1. Code Quality and Structure

1.1 Function Documentation & Comments

Explanation: All functions have headers describing purpose, parameters, outputs, and side effects. Inline comments explain why a particular logic exists (not what it does). Some files also include file headers.

Examples:

- clientController.js (lines 3-30):
    - buyTicket function header documents input (eventId), output (JSON response), and side effect (decrements ticket count).
- App.js (lines 19-24):
    - buyTicket function header describes parameters, return, and state updates.

1.2 Variable Naming & Code Readability

Descriptive, meaningful names are used for variables and functions. Consistent camelCase is used for functions and variables, PascalCase for components. Lines kept reasonably short.

Examples:

- purchaseTicket in clientModel.js (lines 30-45) describes its purpose.

1.3 Modularization

Code is split into small, reusable functions, separated by layer: models for DB, controllers for business logic, routes for API endpoints. No duplicated code; helpers used where needed.

Example:

- clientController.js handles business logic, clientModel.js handles DB.
- clientRoutes.js delegates to controller functions (lines 10–14).

1. 4 Error Handling

Inputs validated, API responses use proper HTTP status codes, clear error messages returned, and errors handled gracefully.

Example:

- clientController.js (lines 37-43): returns 400 if tickets sold out.
- adminController.js (lines 29-34): returns 400 if event info incomplete and 500 if DB insert fails

1.5 Consistent Formatting
Code uses 2-space indentation, logical blocks separated with blank lines, grouped statements. ESLint used in frontend for consistent style.

Example:
- frontend/src/App.js (lines 1–60) follows consistent formatting for React functional component.
- clientModel.js (lines 3–45) uses consistent spacing and line breaks between DB operations.

1.6 Input and Output Handling
APIs expect JSON with defined properties; responses include JSON objects or status codes. All DB calls handled asynchronously with callbacks.

Example:
- clientController.js (lines 10–25): buyTicket receives eventId param and responds with JSON object indicating success/failure.
- App.js (lines 15–50): fetches events from API and updates state asynchronously.

2. Accessibility

The frontend for TigerTix (app.js and voiceInterface.js) follows the rubric's best accessibility practices.

Accessibility features have been added in the form of voice interaction. The user can activate the “speech to text” to communicate with the LLM model through voice input. The microphone will capture the user’s input and utilize the Web Speech API to convert the audio into text. The text is then sent to the LLM. The LLM’s response is vocalized using the Speech Synthesis API.

3. Architecture Design and Explanation

Frontend to Client Service:
1. Fetches list of events from Client Service using GET /api/events.
2. When a ticket is bought, the front end sends a POST request to the Client Service

Client Service to shared DB:
1. The Client Service reads from the shared DB to get the current events, and when a ticket is purchased it will decrement the ticket count in the database and reflect that sale.

Admin Services to shared DB:
1. The admin Service manages the events table in the DB, adds new events, updates info and deletes events
2. Changes made by the Admin Service are reflected in the shared DB.

Input to Voice Interface:

1. The microphone on the user's device takes in the user's audio
2. The Web Speech API converts the user's audio to text

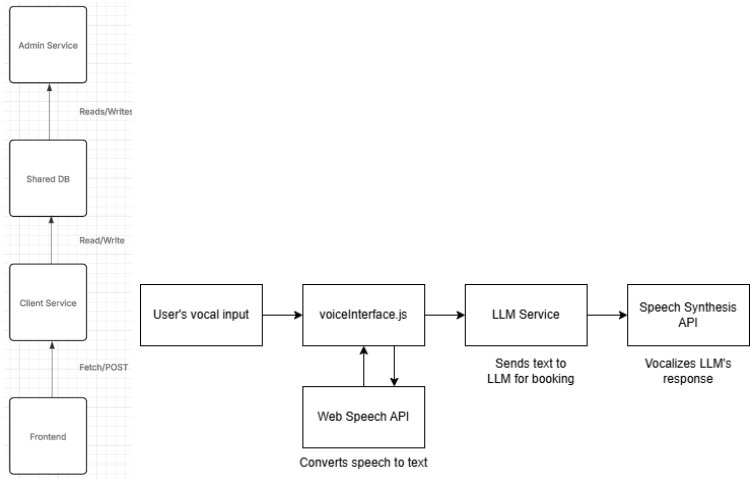
Voice Interface to LLM:

