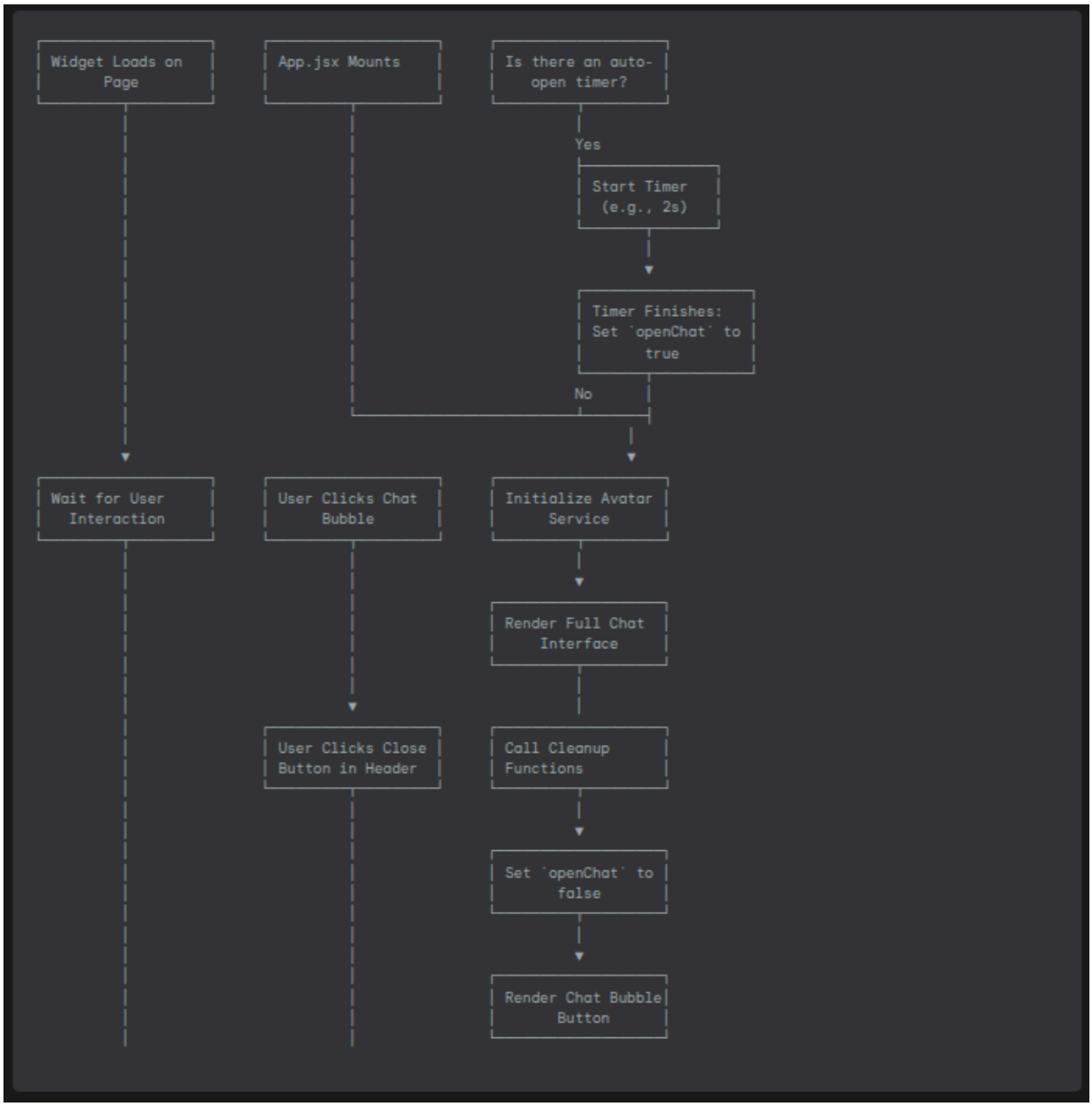
Let's dive into some of the most important components and their functionalities.

6.1. App Component (src/App.jsx)

The App.jsx file is the heart of the widget.

- **Purpose**: It's the main application container and manages the overall state of the chat widget, especially its visibility.
- **Key Features**:
  - Controls chat open/closed state (openChat).
  - Initializes the avatar connection using useAvatarConnection hook.
  - Wraps the application in global context providers (like AppContext).
  - Features an auto-open timer that can open the chat after a delay (e.g., 2 seconds).
- **State Management**:
  - openChat (boolean): Manages if the chat window is visible.
  - avatarService: Holds the instance and methods for interacting with the HeyGen avatar.
  - deepAR: A ref to the DeepAR instance for the face editor.

Component Flow (Opening/Closing Chat)



Explanation:
When App.jsx mounts, it might start an auto-open timer.
If the timer finishes or the user clicks the chat bubble, openChat becomes true, the avatar initializes, and the chat interface appears.
Clicking the close button triggers cleanup of services (like avatar and DeepAR) and hides the chat interface.

6.2. Custom Router System (src/components/router/ and src/routing/)

The widget uses a custom-built, lightweight routing system instead of relying on external libraries like React Router.

- **Components**:
  - Router.jsx: The main container that holds the routing logic.
  - Route.jsx: Defines a specific path and the component to render for that path.

    - Link.jsx: Used for navigation, similar to an <a> tag but for internal routing.
- **Route Configuration** (Example from AppRoutes or similar setup):

```
// Likely found within a component that sets up the routes, e.g.,
AppRoutes.jsx
<Router>
  <Route path="/" component={ChatPage} />
  <Route path="/auth" component={AuthPage} />
  <Route path="/main" component={MainMenuPage} />
  <Route path="/doctor-portfolio" component={DoctorsListPage} />
  <Route path="/appointment-booking" component={AppointmentsBookingPage}
/>
  <Route path="/face-editor" component={FaceEditorPage} />
</Router>
```

This setup maps URL paths (like /main) to their corresponding React components.

6.3. useAvatarConnection Hook (src/hooks/useAvatarConnection.js)

This custom hook is responsible for all interactions with the HeyGen streaming avatar service.

- **Purpose**: Manages the lifecycle and communication with the HeyGen avatar API.
- **Key Features**:
    - **Initialization & Cleanup**: Connects to the avatar service when the chat opens and cleans up the connection when it closes.
    - **Video Stream Management**: Handles the avatar video stream.
    - **Speaking Control**: Provides methods to make the avatar speak (speak()), interrupt (interruptSpeaking()), and stop speaking.
    - **State Monitoring**: Tracks connection status (e.g., 'disconnected', 'connected', 'speaking').
- **Service Methods** (conceptual structure):

```
const avatarService = {
    avatar: null, // Stores the HeyGen avatar instance
    videoEvent: null, // Handles video related events
    connectionStatus: 'disconnected', // e.g., 'connecting', 'connected',
'error'
    initializeAvatar() { /* ... connect to HeyGen ... */ },
    speak(text, options) { /* ... send text to HeyGen for avatar to speak ... */
},
    interruptSpeaking() { /* ... stop current speech ... */ },
    cleanup() { /* ... disconnect from HeyGen, release resources ... */ },
```

```
    subscribe(callback) { /* ... allow components to listen to avatar events ...
*/ },
    unsubscribe(callback) { /* ... remove listeners ... */ }
};
```

