The Complete Hackathon Playbook: MERN Stack, AI Integration, and Collaborative Git Workflow

Part I: Foundation and Environment Setup: Building Your Hackathon Launchpad

A successful hackathon performance is built upon a solid foundation. Time spent wrestling with setup issues is time lost on innovation and implementation. This first part is dedicated to meticulously constructing a flawless development environment. By following these steps, a developer can ensure their machine is perfectly configured and optimized for the MERN stack, eliminating potential friction and creating a launchpad for rapid development.

Section 1.1: The Hackathon Toolkit: Essential Installations

Before a single line of code is written, the necessary software toolkit must be installed and verified. This ensures a consistent and functional environment across the team.

Node.js & npm

Node.js is the JavaScript runtime environment that allows JavaScript to be executed on the server, forming the "N" in MERN. It is the bedrock upon which the Express.js backend is built.[1] Bundled with the Node.js installation is npm (Node Package Manager), an essential tool for managing project dependencies—the external libraries and frameworks the application will use.[1]

- **Installation:** Navigate to the official Node.js website (nodejs.org) and download the Long-Term Support (LTS) version recommended for most users. The installer is straightforward for Windows, macOS, and Linux.[3]
- **Verification:** After installation, open a new terminal or command prompt and run the following commands to verify that both Node.js and npm are correctly installed and available in the system's PATH:
  Bash

```
node -v
npm -v
```

These commands should return the respective version numbers of Node.js and npm.

MongoDB

MongoDB is the NoSQL, document-based database for the MERN stack, representing the "M".[2] Its flexible, JSON-like document structure makes it particularly well-suited for JavaScript-based applications, as data flows seamlessly between the database and the application logic.[1] For a hackathon, a cloud-based setup is highly recommended for ease of collaboration.

- **Local Setup (Optional):** For developers who may need to work offline, MongoDB Community Server can be installed locally. However, this can add complexity when sharing the database with team members.
- **Cloud Setup (Recommended for Hackathons):** MongoDB Atlas provides a free-tier, fully managed cloud database that is ideal for collaborative projects. A single, shared database connection string can be used by all team members, simplifying setup and ensuring everyone is working with the same data.[2]
    1. Visit the MongoDB Atlas website and sign up for a free account.
    2. Create a new project and build a new cluster. Select the "M0" free tier, which is more than sufficient for a hackathon.
    3. Choose a cloud provider and region (select one that is geographically close to your team).
    4. While the cluster is being provisioned (this may take a few minutes), configure database access. Create a database user with a secure username and password.
    5. Configure network access. For a hackathon, the simplest approach is to allow access from anywhere by adding the IP address 0.0.0.0/0. While not recommended for production, this removes potential network-related roadblocks during the event.
    6. Once the cluster is ready, click "Connect," select "Connect your application," and copy the provided connection string (URI). Save this string, as it will be used in the backend's environment configuration.

Visual Studio Code (VS Code)

A powerful and extensible code editor is crucial for productivity. Visual Studio Code has become the industry standard for web development due to its vast ecosystem of extensions, integrated terminal, and debugging capabilities.[3] Download and install it from the official website.

Git

Git is a distributed version control system that is non-negotiable for any team-based software development project, especially in a fast-paced hackathon environment.[4] It allows the team to track changes, collaborate on different features simultaneously without overwriting each other's work, and maintain a complete history of the project. Installation guides are available on the official Git website (
git-scm.com).[4]

Section 1.2: Supercharging VS Code for MERN Development

A default VS Code installation is powerful, but its true potential is unlocked through extensions. The following extensions are tailored to streamline MERN stack development, transforming the editor into an intelligent assistant that helps write cleaner code faster. Installing these is a high-leverage activity that pays dividends throughout the hackathon.

Extension Name	Publisher	Purpose in Hackathon	Key Feature
ES7+ React/Redux/React-Native Snippets	dsznajder	Dramatically reduces the time spent writing boilerplate for React components.	Type rafce and press Enter to generate a complete React Arrow Function Component with export.[6]
Prettier - Code Formatter	Prettier	Enforces a consistent code style across the entire team, eliminating debates over formatting and cleaning up code on save.	Automatically formats JavaScript, JSX, CSS, and more, ensuring clean and readable commits.[6]
ESLint	Microsoft	Catches common JavaScript errors and enforces coding best practices in real-time, preventing bugs before they happen.	Integrates with a project's ESLint configuration to highlight potential issues directly in the editor.[6]
MongoDB for VS Code	MongoDB	Allows direct interaction with the MongoDB Atlas or local database from within	Browse collections, view and edit documents, and run queries without leaving

		the editor, avoiding context switching.	VS Code.[6]
Thunder Client	Ranga Vadhineni	A lightweight, integrated alternative to Postman for testing the backend's REST API endpoints.	Make GET, POST, etc., requests to the Express server directly inside VS Code to quickly test API functionality.[6]
Path Intellisense	Christian Kohler	Autocompletes file paths in import statements, reducing typos and speeding up module imports.	Prevents frustrating "module not found" errors caused by incorrect relative paths.[6]
Auto Rename Tag	Jun Han	Automatically renames the closing HTML/JSX tag when the opening tag is changed, saving time and preventing errors.	Essential for refactoring React components and ensuring valid JSX syntax.[6]
GitLens — Git Supercharged	GitKraken	Enhances VS Code's built-in Git capabilities, providing deep insights into the repository's history.	See inline blame annotations (who wrote a line of code and when), easily compare branches, and explore commit history.[6]

Section 1.3: Architecting the Project: A Monorepo for Speed

The project's directory structure is a critical architectural decision. For a hackathon, a monorepo-style structure—a single repository containing both the frontend and backend projects—is the most efficient approach. This simplifies version control, setup, and the mental model for communication between the two parts of the application, avoiding the complexities of managing multiple repositories and intricate proxy configurations that can consume valuable time.[3]

Step 1: Create the Root Directory

Begin by creating a single parent folder for the entire project.