1. The text from the voice interface is sent to the LLM
2. The LLM parses the information

Output to Audio:

1. The text output from the LLM is sent to the voice interface
2. The Speech Synthesis API converts the output to audio and plays the response

Diagram:



4. Testing

Tests for the TigerTix program were conducted on Admin and Client microservices, LLM-driven booking, voice-enabled interface, accessibility features, database transactions and concurrency, event creation and retrieval, and ticket booking and confirmation.

4.1 Automated Testing

Test Name	**Input**	**Output**	**Description**
Accessibility	Simulates the user pressing the "tab" button	The field that is selected on the page changes	Ensures keys and simulated keys can be used to navigate the application
Accessibility Label	Waits for the mic button, then	Checks that aria labels exist for the	Ensures aria labels work for accessibility

	simulates a request, then waits for the confirm button	mic and confirm buttons	
Create Event	Creates an event for “CPSC Expo” on “November 5, 2025”	An event for a CPSC Expo has been successfully created	Checks that an event can be created
Get Event	Creates an event for “CPSC Expo” on “November 5, 2025” and then tries to retrieve the event	An event for a CPSC Expo is successfully created and retrieved	Checks that an event can be retrieved from the database
Get and Create Event	Creates an event for “Concert” on “November 5, 2026” and then tries to retrieve the event	An event for a concert is successfully created and retrieved	Checks that an event can be retrieved from the database with a date further in the future
Test LLM Booking	Tests booking an event for a festival using the llm	2 tickets for a festival is successfully booked	Checks that the full process of booking for LLMs is sucessful
Voice Chat	Tests booking an event using simulated voice	Booking sucessful	Checks that a booking can be successfully completed using the voice chat function

4.2 Manual Testing

For manual testing, three types of tests were executed: Booking via natural language (text and voice), accessibility navigation with keyboard and screen reader, and concurrent booking attempts to verify database consistency.

Booking via Natural Language:

To conduct tests on booking via natural language, we tested a variety of potential inputs into the chatbot and recorded our input and the output. All tests passed without issue and as expected.

Test	**Input**	**Output**	**Expected Output**
1 Not Real Event	Book 2 tickets for jazz night Yes	I understood: Book 2 ticket(s) for "Jazz Night". Confirm?	I understood: Book 2 ticket(s) for "Jazz Night". Confirm?

		Could not find an event matching "Jazz Night".	Could not find an event matching "Jazz Night".
2 Real Event	Book 2 tickets for Concert Confirm	You: Book 2 tickets to Concert Bot: I understood: Book 2 ticket(s) for "Concert". Confirm? You: Confirm booking 2 for Concert Bot: Booking successful (qty: 2) for "Concert"	You: Book 2 tickets to Concert Bot: I understood: Book 2 ticket(s) for "Concert". Confirm? You: Confirm booking 2 for Concert Bot: Booking successful (qty: 2) for "Concert"
3 Random text	Hello Robot Assistant	Chat Assistant You: Hello Robot Assistant Bot: Missing fields in parsed JSON	Chat Assistant You: Hello Robot Assistant Bot: Missing fields in parsed JSON
4 Infinite Tickets	Book 1000000 tickets to Concert	You: Book 1000000 tickets to Concert Bot: I understood: Book 1000000 ticket(s) for "Concert". Confirm? You: Confirm You: Confirm booking 1000000 for Concert Bot: Not enough tickets available	You: Book 1000000 tickets to Concert Bot: I understood: Book 1000000 ticket(s) for "Concert". Confirm? You: Confirm You: Confirm booking 1000000 for Concert Bot: Not enough tickets available

Accessibility Navigation:

To test the accessibility capabilities, we conducted the same tests as those for the natural language tests. These tests were conducted using the microphone component and a screen reader. All tests passed without issue and as expected.

Concurrent Booking:

To test the ability to book tickets simultaneously, we each attempted to book tickets at the same time using the program. We encountered no errors during this process, and our tickets were booked successfully.