INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
HYDERABAD

# **Project Title**:

Conversational Requirement Elicitation of Reporting Request



**Supervisor**: Dr. Sai Anirudh Karre

**Team Number**: 21

**Team Member**:

| S. No. | Registration no. | Name |
|---|---|---|
| **1.** | 2024202024 | Amarnath kumar |
| **2.** | 2024201015 | Abhirup Bhattacharya |
| **3.** | 2024201064 | Md Fraz Uddin |
| **4.** | 2024201071 | Suraj Lalwani |

**International Institute of Information Technology, Hyderabad**
(**session**: 2024-26)

# Acknowledgment:

We extend our sincere gratitude to our supervisor, Dr. Sai Anirudh Karre, for his invaluable guidance, encouragement, and support throughout the development of this project. His insights and expertise were crucial in shaping the direction and execution of our work.

We would also like to express our appreciation to our institution for providing us with the resources and platform necessary to carry out this project. The collaborative environment and access to tools significantly contributed to our learning and the success of this project.

Lastly, we are thankful to our peers for their continuous encouragement and understanding, which motivated us to persevere through challenges and achieve our objectives.

# Table of content:

# About Project:

## Objective

The project aims to design a conversational tool that enables a Business Analyst to gather and categorize reporting requirements through conversational interactions. The tool focuses on improving efficiency and user engagement while summarizing requirements in a machine-processable format for data engineers.

Project implements a web-based system with separate **frontend** and **backend** components. The structure is a full-stack application designed to handle dynamic user interactions, data processing, and potentially database management.

### System Features:

- ➢ Functional Features:
  - Conversational chatbot interface for eliciting requirements.
  - Real-time updates between frontend, backend, and database.

- ➢ Non-functional Features:
  - **Scalability**: Handles concurrent user interactions efficiently.
  - **Reliability**: Maintains data consistency and error handling during runtime.
  - **Usability**: Ensures an intuitive and seamless user experience through clear design and functionality.

# Implementation detail:

The implementation of the project is structured into several components to ensure a modular and scalable design. Below are the details:

## ➢ Frontend Development:

The frontend development involves creating an interactive user interface using React.js. The `ChatInterface` component serves as the primary interaction point, while `ChatBubble` manages individual user and system messages. These components are dynamically styled using CSS to adapt to different devices and screen sizes. Axios is integrated for API communication, allowing the frontend to send and receive data from the middleware.

## ➢ Middleware Implementation:

The middleware is implemented using Express.js. It provides API endpoints for operations such as retrieving and submitting user queries. Middleware logic ensures that user inputs are validated before being processed. For example, inputs are checked for correct formatting and authentication tokens are verified. This layer also applies business rules to structure data before sending it to the backend or returning responses to the client.

## ➢ Backend Development:

The backend is built with Node.js and acts as the core processing engine. It uses Mongoose to interact with the MongoDB database, allowing seamless execution of CRUD (Create, Read, Update, Delete) operations. For instance, when a user submits a query, the backend retrieves relevant meta-model data, applies logic to filter results, and formats the output as per user requirements. The backend also implements error-handling mechanisms to manage unexpected inputs or database issues.

## ➢ Database Setup:

MongoDB serves as the database, storing user queries, meta-models, and generated reports. Collections are structured to support flexible querying, enabling real-time data updates. The database schema is designed to align with the meta-model requirements, ensuring compatibility with user inputs and facilitating efficient data retrieval.

## ➢ Testing and Quality Assurance:

The project undergoes rigorous testing to ensure reliability and performance. Performance testing evaluates system responsiveness under different loads, ensuring minimal latency and optimal user experience.

Comprehensive testing is performed to validate functionality, performance, and reliability.

- **Unit Testing:** Verifies individual components such as APIs, frontend elements, and database queries.
- **Integration Testing:** Ensures seamless communication between the frontend, middleware, and backend.
- **Performance Testing:** Assesses the system under varying loads, ensuring responsiveness and scalability

# Technology Stack:

Here is the detail of Technology Stack and Purpose:

## ➢ Frontend:

- **React.js**:  Modular, component-based UI design that ensures scalability and reusability.
- **Axios/Fetch:**A promise-based HTTP client for seamless API communication.
- **CSS**: Provides custom styling for UI components, ensuring a responsive and visually appealing interface.