Bash

```bash
mkdir mastercard-hackathon
cd mastercard-hackathon
```

Step 2: Initialize Git Immediately

Before any other files are created, initialize a Git repository in the root directory. This ensures that every subsequent change is tracked from the very beginning.[4]

Bash

```bash
git init
```

Step 3: Create the .gitignore File

A .gitignore file is crucial for keeping the repository clean and secure. It tells Git which files and folders to ignore. Create this file in the root directory and add entries for common artifacts that should not be committed to version control, especially sensitive files like .env and bulky folders like node_modules.[4]
Create a file named .gitignore and add the following content:

```
# Dependencies
/node_modules
/frontend/node_modules
/backend/node_modules

# Build artifacts
/frontend/build
/dist

# Logs
npm-debug.log*
yarn-debug.log*
yarn-error.log*
```

```
# Environment variables
.env
.env.local
.env.development.local
.env.test.local
.env.production.local
/frontend/.env
/backend/.env
```

Step 4: Scaffold the Backend

Navigate into the root directory and create the backend folder. Initialize a Node.js project and install the core dependencies.[3]

Bash

```
mkdir backend
cd backend
npm init -y
npm install express mongoose cors dotenv
```

- **express**: The web server framework for Node.js.[5]
- **mongoose**: An Object Data Modeling (ODM) library for MongoDB, providing a straightforward, schema-based solution to model application data.[3]
- **cors**: A Node.js package for providing a Connect/Express middleware that can be used to enable Cross-Origin Resource Sharing (CORS) with various options.
- **dotenv**: A zero-dependency module that loads environment variables from a .env file into process.env.[3]

Step 5: Scaffold the Frontend

Return to the root directory and use create-react-app to scaffold the frontend application. Then, install axios, the primary library for making HTTP requests to the backend.[3]

Bash

```
cd..
npx create-react-app frontend
cd frontend
npm install axios
```

At the end of this process, the project structure should look like this:

```
mastercard-hackathon/
├──.git/
├──.gitignore
├── backend/
│   ├── node_modules/
│   └── package.json
└── frontend/
   ├── node_modules/
   ├── public/
   ├── src/
   └── package.json
```

This clean, organized structure provides a robust starting point for the hackathon, with both frontend and backend environments ready for development.

Part II: Bridging the Gap: Frontend-Backend Integration

With the development environment fully configured, the next critical task is to establish a communication channel between the React frontend and the Express backend. This "digital handshake" is the backbone of any full-stack application. This part provides a step-by-step guide to building a simple yet robust API, connecting to it from React, and proactively solving the most common roadblock in MERN development: Cross-Origin Resource Sharing (CORS).

Section 2.1: Building the Express API Backbone

The backend server acts as the application's engine, handling business logic, database interactions, and requests from the client. The first step is to create a minimal, working Express server that can listen for and respond to HTTP requests.
In the backend directory, create a new file named server.js. This file will be the entry point for the backend application.

JavaScript

```
// backend/server.js

// 1. Import Dependencies
const express = require('express');
const mongoose = require('mongoose');
const cors = require('cors');
require('dotenv').config();

// 2. Initialize Express App
const app = express();
const PORT = process.env.PORT |

| 5000;

// 3. Middleware Setup
app.use(cors()); // We will configure this properly later
app.use(express.json()); // Middleware to parse JSON bodies

// 4. Database Connection
const MONGODB_URI = process.env.MONGODB_URI;
mongoose.connect(MONGODB_URI)
.then(() => console.log('MongoDB connected successfully.'))
.catch(err => console.error('MongoDB connection error:', err));

// 5. API Endpoints (Routes)
// A simple "health check" endpoint to verify the server is running
app.get('/api/health', (req, res) => {
  res.status(200).json({ status: 'ok', message: 'Server is healthy' });
});

// 6. Start the Server
app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

Code Explanation:

1. **Dependencies:** We import the necessary packages: express for the server, mongoose for the database connection, cors to handle cross-origin requests, and dotenv to load environment variables.[3]

2. **Initialization:** An instance of the Express application is created, and a port is defined, preferably from an environment variable for flexibility.[13]
3. **Middleware:** express.json() is crucial middleware that allows the server to automatically parse incoming JSON payloads in request bodies. cors() is added provisionally and will be configured in a later section.[16]
4. **Database Connection:** The server uses Mongoose to connect to the MongoDB Atlas cluster using the connection string stored in the .env file. Success and error messages are logged to the console for easy debugging.[2]
5. **Health Check Endpoint:** The /api/health route is a simple GET endpoint. Its sole purpose is to provide a verifiable sign that the server is running and reachable. This is an invaluable debugging tool; if this endpoint works, the fundamental server setup is correct. This helps isolate problems, as a failure in a more complex route would then point to issues within that route's specific logic or database query, not the server's basic connectivity.[2]
6. **Server Start:** app.listen() starts the server and makes it listen for incoming connections on the specified port.

To run this server, create a .env file in the backend directory with your MongoDB Atlas connection string:

```
# backend/.env
MONGODB_URI="mongodb+srv://<username>:<password>@<cluster-url>/<database-name>?retryWrites=true&w=majority"
PORT=5000
```

Then, from the backend directory, start the server:

Bash