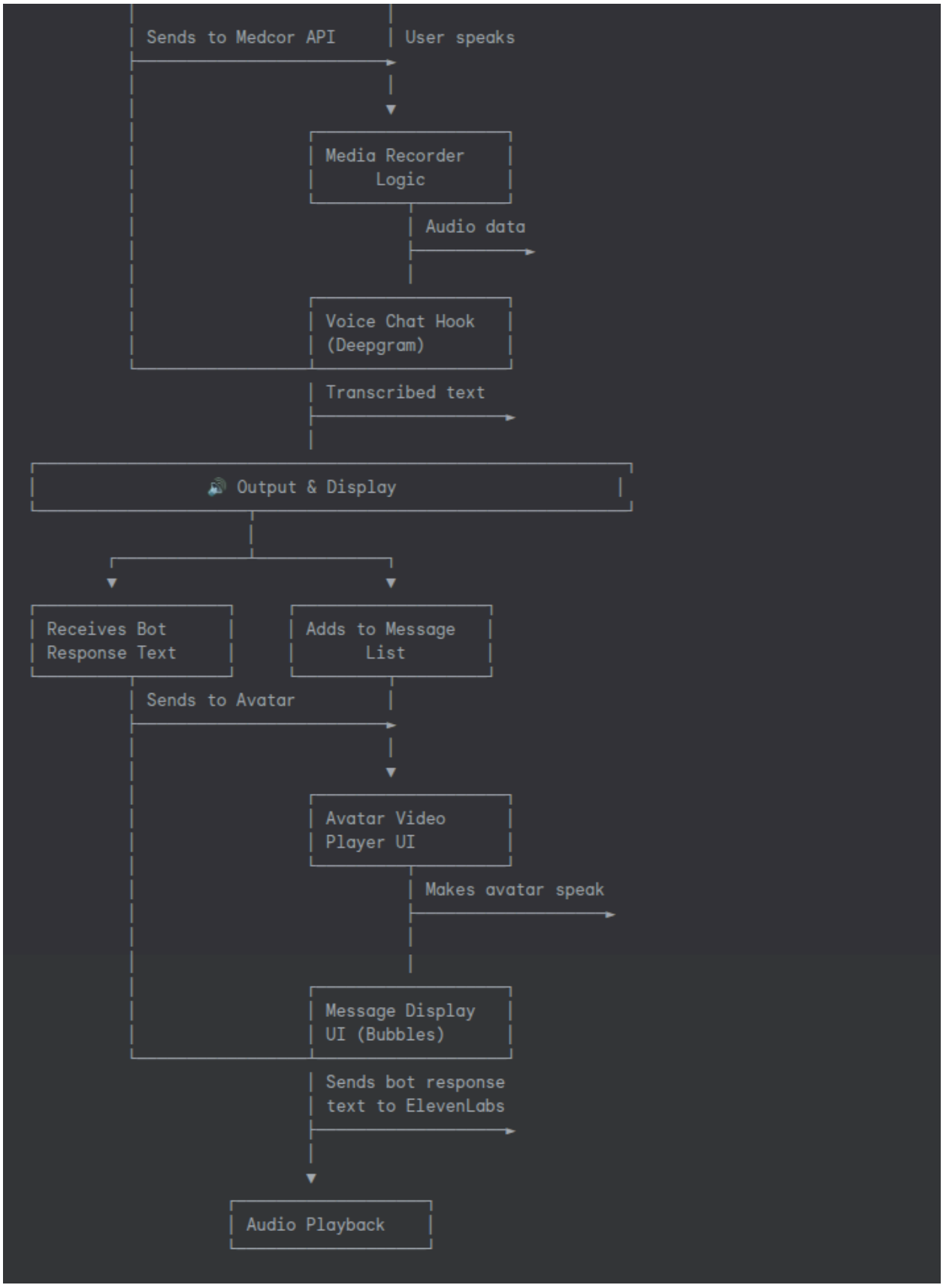
6.4. Chat Page / Interface (src/pages/chat/index.jsx)

This is where the user interacts with the chatbot.

- **Key Features**:
  - **Dual Input Modes**: Supports both text input and voice input (speech-to-text).
  - **Real-time Avatar Responses**: Displays the avatar video, which speaks the chatbot's responses.
  - **Session Timeout**: Manages user sessions and can automatically log out or end the chat after inactivity.
  - **Message History**: Shows the conversation between the user and the chatbot.

Chat Component Architecture


💬 Chat Page Core Logic (Chat/index.jsx)
Manages Message List State
Manages Recording State (Voice)
Handles Session Timeout & Mgmt.
Integrates with AuthContext
Uses `useAvatarC.` Hook
Uses `use-chatbot` Hook
Uses Voice Chat Hook (Deepgram/E.)
Input Handling
Text Input Component (User types)
Voice Input Component (Mic Button)



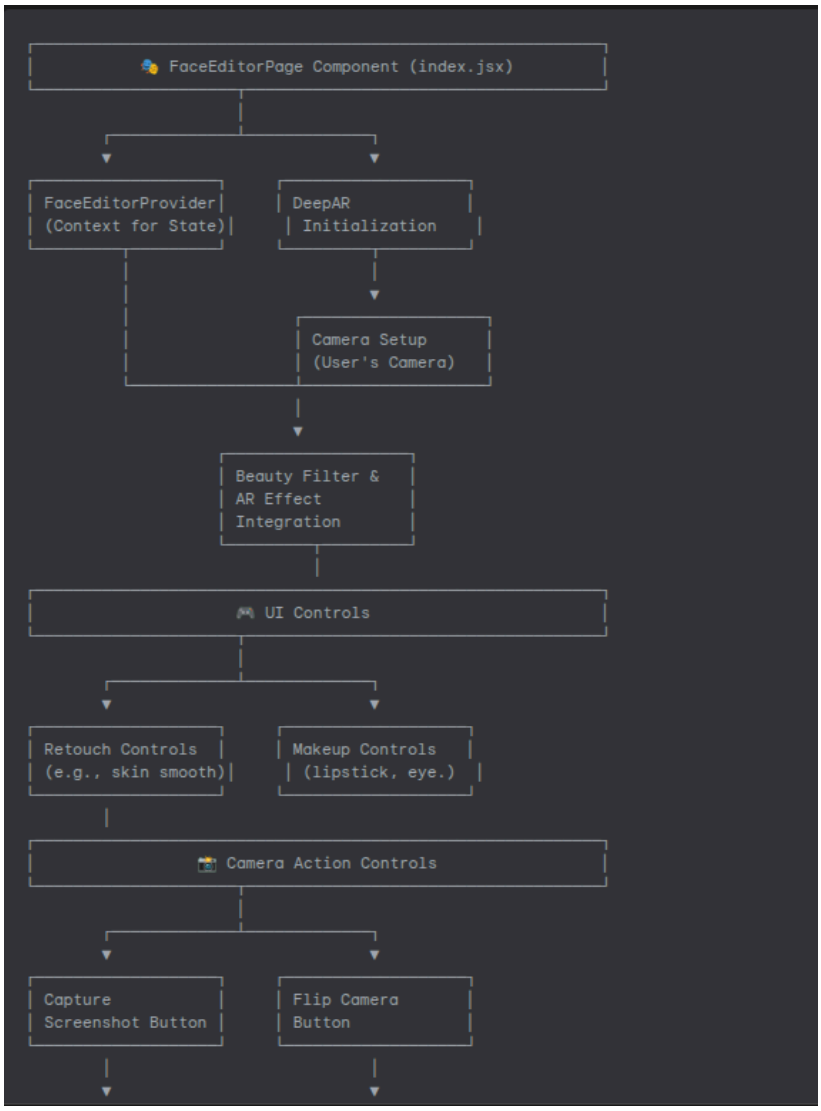
Explanation:
The ChatPage Component orchestrates interactions using various hooks and state.
User Input: Text is sent directly to the chatbot hook. Voice input is captured, sent to Deepgram (via a voice hook) for transcription, and then the text is sent to the chatbot hook.