## ➢ Backend:

- **Node.js:**Provides a lightweight and efficient environment for handling asynchronous operations and managing the backend logic
- **Express.js**: A middleware framework that simplifies routing, request handling, and integration with external services..
- **MongoDB**: A NoSQL database for flexible and scalable data storage.

## ➢ Tools and libraries:

- **Mongoose**: Simplifies interactions with MongoDB by providing a schema-based solution for modeling application data.
- **Nodemon**: Facilitates development by automatically restarting the server on code changes.
- **JSON/XML Parsing**: Enables the system to process meta-model files for customized report generation.
- **Git**: Version control for collaboration.

# Control flow and Data flow using diagrams:

## System Scope

➢ Inputs
- Meta-Model File: JSON/XML containing predefined attributes for reporting requirements.
- User Queries: Interactions through a chatbot interface.

➢ Outputs
- Visual data representations.
- Dynamic summaries of requirements.

## System Design

➢ Architecture

The system is based on a Client-Server-Middleware Architecture, ensuring modularity, scalability, and clear separation of concerns.

➢ Components:
- Client: React-based frontend for user interaction.
- Middleware: Express.js handles authentication, validation, and data processing.
- Server: Node.js for backend logic and database queries.
- Database: MongoDB cloud for real-time operations.

➢ System Design Diagram:

[Client] ⇔ [Middleware] ⇔ [Server] ⇔ [Database]

➢ Data Flow

The data flow of the system starts with user interaction through the chatbot interface. Users input their queries, which are sent to the middleware using Axios. The middleware authenticates the request and forwards it to the server for processing. The server communicates with the MongoDB to retrieve or store the required data. The processed response is returned to the middleware, which

applies business logic and sends it back to the frontend for display. Users can view the results, export reports, or interact further.

- Control Flow
  - Frontend

    The frontend is built using React.js and provides an intuitive user interface. Components such as ChatInterface and ChatBubble are used to display messages and capture user input. These components are styled with CSS to ensure responsiveness and user-friendly design. The Axios library is used for making HTTP requests to the middleware.

  - Middleware

    The middleware is implemented using Express.js. It acts as a bridge between the frontend and the backend, handling API endpoints for different operations such as meta-model retrieval and submission. /api/meta-model/list /api/meta-model/upload /api/conversation/start /api/conversation/save
    Middleware logic includes input validation, authentication, and business rule application to ensure that requests are processed securely and accurately.

  - Backend

    The backend is developed in Node.js and performs the core application logic. It handles requests forwarded by the middleware, interacts with the MongoDB database using the Mongoose library, and executes dynamic queries. The backend is responsible for data integrity and consistency. The backend also handles forwarding of meta model data to the frontend UI.

  - Database

    The database is responsible for storing all the meta models as well as all the conversions made with the chatbot to date. It is also responsible for storing the list of authenticated users which have appropriate privileges to upload meta models to the backend.

# Views and Diagrams:

## user view:

The **User View** provides an overview of the interaction between the user and the system. It represents how the user engages with the system's features through the frontend interface. Users interact with the application via the **chatbot interface**, entering queries or requirements that are processed in real time.

Key actions include:

- Submitting queries to retrieve or generate reports.
- Viewing dynamically updated results and visualizations.
- Exporting reports or summaries based on their requirements.

The **User View** emphasizes the simplicity and responsiveness of the user interface, ensuring an intuitive and engaging experience for all users. The seamless interaction flow between the user, frontend, and backend ensures that user requirements are efficiently handled and displayed.