```
node server.js
```

Section 2.2: The First Handshake: Connecting React to Node.js

With the backend server running, the next step is to make the React frontend fetch data from it. This is accomplished using axios to make an HTTP request from within a React component. Modify the frontend/src/App.js file to perform this "first handshake."

JavaScript

```
// frontend/src/App.js

import React, { useState, useEffect } from 'react';
import axios from 'axios';
import './App.css';

function App() {
  const = useState('');
  const [error, setError] = useState('');

  useEffect(() => {
    // The URL of our backend health check endpoint
    const backendUrl = 'http://localhost:5000/api/health';

    axios.get(backendUrl)
    .then(response => {
        // On success, update the state with the server's message
        setServerMessage(response.data.message);
        setError(''); // Clear any previous errors
      })
    .catch(error => {
        // On failure, log the error and update the error state
        console.error('There was an error fetching data from the backend!', error);
        if (error.response) {
          // The request was made and the server responded with a status code
          // that falls out of the range of 2xx
          setError(`Server responded with status: ${error.response.status}`);
        } else if (error.request) {
          // The request was made but no response was received
          // This often means the server is not running or is unreachable
          setError('Could not connect to the server. Is it running?');
        } else {
          // Something happened in setting up the request that triggered an Error
          setError(`Error: ${error.message}`);
        }
        setServerMessage(''); // Clear any previous success message
      });
  },); // The empty dependency array ensures this effect runs only once on component mount

  return (
    <div className="App">
```

```
      <header className="App-header">
        <h1>MERN Stack Hackathon</h1>
        <h2>Connecting Frontend to Backend</h2>
        <p>
          <strong>Status of Backend Connection:</strong>
        </p>
        {serverMessage && <p style={{ color: 'lightgreen' }}>{serverMessage}</p>}
        {error && <p style={{ color: 'salmon' }}>{error}</p>}
      </header>
    </div>
  );
}

export default App;
```

Code Explanation:

- **React Hooks:** The component uses useState to manage two pieces of state: serverMessage for the successful response and error for any connection issues. The useEffect hook is used to perform the side effect of fetching data. The empty dependency array `` ensures this effect runs only once, right after the component first renders, preventing an infinite loop of API calls.[3]
- **Axios Request:** axios.get() sends an HTTP GET request to the backend's /api/health endpoint.
- **Response Handling:** The .then() block executes if the request is successful (receives a 2xx status code). It updates the serverMessage state with the data from the backend. The .catch() block handles any errors, providing detailed feedback to the UI, which is crucial for quick debugging during a hackathon.

At this point, if both the backend server and the React development server (npm start in the frontend directory) are running, the browser console will likely show a CORS error. This is expected and is the final piece of the integration puzzle.

Section 2.3: Demystifying and Conquering CORS

Cross-Origin Resource Sharing (CORS) is a browser security mechanism, not an application error. It is one of the most common points of confusion for developers new to the MERN stack. The browser enforces a "Same-Origin Policy," which prevents a web page from making requests to a different domain, protocol, or port than the one it was served from.[18]
In this setup, the React app is served from http://localhost:3000, and the Express API is running on http://localhost:5000. Because the ports are different, the browser considers them different "origins" and blocks the request from the frontend by default.[20]
The error message appears in the *browser's* console, leading many to mistakenly debug their

React code. However, the solution lies entirely on the *server*. The server must explicitly tell the browser which origins are allowed to access its resources.
The fix is to properly configure the cors middleware in the Express backend.

1. **Ensure cors is installed:** npm install cors in the backend directory.
2. **Configure it in server.js:** Replace the provisional app.use(cors()); with a more specific configuration.

JavaScript

```
// backend/server.js

//... (imports and initial setup)...

// 3. Middleware Setup
// Specific CORS configuration for the hackathon
const corsOptions = {
  origin: 'http://localhost:3000', // Allow only the React app to make requests
  credentials: true, // Allow cookies/authorization headers to be sent
};
app.use(cors(corsOptions));

app.use(express.json()); // Middleware to parse JSON bodies

//... (rest of the server code)...
```

Code Explanation:

- **origin: 'http://localhost:3000'**: This is the critical line. It tells the Express server to add an Access-Control-Allow-Origin header to its responses, but *only* for requests coming from http://localhost:3000. When the browser sees this header, it knows the server has permitted the request and allows it to proceed.[20]
- **credentials: true**: This option is important for future-proofing. If the application later adds authentication features that rely on cookies or authorization headers, this setting will be required to allow them to be passed in cross-origin requests.[19]

With this change, restart the backend server. Now, when the React app loads, the CORS error will be gone, and the success message "Server is healthy" will be displayed, confirming that the frontend and backend are fully and correctly integrated.

Part III: The AI Innovator's Guide: Building an Intelligent Chatbot with Gemini

This part addresses the core AI integration responsibilities using Google's Gemini API. The approach follows a logical progression, starting with a simple, stateless AI chat engine and evolving it into a sophisticated conversational agent with memory and the ability to consult external knowledge. This layered approach allows for strategic implementation based on hackathon time constraints, ensuring a functional product at each stage.

Section 3.1: Engineering the AI Conversation Engine (Backend)

The heart of the chatbot is a backend API endpoint that communicates with a Large Language Model (LLM). This endpoint will receive a user's message, forward it to the Gemini API, and return the model's response.
First, install the official Google Generative AI library in the backend.[34]

Bash

```bash
# In the backend directory
npm install @google/generative-ai
```

Next, add a new API route to backend/server.js to handle chat requests.

JavaScript