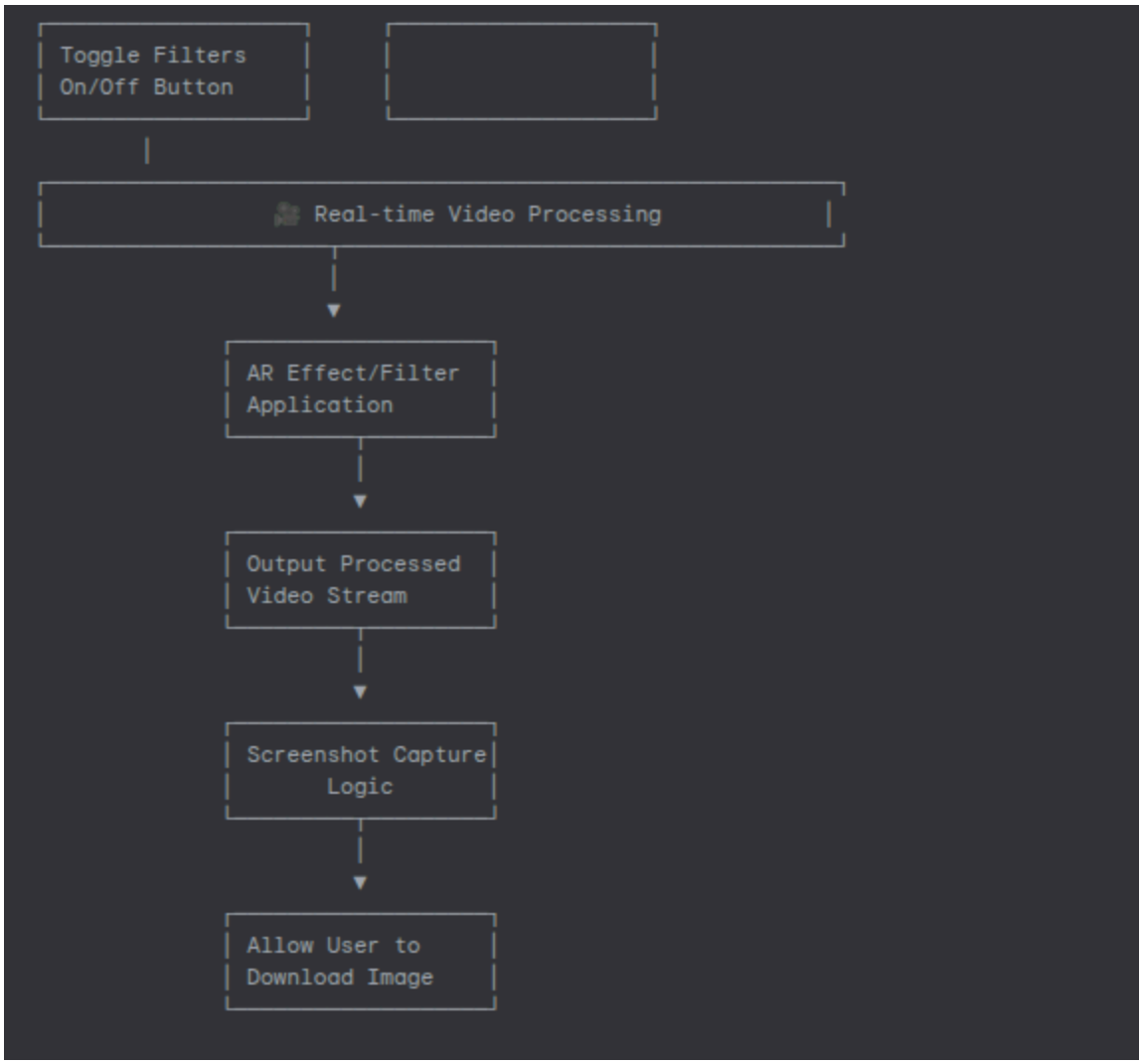
Bot Response: The chatbot hook gets a text response from the Medcor API. This text is:

- Added to the displayed message list.
- Sent to the useAvatarConnection hook to make the avatar speak.
- Potentially sent to ElevenLabs for text-to-speech audio output (if voice chat mode is active). The UI updates with new messages and the avatar's video/speech.

6.5. Face Editor Component Flow (src/pages/face-editor/index.jsx)

This component allows users to apply AR filters to their camera feed.