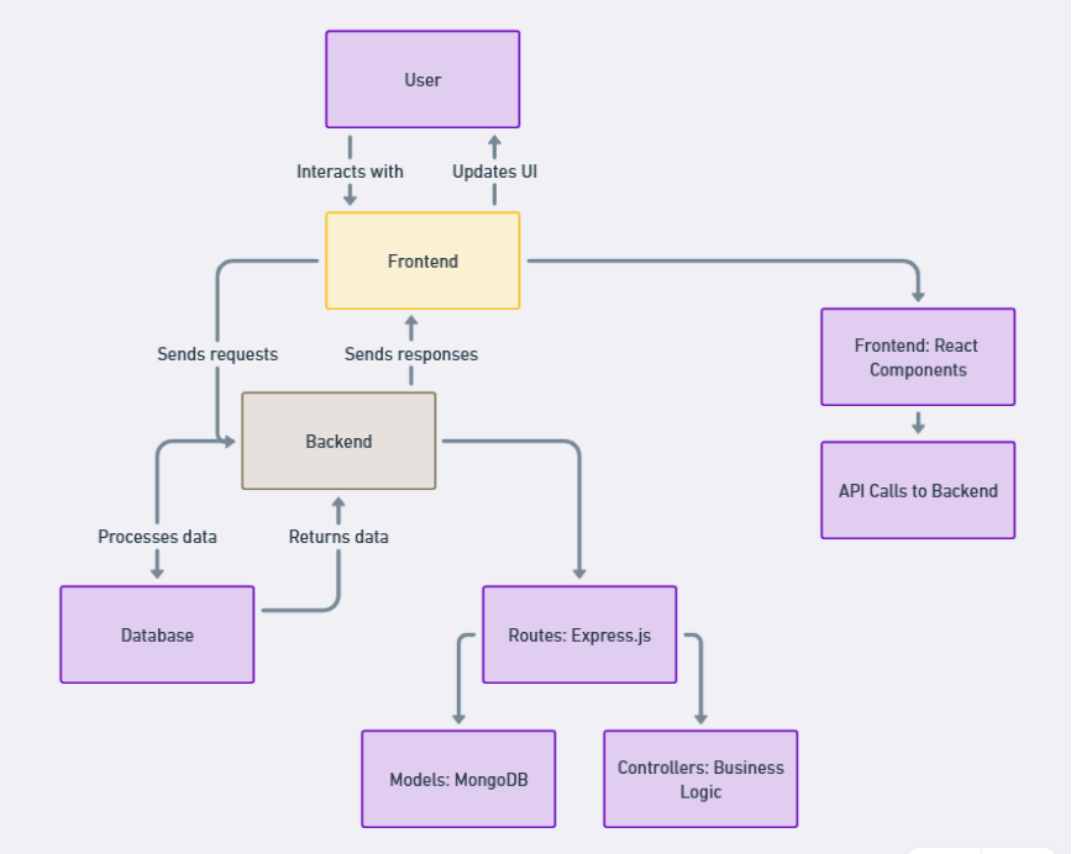


**figure**: Image shows the overall view of whole system. User interacts with front-end file and which takes data from backend file which usage the shown stacks.

## System level view front end:

This system design illustrates the flow of user input through React components, state management (using Context or Redux), and API interactions. User actions trigger API calls, and backend responses are processed to update the UI dynamically. Error handling ensures robustness, while caching improves performance. Finally, React renders dynamic content for a seamless user experience.