```javascript
// backend/server.js

//... (existing imports)
const { GoogleGenerativeAI } = require('@google/generative-ai');

//... (app initialization, middleware, DB connection)

// Initialize Google Gemini client
// IMPORTANT: The API key will be loaded from environment variables
const genAI = new GoogleGenerativeAI(process.env.GEMINI_API_KEY);
const model = genAI.getGenerativeModel({ model: 'gemini-pro' });

//... (existing /api/health route)

// POST endpoint for chat functionality
app.post('/api/chat', async (req, res) => {
  try {
    const { message } = req.body; // Extract user message from request body

    if (!message) {
      return res.status(400).json({ error: 'Message is required' });
    }

    // Call the Gemini API for a stateless, single-turn conversation
    const result = await model.generateContent(message);
    const response = await result.response;
    const aiResponse = response.text();

    // Send the AI's response back to the frontend
    res.status(200).json({ response: aiResponse });

  } catch (error) {
    console.error('Error calling Gemini API:', error);
    res.status(500).json({ error: 'Failed to get response from AI' });
  }
});

//... (app.listen)
```

Code Explanation:

- **Gemini Client:** An instance of the GoogleGenerativeAI client is created, configured with an API key that will be securely loaded from environment variables.[34] We then get a specific generative model, in this case,
  gemini-pro, which is well-suited for text-based chat.
- **API Route:** A POST endpoint at /api/chat is defined. It expects a JSON body with a message property containing the user's input.
- **Gemini API Call:** The model.generateContent(message) method sends the user's message to the Gemini model.[34] This is the simplest way to get a response for a single query.
- **Response Handling:** The function awaits the response, extracts the text content using response.text(), and sends it back to the client in a JSON object.[34] Robust error handling is included to manage potential API failures.

Section 3.2: Fort Knox Security for API Keys

Exposing an API key is a critical security vulnerability that can lead to unauthorized use and significant financial cost. It is imperative to handle keys securely by using environment variables and ensuring they are never committed to version control.[21]

1. **Create the .env File:** In the backend directory, create a file named .env (if it doesn't already exist). Add your Gemini API key to this file. You can obtain a key from Google AI Studio.[34]
```
# backend/.env
MONGODB_URI="..."
PORT=5000
GEMINI_API_KEY="YourSecretKeyHere"
```
2. **Verify .gitignore:** Open the root .gitignore file and confirm that /backend/.env is listed. This prevents the file containing the secret key from ever being uploaded to GitHub.[21]
3. **Load with dotenv:** The line require('dotenv').config(); at the top of backend/server.js is responsible for loading these variables into the Node.js environment via the process.env object. The Gemini client is then safely initialized using process.env.GEMINI_API_KEY.[22]

This separation of configuration from code is a fundamental best practice in modern software development.

Section 3.3: Crafting the React Chat Interface (Frontend)

With the backend ready, the focus shifts to building a user-friendly chat interface in React. A component-based architecture will keep the code organized and maintainable. The frontend code remains unchanged as the API contract (/api/chat) is the same.

Component Structure

Create three new components in the frontend/src/ directory:

1. **MessageList.js**: Displays the list of messages.
JavaScript
```javascript
// frontend/src/MessageList.js
import React from 'react';

const MessageList = ({ messages }) => {
  return (
    <div className="message-list">
      {messages.map((msg, index) => (
        <div key={index} className={`message ${msg.role}`}>
          {msg.content}
        </div>
      ))}
    </div>
  );
};

export default MessageList;
```

2. **MessageInput.js**: Provides the input field and send button.
   JavaScript
```
// frontend/src/MessageInput.js
import React, { useState } from 'react';

const MessageInput = ({ onSendMessage, isLoading }) => {
  const [inputValue, setInputValue] = useState('');

  const handleSubmit = (e) => {
    e.preventDefault();
    if (inputValue.trim() &&!isLoading) {
      onSendMessage(inputValue);
      setInputValue('');
    }
  };

  return (
    <form onSubmit={handleSubmit} className="message-input-form">
      <input
        type="text"
        value={inputValue}
        onChange={(e) => setInputValue(e.target.value)}
        placeholder="Type your message..."
        disabled={isLoading}
      />
      <button type="submit" disabled={isLoading}>
        {isLoading? '...' : 'Send'}
      </button>
    </form>
  );
};

export default MessageInput;
```

3. **ChatWindow.js**: The main container that orchestrates the other components and manages the application state.
   JavaScript
```
// frontend/src/ChatWindow.js
```

```
import React, { useState } from 'react';
import axios from 'axios';
import MessageList from './MessageList';
import MessageInput from './MessageInput';

const ChatWindow = () => {
  const [messages, setMessages] = useState();
  const [isLoading, setIsLoading] = useState(false);

  const handleSendMessage = async (userInput) => {
    // Add user's message to the state immediately for a responsive feel
    const newUserMessage = { role: 'user', content: userInput };
    setMessages(prevMessages => [...prevMessages, newUserMessage]);
    setIsLoading(true);

    try {
      // Send user's message to the backend
      const response = await axios.post('http://localhost:5000/api/chat', {
        message: userInput,
      });

      // Add AI's response to the state
      const aiMessage = { role: 'assistant', content: response.data.response };
      setMessages(prevMessages => [...prevMessages, aiMessage]);