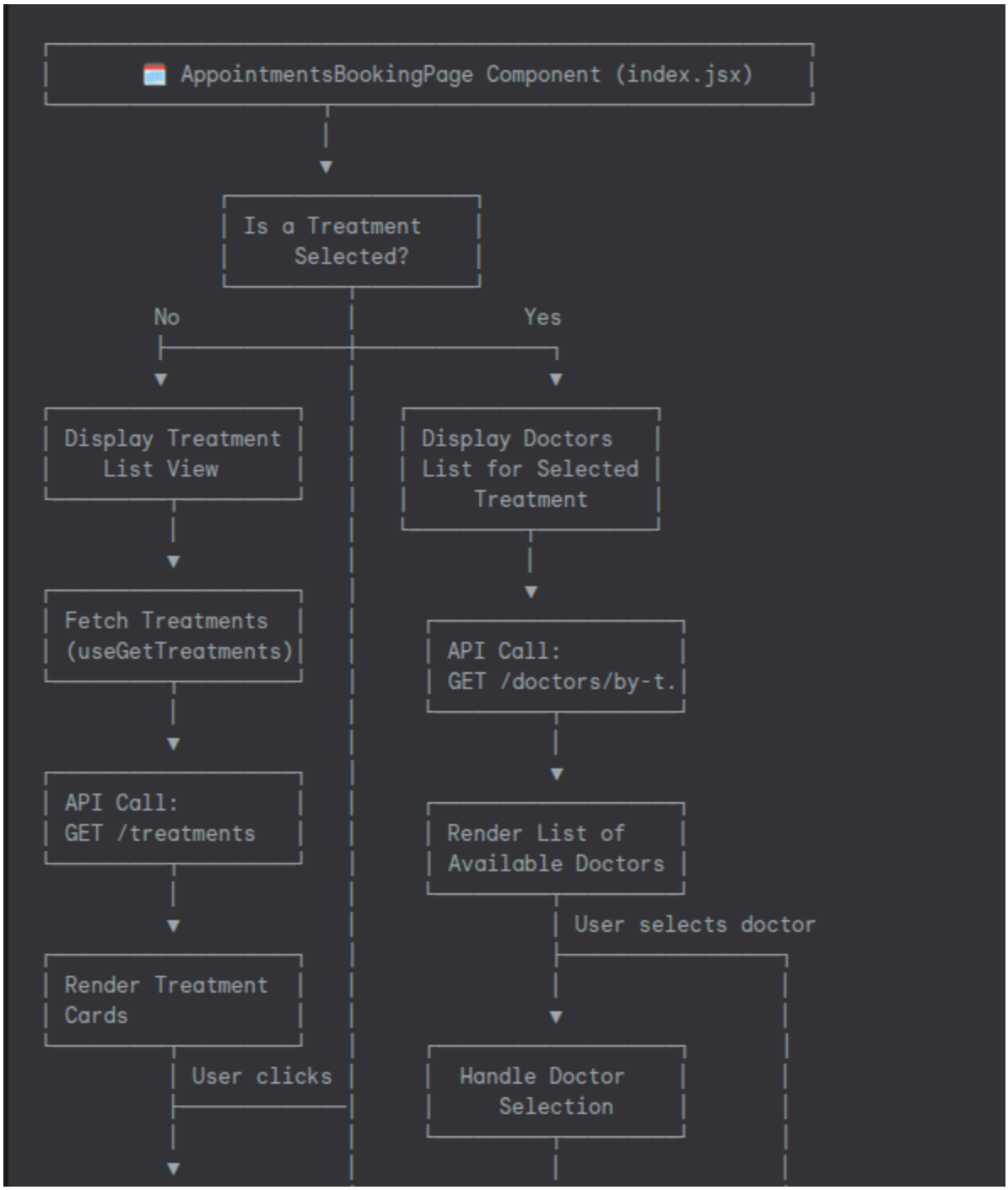


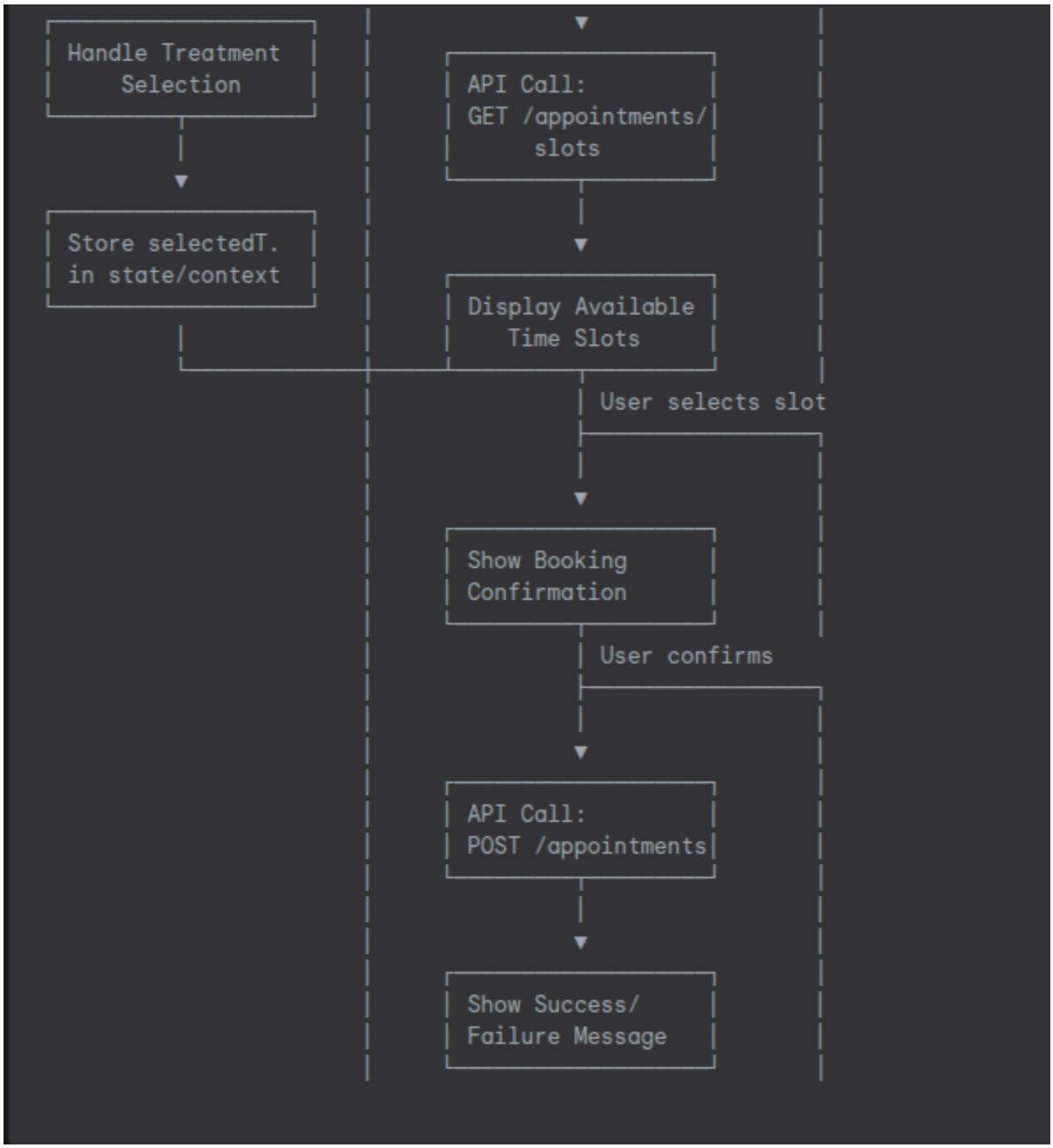


Explanation:
The FaceEditorPage initializes DeepAR using a license key and sets up the camera.
It provides UI Controls for various beauty filters, makeup effects, and camera actions (like capturing a screenshot or flipping the camera).
DeepAR processes the camera feed in real-time, applies the selected AR effects, and displays the modified video stream.
Users can capture a screenshot of the filtered video.

6.6. Appointment Booking Flow (src/pages/appointments/index.jsx)

This flow guides users through booking an appointment.


AppointmentsBookingPage Component (index.jsx)
Is a Treatment Selected?
No
Yes
Display Treatment List View
Fetch Treatments (useGetTreatments)
API Call: GET /treatments
Render Treatment Cards
User clicks
Display Doctors List for Selected Treatment
API Call: GET /doctors/by-t.
Render List of Available Doctors
User selects doctor
Handle Doctor Selection

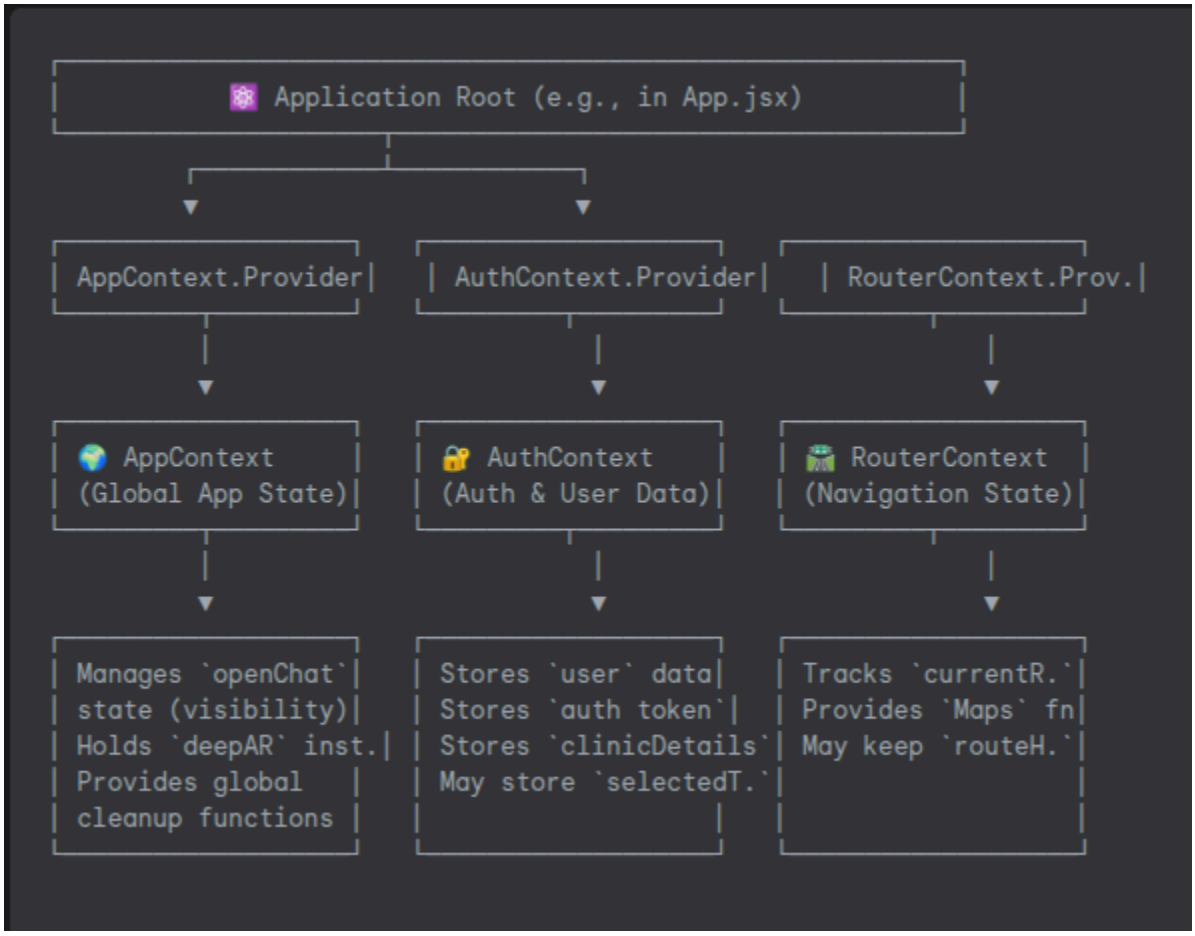


Explanation:
The user starts on the AppointmentsBookingPage. If no treatment is selected, a list of available treatments (fetched from the API) is shown.
Once the user selects a treatment, the system fetches and displays a list of doctors who offer that treatment.
After selecting a doctor, available time slots are fetched and displayed.
The user picks a slot, reviews the booking confirmation details, and confirms. An API call is then made to create the appointment.