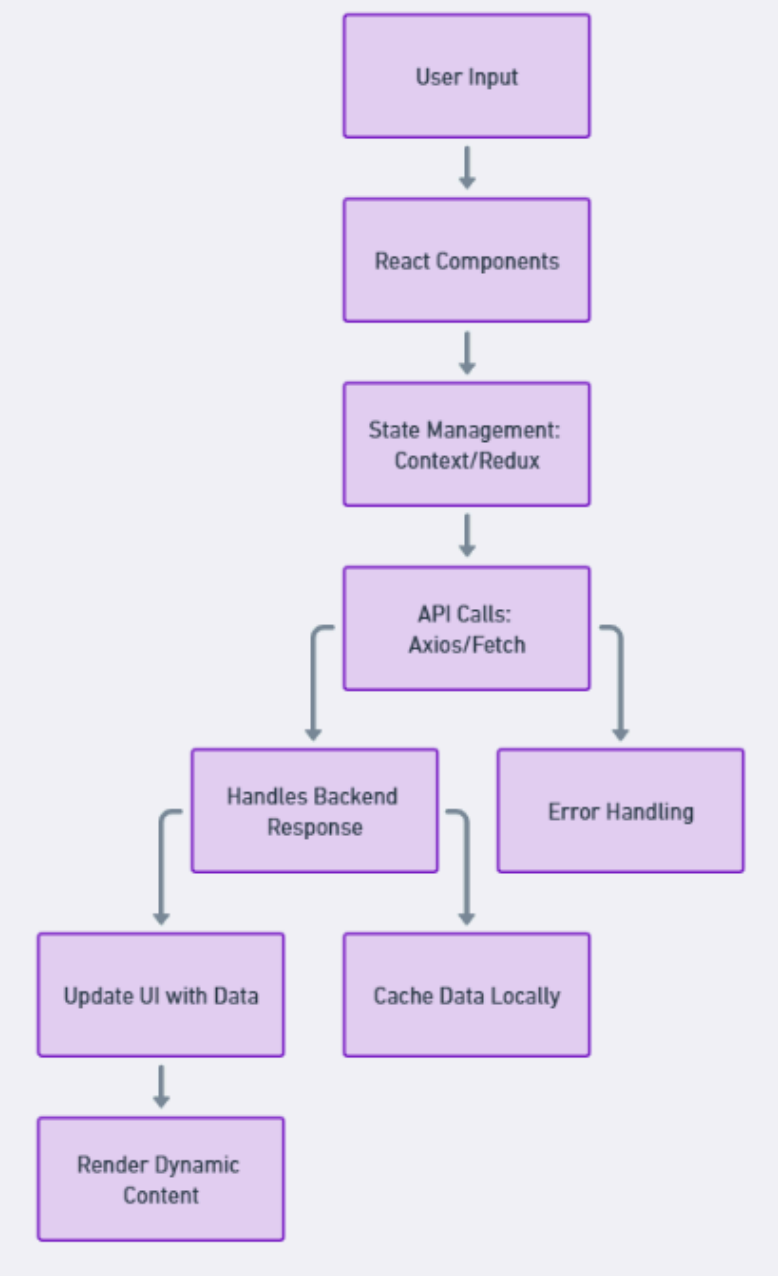


**figure:** User input to dynamic UI rendering via React, State management, and API interactions."User input to dynamic UI rendering via React, state management, and API interactions."

## System level view of Back end:

This backend system design outlines the flow of an incoming API request through various stages. Requests are routed via Express.js, passing through middleware for authentication and validation. Controller functions handle the core logic, interacting with a MongoDB database for queries. Results are processed, errors are managed, and the API response is prepared and sent to the frontend, ensuring secure and efficient request handling.