    } catch (error) {
      console.error('Error sending message:', error);
      const errorMessage = { role: 'assistant', content: 'Sorry, I encountered an error.
Please try again.' };
      setMessages(prevMessages => [...prevMessages, errorMessage]);
    } finally {
      setIsLoading(false);
    }
  };

  return (
    <div className="chat-window">
      <MessageList messages={messages} />
      <MessageInput onSendMessage={handleSendMessage} isLoading={isLoading} />
    </div>
  );
};
```

```
export default ChatWindow;
```

Finally, replace the content of frontend/src/App.js to render the ChatWindow.

JavaScript

```
// frontend/src/App.js
import React from 'react';
import ChatWindow from './ChatWindow';
import './App.css'; // Make sure to add some basic styling

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <h1>Mastercard AI Chatbot</h1>
      </header>
      <main>
        <ChatWindow />
      </main>
    </div>
  );
}

export default App;
```

This setup creates a functional, stateless chatbot. Each question is treated as an independent event without any memory of past interactions.[17]

Section 3.4: Giving the Chatbot a Memory: Persisting History in MongoDB

To create a truly conversational experience, the chatbot must remember the context of the conversation. The Gemini SDK provides a stateful chat object that simplifies this process. We will store the chat history in MongoDB and use it to initialize the chat session for each user.[35]

Mongoose Schema Design

The data structure in MongoDB is critical. We will store messages embedded within a single

conversation document. Note the change in the role enum to accommodate Gemini's terminology (model instead of assistant).[36]
Create a new folder backend/models and add a file Conversation.js.

JavaScript

```
// backend/models/Conversation.js
const mongoose = require('mongoose');

const MessageSchema = new mongoose.Schema({
  role: {
    type: String,
    enum: ['user', 'model'], // Changed 'assistant' to 'model' for Gemini
    required: true,
  },
  content: {
    type: String,
    required: true,
  },
  timestamp: {
    type: Date,
    default: Date.now,
  },
});

const ConversationSchema = new mongoose.Schema({
  sessionId: {
    type: String,
    required: true,
    unique: true,
    index: true,
  },
  messages:,
});

module.exports = mongoose.model('Conversation', ConversationSchema);
```

Updated Backend Logic

Now, modify the POST /api/chat endpoint and add a new GET endpoint to retrieve history.

JavaScript

```
// backend/server.js
//... (imports)
const Conversation = require('./models/Conversation'); // Import the model

//... (Gemini client, etc.)

// NEW: GET endpoint to retrieve chat history for a session
app.get('/api/chat/history/:sessionId', async (req, res) => {
  try {
    const { sessionId } = req.params;
    const conversation = await Conversation.findOne({ sessionId });
    if (conversation) {
      // Map role 'model' back to 'assistant' for frontend compatibility
      const frontendMessages = conversation.messages.map(msg => ({
        role: msg.role === 'model'? 'assistant' : 'user',
        content: msg.content
      }));
      res.status(200).json(frontendMessages);
    } else {
      res.status(200).json(); // No history found, return empty array
    }
  } catch (error) {
    res.status(500).json({ error: 'Failed to retrieve chat history' });
  }
});

// UPDATED: POST endpoint for chat, now with memory
app.post('/api/chat', async (req, res) => {
  try {
    const { message, sessionId } = req.body;
    if (!message ||!sessionId) {
      return res.status(400).json({ error: 'Message and sessionId are required' });
    }

    // Find existing conversation or create a new one
    let conversation = await Conversation.findOne({ sessionId });
    if (!conversation) {
      conversation = new Conversation({ sessionId, messages: });
    }
    // Map database history to Gemini's expected format { role, parts: content }
    const geminiHistory = conversation.messages.map(msg => ({
      role: msg.role,
      parts: msg.content,
    }));

    // Start a chat session with the existing history
    const chat = model.startChat({ history: geminiHistory });
    const result = await chat.sendMessage(message);
    const response = await result.response;
    const aiResponseText = response.text();

    // Add new user message and AI response to our database record
    conversation.messages.push({ role: 'user', content: message });
    conversation.messages.push({ role: 'model', content: aiResponseText });

    // Save the updated conversation
    await conversation.save();

    res.status(200).json({ response: aiResponseText });

  } catch (error) {
    console.error('Error in stateful chat endpoint:', error);
    res.status(500).json({ error: 'Failed to get response from AI' });
  }
});

//... (app.listen)
```

Updated Frontend Logic

The ChatWindow.js component now needs to manage a sessionId and fetch history when it loads. This code remains the same as it is agnostic to the backend AI provider.

JavaScript

```
// frontend/src/ChatWindow.js
import React, { useState, useEffect } from 'react';
//... (imports)
```

```
const ChatWindow = () => {
  const [messages, setMessages] = useState();
  const [isLoading, setIsLoading] = useState(false);
  // A simple way to get a persistent session ID for the hackathon
  const [sessionId] = useState(() => localStorage.getItem('sessionId') |

| `session_${Date.now()}`);

  useEffect(() => {
    // Store session ID in local storage to persist across reloads
    localStorage.setItem('sessionId', sessionId);

    // Fetch chat history on component mount
    const fetchHistory = async () => {
      setIsLoading(true);
      try {
        const response = await axios.get(`http://localhost:5000/api/chat/history/${sessionId}`);
        setMessages(response.data);
      } catch (error) {
        console.error('Failed to fetch history:', error);
      } finally {
        setIsLoading(false);
      }
    };
    fetchHistory();
  }, [sessionId]);

  const handleSendMessage = async (userInput) => {
    const newUserMessage = { role: 'user', content: userInput };
    setMessages(prev => [...prev, newUserMessage]);
    setIsLoading(true);

    try {
      const response = await axios.post('http://localhost:5000/api/chat', {
        message: userInput,
        sessionId: sessionId, // Send sessionId with the request
      });
      const aiMessage = { role: 'assistant', content: response.data.response };
      setMessages(prev => [...prev, aiMessage]);
    } catch (error) {
      //... (error handling)
    } finally {
      setIsLoading(false);
    }
  };