7. State Management 🧠

State management in the widget primarily uses React Context and custom hooks.

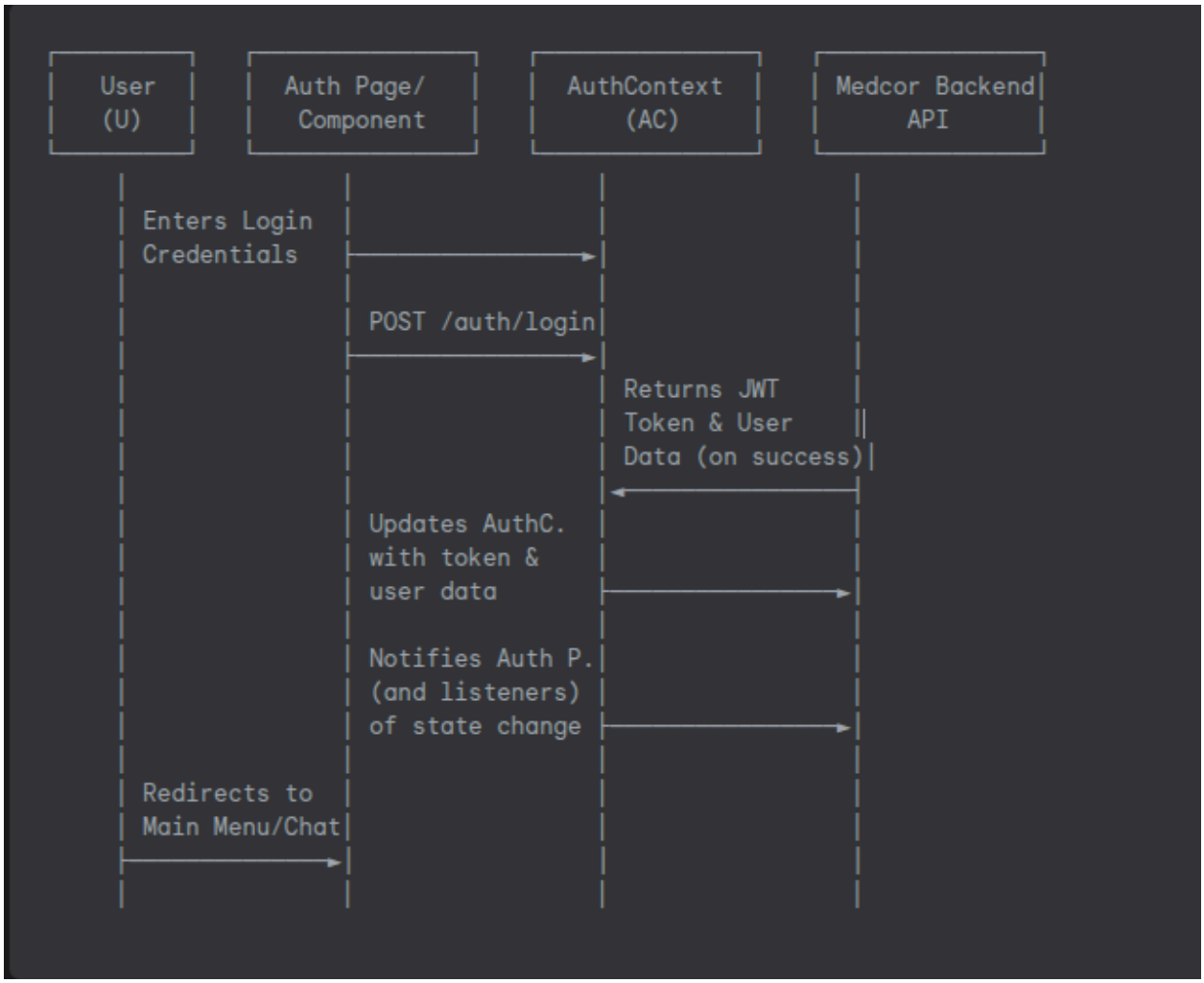
7.1. Context Architecture



Explanation:
AppContext: Handles global application states like chat visibility (openChat), the DeepAR instance, and any global cleanup logic.
AuthContext: Manages user authentication status, user data, tokens, and potentially clinic-specific details or selections relevant to the authenticated user (like a chosen treatment).
RouterContext: If the custom router leverages context, it would manage the current route, navigation functions, and history.

7.2. State Flow Patterns

Authentication Flow



Explanation: The user provides credentials on the Auth Page. These are sent to the backend API. If successful, the API returns a token and user info, which are then stored in AuthContext, making the user authenticated throughout the widget.

Avatar State Management (within useAvatarConnection)

Explanation: The avatar connection transitions through states like Disconnected, Connecting, Connected, and Speaking. Actions like initializing, speaking, or closing the chat trigger these state changes within the useAvatarConnection hook.

8. External Integrations & API 🌐

The widget relies on several external services and a backend API.

8.1. External Service Integrations

- **HeyGen (Avatar)**: For real-time avatar video streaming.

```
// Conceptual: How the hook might interact with HeyGen
const avatar = await StreamingAvatar.create({
  token: HEYGEN_TOKEN, // Your HeyGen API token
  avatarId: AVATAR_ID, // Specific avatar model ID
  quality: "high" // Optional: specify stream quality
});
await avatar.speak({
    text: message,
    // task_type: TaskType.REPEAT // Example option from dev docs
});
```

- **ElevenLabs (Text-to-Speech)**: For converting text messages into natural-sounding voice.

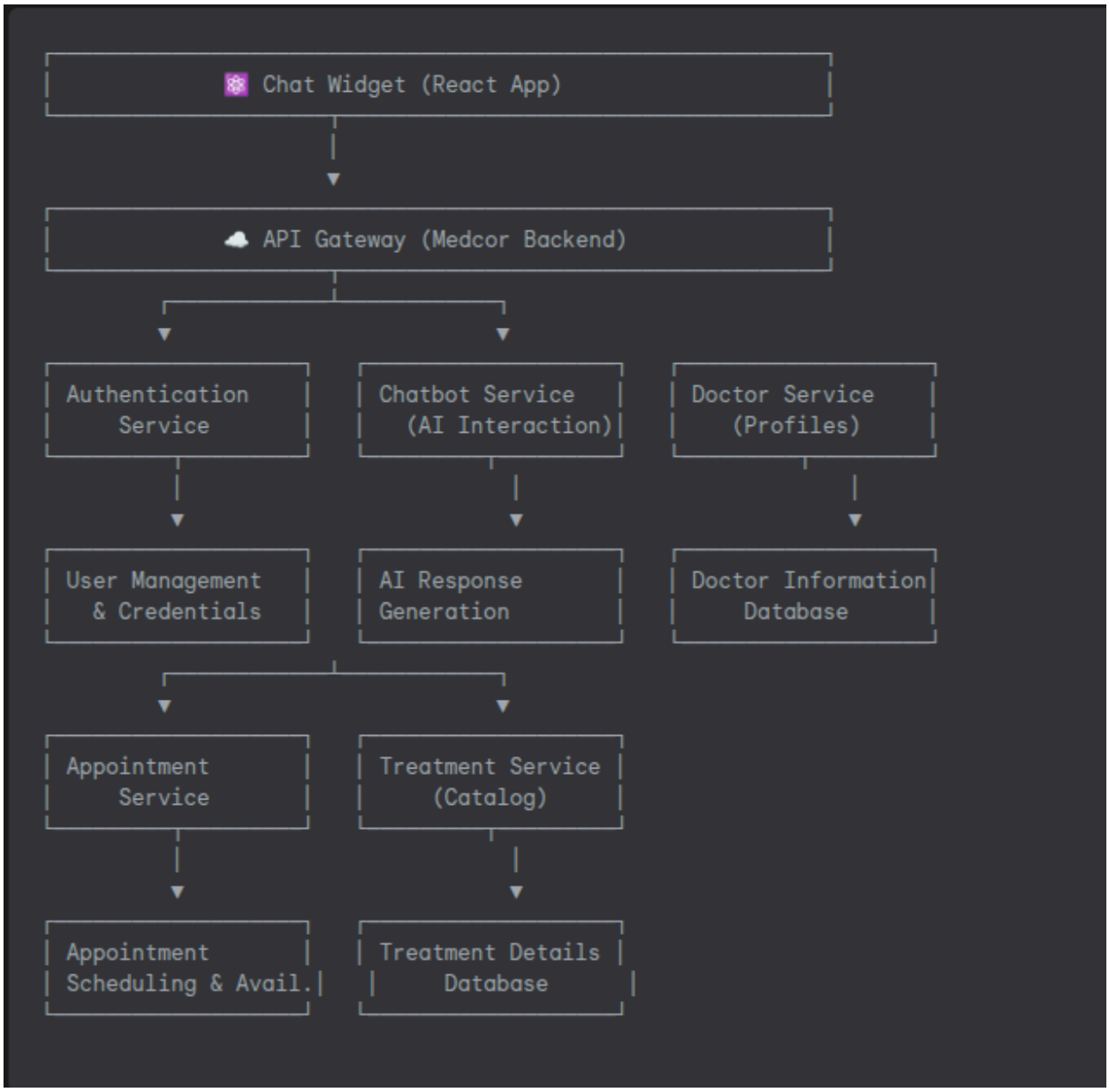
```
// Conceptual: How a voice hook might use ElevenLabs
const audioStream = await elevenlabs.textToSpeech({
  text: message,
  voice_id: VOICE_ID, // Specific voice model ID
  model_id: "eleven_multilingual_v2" // Example model
});
// stream can then be played using an <audio> element or Web Audio API
```