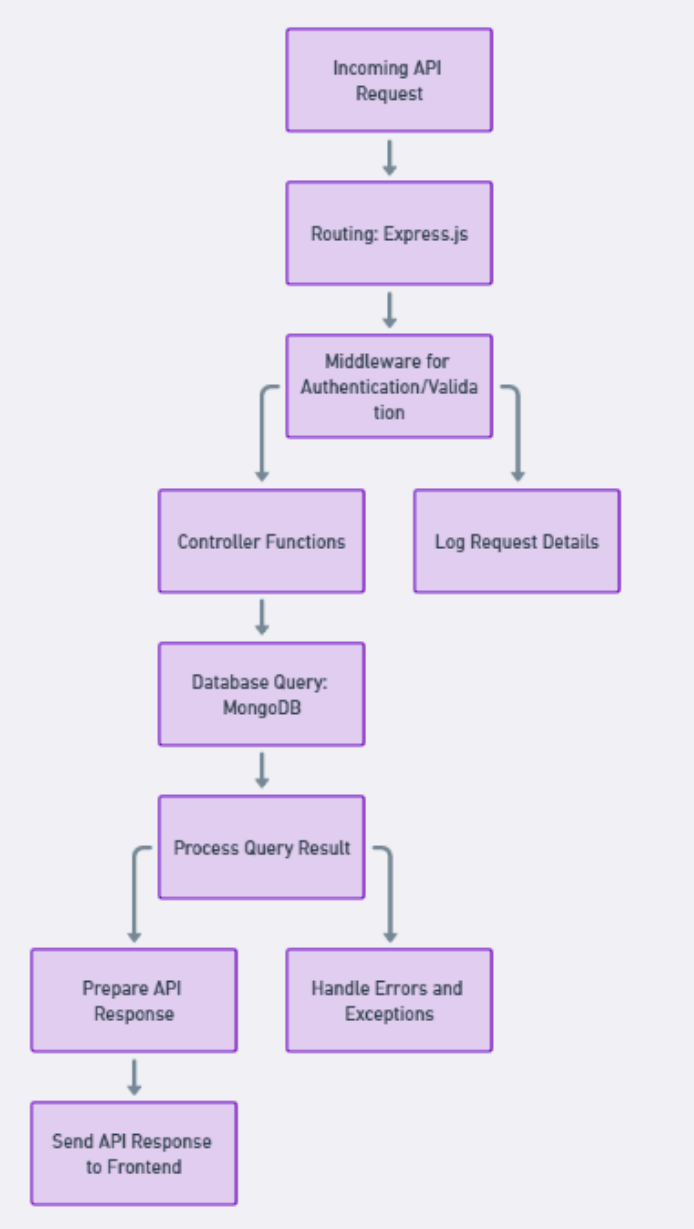


**figure**:API request processing via Express.js routing, middleware, database queries (MongoDB), and response handling.

## Data flow and framework:

The diagram illustrates the **Data Flow and Frameworks Used** in the project. The **Data Flow** begins with user interactions on the frontend, where requests are sent to the backend for processing. The backend interacts with the database to store or retrieve relevant data. The processed information is then sent back to the frontend, which updates the user interface dynamically.

The **Frameworks Used** include:

- **Frontend:** Built using React and JavaScript, with Axios facilitating API communication.
- **Middleware:** Powered by Express.js for secure and efficient request handling.
- **Backend:** Developed with Node.js, utilizing Mongoose for seamless interaction with the database.
- **Database:** MongoDB serves as the database, providing scalable and flexible data storage.

This architecture ensures a smooth and efficient flow of data between the user interface, server, and database, maintaining system reliability and performance.

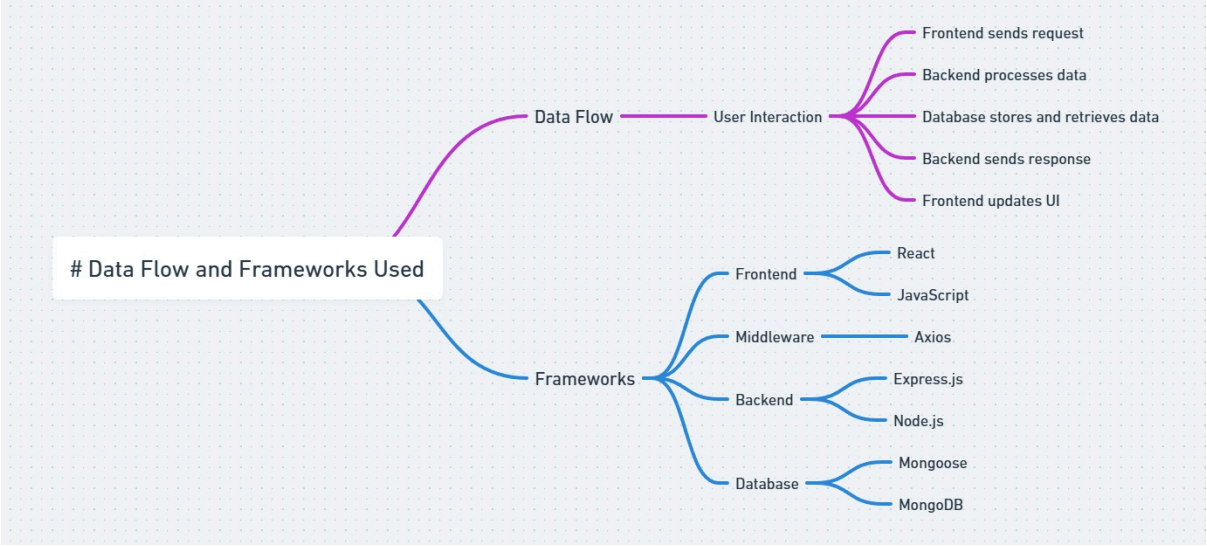


**figure**:Visual representation of user interaction, data flow between frontend (React), backend (Node.js with Express.js), and database (MongoDB).