  //... (return JSX)
};

export default ChatWindow;
```

This enhancement transforms the application into a stateful conversational agent, capable of maintaining context throughout a user's session.[24]

Section 3.5: The Final Evolution: An Introduction to RAG

Retrieval-Augmented Generation (RAG) is an advanced AI technique that grounds the LLM's responses in a specific, provided knowledge base. In simple terms, it's like giving the AI an "open book" for an exam. Instead of relying only on its vast but general training data, the model first *retrieves* relevant information from a trusted source and then uses that information to *generate* a more accurate and contextually appropriate answer.[26] This is a powerful way to make the chatbot an expert on a specific topic, such as Mastercard's initiatives.
For a hackathon, a full-fledged vector database implementation might be too time-consuming. However, a simplified "proof-of-concept" RAG system is highly achievable and impressive.

Simplified RAG Implementation

1. **Create a Knowledge Base:** In the backend directory, create a file named knowledge.txt. Populate it with specific facts relevant to the hackathon theme.
   ```
   # backend/knowledge.txt
   Mastercard's key initiatives are financial inclusion and digital economy transformation.
   The Mastercard Code for Change hackathon encourages innovation in fintech.
   A major goal is to develop solutions that empower small businesses.
   ```

2. **Modify the Backend:** Update the /api/chat endpoint to read this file and use it to provide context to the Gemini model at the start of the conversation.
   JavaScript
   ```
   // backend/server.js
   const fs = require('fs').promises; // Use promises-based file system module
   const path = require('path');
   //... (rest of the file)
   ```

```
//... inside the POST /api/chat endpoint, before the chat session starts
try {
  // 1. RETRIEVAL: Read the knowledge base
  const knowledgeBasePath = path.join(__dirname, 'knowledge.txt');
  const knowledgeBaseContent = await fs.readFile(knowledgeBasePath, 'utf-8');

  // 2. AUGMENTATION: Create an initial context for the AI
  const initialContext =;

  //... (find or create conversation in DB)

  // Map database history to Gemini's format
  const geminiHistory = conversation.messages.map(msg => ({
    role: msg.role,
    parts: msg.content,
  }));

  // Combine the RAG context with the user's chat history
  const fullHistory = [...initialContext,...geminiHistory];

  // 3. GENERATION: Start a chat session with the augmented history
  const chat = model.startChat({ history: fullHistory });
  const result = await chat.sendMessage(message);

  //... (rest of the logic: get response, save to DB, send to client)
} catch (error) {
   //...
}
```

With this change, if a user asks, "What are Mastercard's goals?", the chatbot will now provide a focused answer based on the content of knowledge.txt, rather than a generic answer from its pre-trained knowledge. This demonstrates a sophisticated understanding of modern AI architecture and is a powerful way to tailor the application to the hackathon's specific theme.[27]

Part IV: Team Synergy: A Hackathon Git Workflow

In the high-pressure, fast-paced environment of a hackathon, effective collaboration is paramount. A chaotic version control process can bring a team's progress to a grinding halt. This final part provides a simple, robust, and practical Git workflow designed to maximize team synergy and minimize friction. The focus is on clear communication and protecting the stability of the main codebase.

Section 4.1: Git & GitHub Quickstart

With the project structure initialized with git init in Part I, the next step is to create a shared, central repository on GitHub where all team members can push their changes and sync their work.

1. **Create the Remote Repository:** One team member should go to GitHub, create a new repository (e.g., mastercard-hackathon), and leave it empty (do not initialize it with a README or license file).
2. **Connect Local to Remote:** In the local project's root directory, link the local Git repository to the newly created remote repository on GitHub.[4]
   Bash
   ```
   git remote add origin <your-github-repository-url.git>
   ```
3. **Initial Push:** Make an initial commit with the project structure and push the main branch to GitHub. The -u flag sets the remote main branch as the "upstream" for the local main branch, simplifying future pushes.[4]
   Bash
   ```
   git add.
   git commit -m "Initial commit: project structure setup"
   git push -u origin main
   ```
4. **Team Members Clone:** All other team members can now clone this repository to their local machines to get the project started.[29]
   Bash
   ```
   git clone <your-github-repository-url.git>
   ```

Section 4.2: A Simple and Effective Branching Strategy

Working directly on the main branch is a recipe for disaster in a team setting. It inevitably leads to merge conflicts and a high risk of breaking the application for everyone. The **Feature Branch Workflow** is the industry standard and is perfectly suited for a hackathon's needs. It isolates work, facilitates collaboration, and protects the integrity of the main codebase.[29]
The core principle is simple: **the main branch should always be stable and deployable.** All new work happens on separate, temporary branches.

The Workflow in Practice:

1. **Start a New Task:** Before starting any new work (e.g., building the chat UI), always switch to the main branch and pull the latest changes to ensure the local version is up-to-date. Then, create a new, descriptively named branch for the task.[29]
   Bash
   ```
   # Switch to the main branch
   git checkout main