- **Deepgram (Speech-to-Text)**: Captures microphone input and converts spoken words into text for the chatbot. This typically involves:
  - Requesting microphone access.
  - Streaming audio data to Deepgram's API.
  - Receiving transcribed text in real-time or near real-time.
- **DeepAR (Face Editor)**: For applying AR face filters and beauty enhancements.

```
// Conceptual: How the FaceEditor component might initialize DeepAR
const deepAR = await Deepar.initialize({
  licenseKey: DEEPAR_LICENSE_KEY, // Your DeepAR license key
  previewElement: videoElement, // The HTML <video> element to display the
camera feed
  additionalOptions: { // Optional advanced settings
    cameraConfig: { facingMode: "user" }, // e.g., default to front camera
  },
```

```
});
// To apply beauty filters:
// const beauty = await Beauty.initializeBeauty(deepAR);
// beauty.setBeautify({ /* options */ });
```

8.2. Medcor Backend API Structure



Explanation: The React widget sends requests to an API Gateway, which routes them to specific microservices in the Medcor backend. These services handle tasks like user login, chatbot message processing, fetching doctor data, managing appointments, and listing treatments.

8.3. API Endpoints

Here are the primary API endpoints the widget uses:

- **Authentication**
  - POST /auth/login: For user login.
  - POST /auth/logout: For user logout.
  - GET /auth/clinic-details: To fetch details about the clinic.
- **Chatbot**

    - POST /chatbot/message: To send a user's message to the chatbot and get a response.
    - GET /chatbot/session: For managing chat sessions (e.g., checking status, timeout).
- **Appointments & Treatments**
    - GET /treatments: To fetch a list of available treatments.
    - POST /appointments: To create a new appointment.
    - GET /appointments/slots: To get available time slots for booking.
- **Doctors**
    - GET /doctors: (Likely) To list all doctors.
    - GET /doctors/{id}: To get details for a specific doctor.
    - GET /doctors/by-treatment/{treatmentId}: To find doctors offering a specific treatment.

9. Installation (for Development) 💻

To set up the project for local development:

9.1. Prerequisites

- **Node.js**: Version 18 or higher (node >= 18.0.0).
- **npm**: Version 8 or higher (npm >= 8.0.0).
- **Environment Variables**: Create a .env file in the project root (chat_widget/) with the following:

```
VITE_ENV=development
VITE_AVTAR_SESSION=10 # Example: session duration for avatar, might be minutes
VITE_FACE_EDITOR=your_deepar_license_key # Your actual DeepAR license key
```

9.2. Setup Steps

1. **Clone the repository**:
   git clone <repo-url>

   (Replace <repo-url> with the actual repository URL).
2. **Navigate to the project directory**:
   cd chat_widget

3. **Install dependencies**:
   npm install

4. **Start the development server**:
   npm run dev

   This will usually start the app on http://localhost:5173 or a similar address (Vite's default). The --host flag in the script ensures it's accessible on your local network.

10. Build & Deployment 🚀

10.1. Local Build

To create a production build of the widget locally:

npm run build

This command will compile and bundle the React application into the dist/ directory.

10.2. Vite Configuration Highlights (vite.config.js)

The vite.config.js file controls the build process. Key aspects include:

```
// vite.config.js
import { defineConfig } from 'vite';
import react from '@vitejs/plugin-react';
import { copyFileSync } from 'fs'; // For copying the loader script

export default defineConfig({
  // Base URL for assets in production.
  // This ensures that assets are loaded correctly from the CDN.
  base: "https://app.medcor.ai/static/chat_widget/dist/", //

  plugins: [
    react(),
    // Custom plugin to copy chatWidget.min.js to the dist folder during build
    {
      name: "copy-widget-loader",
      closeBundle() {
        copyFileSync("chatWidget.min.js", "dist/chatWidget.min.js");
      },
    },
  ],

  build: {
    outDir: "dist", // Output directory for the build
    rollupOptions: {
      output: {
        // Defines the output filename for the main JavaScript bundle.
        entryFileNames: "chat_widget_production.js", //
        // 'iife' (Immediately Invoked Function Expression) is suitable for
widgets
        // as it encapsulates the code and avoids polluting the global scope.
        format: "iife", //
```

```
        // When format is 'iife', dynamic imports are usually inlined to keep it
a single file.
        inlineDynamicImports: true,
      },
    },
  },
});
```

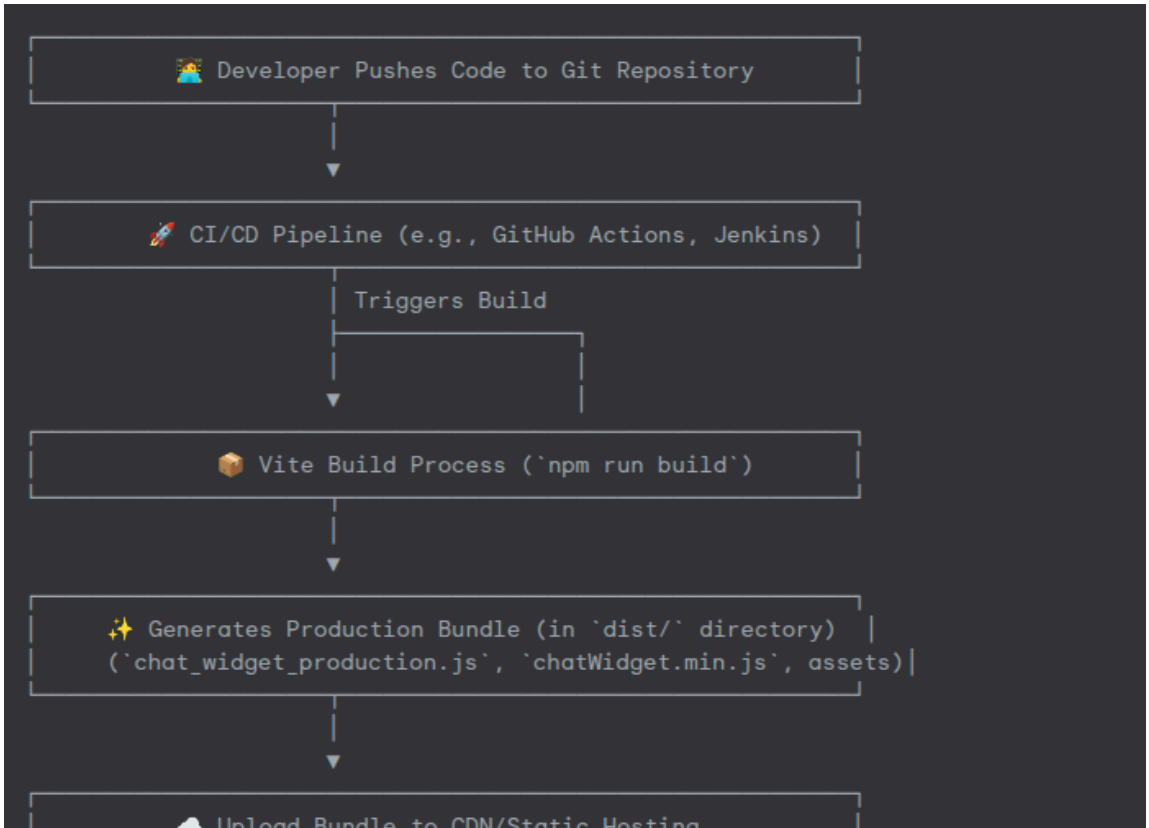
Explanation:

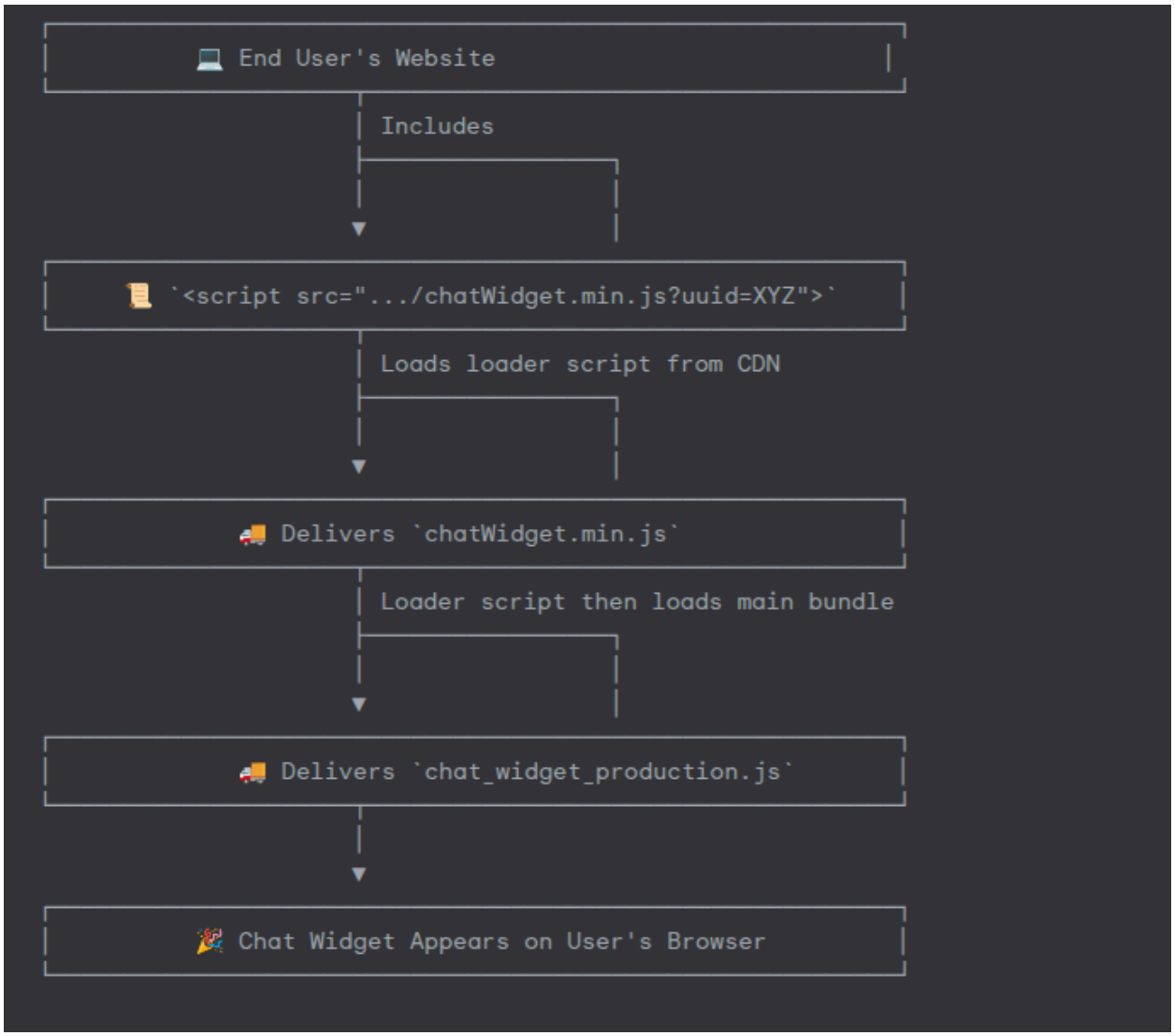
- base: Sets the public path for assets when deployed to a CDN, ensuring correct asset loading.
- plugins: Includes the React plugin and a custom plugin to copy the chatWidget.min.js loader script into the dist folder.
- build.outDir: Specifies dist as the output directory for build files.
- build.rollupOptions.output:
  - entryFileNames: Names the main production JavaScript file chat_widget_production.js.
  - format: "iife": Bundles the code as an IIFE, which is good for self-contained widgets.
  - inlineDynamicImports: true: For IIFE format, it's common to inline dynamic imports to produce a single JS file for easier embedding if code splitting isn't strictly needed for the main bundle itself.

10.3. CDN Delivery

Production assets (like chat_widget_production.js and chatWidget.min.js) are hosted on a CDN for fast delivery: https://app.medcor.ai/static/chat_widget/dist/.

10.4. Deployment Architecture





Explanation:
Code is built using Vite.
The resulting production bundle (JS files, assets) is deployed to a CDN.
The chatWidget.min.js loader script, also on the CDN, is embedded in client websites. This loader then fetches the main application bundle (chat_widget_production.js) from the CDN.

10.5. Widget Embedding Process (chatWidget.min.js)

The chatWidget.min.js script is the small loader that customers embed on their sites. Here's a breakdown of its logic:

```
// Simplified logic within chatWidget.min.js
(function () {
  // 1. Get the current script tag to access its parameters
  const scriptTag = document.currentScript;
  const urlParams = new URLSearchParams(scriptTag.src.split("?")[1]);
  const storeUUID = urlParams.get("uuid"); // Extract 'uuid'

  // 2. Environment Detection (optional, but good practice)
  //    Determines whether to load from local dev server or production CDN.
  const localhostHostnames = ["localhost", "127.0.0.1"];
  const isProductionEnv =
!localhostHostnames.includes(window.location.hostname);
  const mainBundleUrl = isProductionEnv
    ? "https://app.medcor.ai/static/chat_widget/dist/chat_widget_production.js"
    : "http://127.0.0.1:5173/src/main.jsx"; // Vite dev server path (or local
dist path)
     // Note: The dev docs mention loading `chat_widget_production.js` even for
local,
     // so ensure this matches the actual dev setup.
     // "http://127.0.0.1:5500/chat_widget/dist/chat_widget_production.js"

  // 3. Store Configuration (e.g., UUID) in localStorage
  if (storeUUID) {
    localStorage.setItem("chat_widget_uuid", storeUUID); //
  }

  // 4. Create a container div for the React application
  const containerDiv = document.createElement("div");
  containerDiv.id = "medcor-chat-widget"; // A unique ID for the widget's root
  document.body.appendChild(containerDiv);

  // 5. Load the main React application script
  const mainAppScript = document.createElement("script");
  mainAppScript.src = mainBundleUrl;
  // mainAppScript.type = "module"; // If loading Vite's dev entry point
```

```
directly
  document.head.appendChild(mainAppScript);
})();
```

Explanation: This IIFE (Immediately Invoked Function Expression) runs as soon as it's loaded:
It finds itself in the DOM to read the uuid parameter from its own src attribute.
It might check if the environment is production or development to load the main JS bundle from the correct URL (CDN or local dev server).
The uuid is stored in localStorage for the React app to access.
A div with a specific ID (e.g., medcor-chat-widget) is created and appended to the document.body. This div will serve as the mount point for the React application.
A new <script> tag is created to load the main application bundle (e.g., chat_widget_production.js), which will then initialize React and render the widget into the prepared div.