   # Pull the latest changes from the remote repository
   git pull origin main

   # Create and switch to a new feature branch
   git checkout -b feature/react-chat-ui
   ```
2. **Work and Commit:** Make changes and commit them frequently on the feature branch. Good commit messages are crucial for communication.
   Bash
   ```
   # After making some changes...
   git add.
   git commit -m "feat: create basic ChatWindow component"
   ```
3. **Share Your Work:** Push the feature branch to the remote repository on GitHub. This makes the work visible to teammates and backs it up.
   Bash
   ```
   git push -u origin feature/react-chat-ui
   ```
4. **Integrate via Pull Request (PR):** When the feature is complete and tested locally, go to the GitHub repository. A prompt will likely appear to create a Pull Request from the recently pushed branch. Create the PR, give it a title and description, and assign a teammate to review it.
5. **Review and Merge:** The PR serves as a communication checkpoint. A teammate can briefly review the code for obvious issues. Once approved, the PR can be merged into the main branch via the GitHub interface. This keeps the merge process clean and documented.
6. **Clean Up:** After merging, the feature branch can be deleted from the remote repository through the GitHub UI.

This cycle (pull from main -> create branch -> work -> push branch -> PR to main) ensures that main remains a stable source of truth, preventing one person's work-in-progress from blocking the rest of the team.[31]

Section 4.3: The Ultimate Hackathon Git Cheat Sheet

During a high-stress event, it's easy to forget command syntax. This cheat sheet maps common hackathon goals to the specific Git commands needed to achieve them, serving as a quick and reliable reference.

Task / Goal	Command Sequence	Explanation for Hackathon Context
Start a new feature	git checkout main git pull origin main git checkout -b feature/my-new-idea	Always start from the latest stable version of main to avoid conflicts later. Use a descriptive branch name.[32]
Save local progress	git add. git commit -m "feat: implemented user login API"	Commit small, logical chunks of work frequently. A good message explains *what* and *why*.[32]
Share work with the team	git push -u origin feature/my-new-idea	Pushes the branch to GitHub so others can see it and it's safely backed up. Use -u the first time you push a new branch.[33]
Get updates from teammates	git checkout main git pull origin main	Syncs your local main branch with the latest changes that have been merged on GitHub.[32]
Update feature branch with latest main	git checkout feature/my-new-idea git merge main	After pulling updates to main, merge those changes into your feature branch to keep it current and resolve conflicts early.
See what has changed	git status	Shows which files are modified, staged, or untracked. The most-used Git command.[32]
Discard local changes	git checkout -- <file-name>	Throws away all changes in a specific file since the last commit. Use with caution! [33]
Fix the last commit message	git commit --amend -m "A better commit message"	Corrects a typo or improves the message of the most recent commit before pushing.

Handle a merge conflict	1. git status to see conflicted files. 2. Open the file in VS Code. 3. Manually edit to resolve <<<<<<<, =======, >>>>>>> markers. 4. git add <resolved-file-name> 5. git commit	When Git can't automatically merge changes, it's a signal for human intervention. Communication with the teammate who wrote the conflicting code is the fastest way to resolve it.[30]

Conclusion

This comprehensive playbook provides the foundational knowledge, practical code, and strategic workflows necessary to excel in the assigned roles at the Mastercard Code for Change hackathon. By systematically establishing a robust development environment, mastering the frontend-backend connection, and progressively building a sophisticated AI chatbot, a developer can confidently tackle the technical challenges. The evolution from a simple, stateless AI to a conversational agent with memory and external knowledge via RAG demonstrates a deep, modern skillset.
Furthermore, the adoption of a simple yet effective Git feature branch workflow is the key to unlocking team synergy under pressure. It transforms version control from a potential obstacle into a powerful tool for communication and collaboration, ensuring the main branch remains stable and the team maintains forward momentum.
Success in a hackathon is a function of preparation, strategy, and execution. This guide equips the developer with all three, turning a feeling of being "severely underprepared" into a state of readiness and empowerment. The path is now clear to not just participate, but to innovate, build, and make a significant impact.

Works cited

1. Comprehensive MERN Stack Development Guide - Ropstam Solutions, accessed on August 24, 2025, https://www.ropstam.com/mern-stack-development-guide/
2. MERN Stack: A Guide for New Developers - HackerNoon, accessed on August 24, 2025, https://hackernoon.com/mern-stack-a-guide-for-new-developers
3. Setting Up Your First MERN Stack Project: A Step-by-Step Tutorial ..., accessed on August 24, 2025, https://medium.com/@sindoojagajam2023/setting-up-your-first-mern-stack-project-a-step-by-step-tutorial-0a4f88fa4e98
4. MERN Stack Project SetUp - A Complete Guide - GeeksforGeeks, accessed on August 24, 2025, https://www.geeksforgeeks.org/git/mern-stack-project-setup-a-complete-guide/
5. A Comprehensive Guide To MERN Stack Web Development For Beginners - Pesto Tech, accessed on August 24, 2025, https://pesto.tech/resources/a-comprehensive-guide-to-mern-stack-web-develo