11. Development Scripts 📜

These npm scripts are available for development:

```
"scripts": {
  "dev": "vite --host",          // Starts the development server, accessible on
the local network
  "build": "vite build",         // Creates a production build in the 'dist'
folder
  "watch": "vite build --watch",// Builds in watch mode, recompiling on file
changes (useful for linking)
  "lint": "eslint .",            // Runs ESLint to check for code style and
errors
  "preview": "vite preview"      // Serves the 'dist' folder locally to preview
the production build
}
```

12. Performance Considerations ⚡

Performance is crucial for a widget that runs on other websites.

12.1. General Performance Tips

- **Keep Loader Small**: The initial chatWidget.min.js loader script should be under 5KB to ensure it loads and executes quickly.
- **Lazy Loading**: Use lazy loading for heavy components or features that are not immediately needed when the widget first appears.
- **Cleanup Resources**: Properly clean up avatar connections, DeepAR instances, event listeners, and media streams when the chat widget is closed or unmounted to prevent memory leaks.

- **Minify & Compress**: Ensure all bundles are minified and compressed (e.g., Gzip, Brotli) for CDN delivery to reduce file sizes.

12.2. Bundle Optimization Strategy

Explanation: The build process employs several techniques like tree shaking, (potential) code splitting, minification, and relies on server/CDN compression to optimize the bundle size and loading speed.

12.3. Loading Strategy

- **Critical Path**: Load the tiny chatWidget.min.js loader first. It's essential and should be very small.
- **Lazy Loading**: The main application bundle (chat_widget_production.js) can be considered the next step, but internal parts of this bundle can also be lazy-loaded (e.g., specific pages or heavy components like the Face Editor).
- **Progressive Enhancement**: Core chat functionality loads first. More complex features (like face editor) can load on demand when the user accesses them.
- **Caching**: Implement aggressive caching strategies for static assets on the CDN to ensure returning users load the widget quickly.

12.4. Memory Management

- **Avatar Service**: Ensure avatar.cleanup() is called to disconnect from HeyGen and release resources.
- **DeepAR**: Dispose of the DeepAR instance when the face editor is closed or the widget is unmounted to free up camera and processing resources.
- **Event Listeners**: Remove any manually added event listeners to prevent memory leaks.
- **Media Streams**: Terminate microphone and camera streams when they are no longer needed.

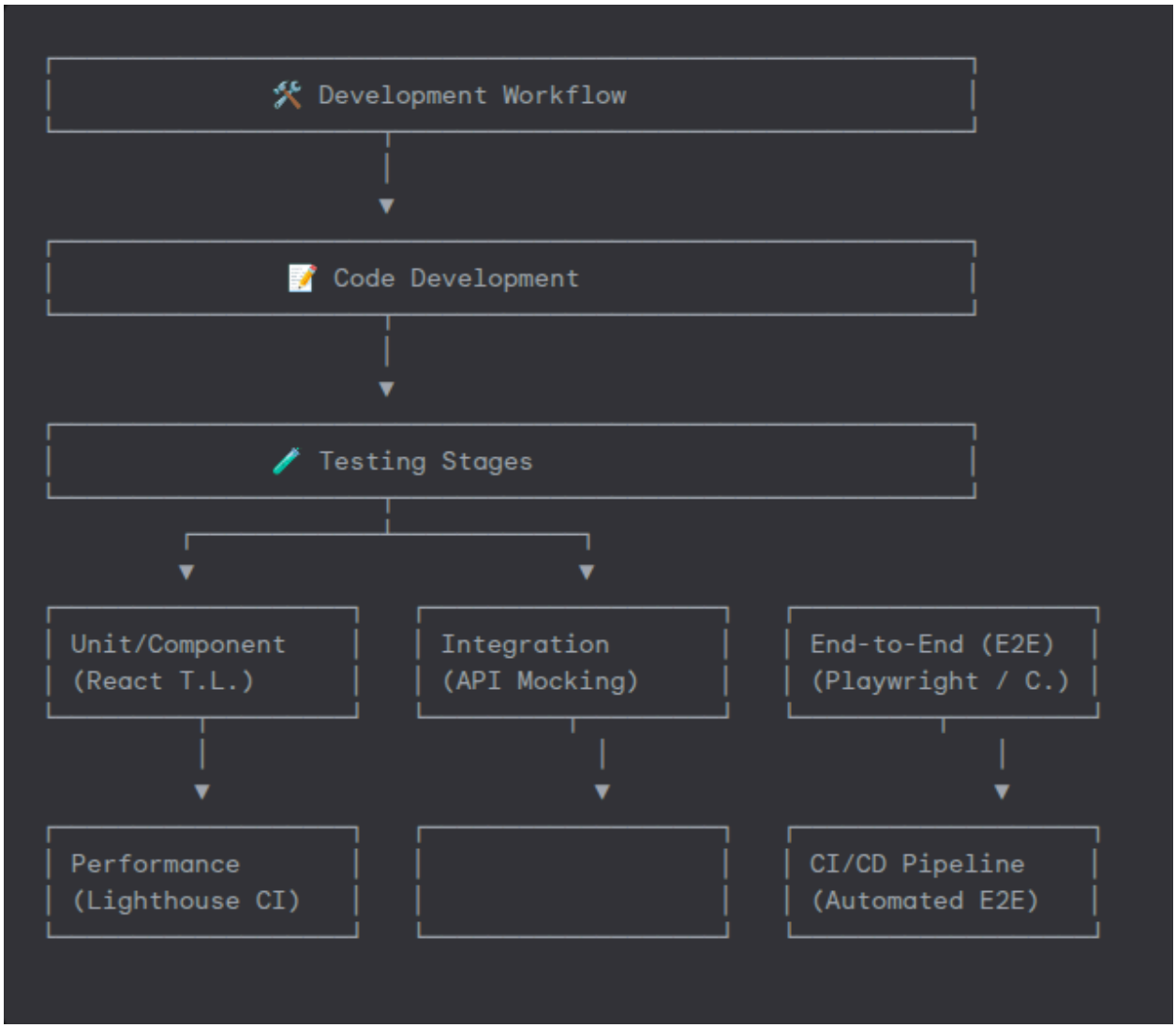
13. Testing 🧪

A robust testing strategy ensures the widget is reliable.

13.1. Testing Levels

- **Component Tests**: Use Jest and React Testing Library to test individual React components in isolation.
- **Integration Tests**: Test interactions between components or between the widget and mocked API services. API mocking is key here.
- **End-to-End (E2E) Tests**: Use tools like Cypress or Playwright to simulate real user scenarios across the entire application flow.
- **Performance Tests**: Use Lighthouse CI to regularly check performance metrics and regressions.

13.2. Testing Integration Overview



Explanation: Different testing tools are used at various stages. Unit and integration tests can run frequently during development. E2E and performance tests are often integrated into a CI/CD pipeline to catch issues before deployment.

13.3. Code Quality Tools

- **ESLint**: Enforces JavaScript/TypeScript coding standards and catches potential errors.
- **Prettier**: Automatically formats code for consistent style.
- **Husky**: Can be used to set up Git hooks (e.g., run linters/tests before committing).
- **TypeScript**: (If used) Provides static type checking to catch errors early.

14. License 📄