pment-for-beginners
6. Most Useful Visual Studio Code Extensions for MERN Stack ..., accessed on August 24, 2025, https://medium.com/apptastic-coder/most-useful-visual-studio-code-extensions-for-mern-stack-development-6c12658ff8de
7. VS Code Extensions For MERN - GitHub, accessed on August 24, 2025, https://github.com/prince-noman/VS_Code_Extensions_For_MERN_Stack/blob/main/VS_CODE_EXTENSIONS_FOR_MERN.md
8. The 7 VS-Code Plugins for MERN Stack developers - Medium, accessed on August 24, 2025, https://medium.com/@ibrahimhz/the-7-vs-code-plugins-for-mern-stack-developers-b40891dc6b1e
9. Top 10 VS Code Extensions for MERN Developers - DEV Community, accessed on August 24, 2025, https://dev.to/nadim_ch0wdhury/top-10-vs-code-extensions-for-mern-developers-1nga
10. Best Vs code extensions you use? : r/Frontend - Reddit, accessed on August 24, 2025, https://www.reddit.com/r/Frontend/comments/18f9fto/best_vs_code_extensions_you_use/
11. Node JS Express for backend and React JS for frontend - Stack Overflow, accessed on August 24, 2025, https://stackoverflow.com/questions/44645722/node-js-express-for-backend-and-react-js-for-frontend
12. Building a Powerful API with Node.js, Express, and React | by August - Medium, accessed on August 24, 2025, https://medium.com/@2957607810/building-a-powerful-api-with-node-js-express-and-react-62bffe10d099
13. Creating a React, Node, and Express App - DEV Community, accessed on August 24, 2025, https://dev.to/techcheck/creating-a-react-node-and-express-app-1ieg
14. AI-Powered Chatbot Platform with Node and Express.js ..., accessed on August 24, 2025, https://www.geeksforgeeks.org/node-js/ai-powered-chatbot-platform-with-node-and-express-js/
15. Express - Node.js web application framework, accessed on August 24, 2025, https://expressjs.com/
16. Build a Fullstack Chatbot in 20 minutes with React and Node.js: A Step-By-Step Guide, accessed on August 24, 2025, https://medium.com/rewrite-tech/build-your-own-fullstack-chatbot-with-react-and-node-js-a-step-by-step-guide-922b392bfbf2
17. Building a Chat UI with React - NamasteDev Blogs, accessed on August 24, 2025, https://namastedev.com/blog/building-a-chat-ui-with-react-13/
18. Node.js CORS Guide: What It Is and How to Enable It, accessed on August 24, 2025, https://www.stackhawk.com/blog/nodejs-cors-guide-what-it-is-and-how-to-ena

ble-it/
19. Understanding CORS: A Comprehensive Guide with MERN Stack Implementation Using Axios | by Rahul, accessed on August 24, 2025, https://techtalkstrend.medium.com/understanding-cors-a-comprehensive-guide-with-mern-stack-implementation-using-axios-1bebc7f9dda7
20. Fixing CORS Errors in MERN Stack - DEV Community, accessed on August 24, 2025, https://dev.to/prathvihan108/fixing-cors-errors-in-mern-stack-fph
21. Best Practices for API Key Safety | OpenAI Help Center, accessed on August 24, 2025, https://help.openai.com/en/articles/5112595-best-practices-for-api-key-safety
22. How To Secure an API Key In JavaScript - DEV Community, accessed on August 24, 2025, https://dev.to/theudemezue/how-to-secure-an-api-key-in-javascript-51bh
23. Create a React component that implements a basic chat interface with the following requirements : r/reactjs - Reddit, accessed on August 24, 2025, https://www.reddit.com/r/reactjs/comments/1iaec60/create_a_react_component_that_implements_a_basic/
24. Building an AI Agent With Memory Using MongoDB, Fireworks AI, and LangChain, accessed on August 24, 2025, https://www.mongodb.com/developer/products/atlas/agent-fireworksai-mongodb-langchain/
25. chat app with MongoDB Chat Message History using langchain| Tutorial:113 - YouTube, accessed on August 24, 2025, https://www.youtube.com/watch?v=0AYzyQ5qKBM&pp=0gcJCfwAo7VqN5tD
26. How to build an AI-Powered Retrieval-Augmented Generation (RAG ..., accessed on August 24, 2025, https://dev.to/omogbai/how-to-build-an-ai-powered-retrieval-augmented-generation-rag-chatbot-assistant-with-typescript-58ol
27. Build a RAG System with Node.js & OpenAI - Zignuts Technolab, accessed on August 24, 2025, https://www.zignuts.com/blog/build-rag-system-nodejs-openai
28. Building a Fullstack RAG (Retrieval-Augmented Generation) System with Node.js, React, and LLM | by MD. SHARIF ALAM | Towards Dev - Medium, accessed on August 24, 2025, https://medium.com/towardsdev/building-a-rag-retrieval-augmented-generation-system-with-node-js-react-and-llm-d7c362d5688c
29. GITHUB Hackathon | PDF | Computing | Software - Scribd, accessed on August 24, 2025, https://www.scribd.com/document/873328190/GITHUB-Hackathon
30. Git cheat sheet | Atlassian Git Tutorial, accessed on August 24, 2025, https://www.atlassian.com/git/tutorials/atlassian-git-cheatsheet
31. Git for Hackathons - Brandon Nicoll, accessed on August 24, 2025, https://nicoll.io/posts/git-for-hackathons/
32. GIT CHEAT SHEET - GitHub Education, accessed on August 24, 2025, https://education.github.com/git-cheat-sheet-education.pdf
33. Git Cheat Sheet - GitLab, accessed on August 24, 2025, https://about.gitlab.com/images/press/git-cheat-sheet.pdf

34. Integrating Google Gemini to Node.js Application | by Reetesh ..., accessed on August 24, 2025, https://medium.com/@rajreetesh7/integrating-google-gemini-to-node-js-application-e45328613130
35. Create a multi-turn non-streaming conversation with Vertex AI ..., accessed on August 24, 2025, https://cloud.google.com/vertex-ai/docs/samples/aiplatform-gemini-multiturn-chat-nonstreaming
36. Building a Chat Integration with Google Gemini - Raymond Camden, accessed on August 24, 2025, https://www.raymondcamden.com/2024/04/30/building-a-chat-integration-with-google-gemini