## low level flow chart:

The diagram presents a **Low-Level Flow Chart** detailing the user-level and system-level functions of the project.

- **User-Level Functions:**
  - **Login/Signup:** Validates user credentials and provides access to meta-model upload features.
  - **Search Functionality:** Retrieves filtered results from the backend and displays them to the user.
  - **Error Notifications:** Notifies users about failed operations or errors during interactions.
- **System-Level Functions:**
  - **Authentication:** Utilizes encryption libraries (e.g., bcrypt) for secure password handling.
  - **Data Processing:** Fetches meta-model data from the database and stores conversational requirements securely.
  - **Error Handling:** Logs errors for debugging and sends appropriate HTTP status codes to the frontend.
  - **API Security:** Implements input sanitization to prevent malicious activities.
  - **Deployment:** Ensures version control integration for efficient deployment and updates.

This flow chart effectively demonstrates the integration of user-facing features with back-end capabilities, ensuring a seamless and secure system functionality.

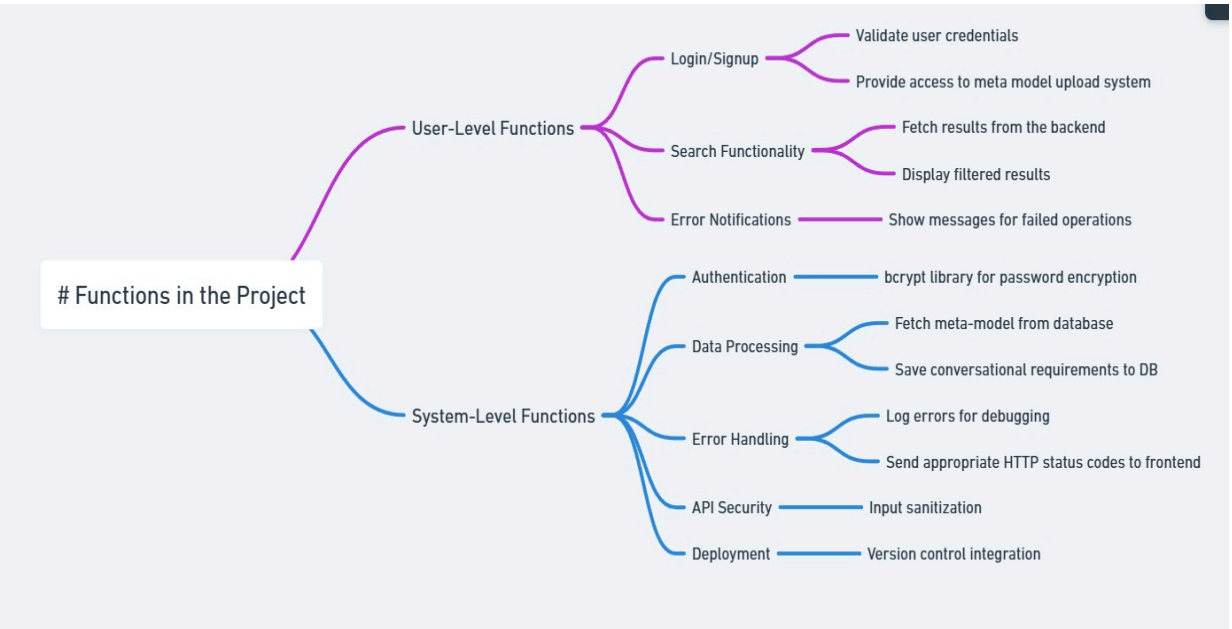


**figure:**User-Level and System-Level Capabilities.

# Use Cases:

## Use Cases of the Project

Here are the primary use cases based on the structure and components of the project:

### 1. User Authentication and Authorization

- **Actors**: User, System
- **Description**: Allow users to sign up, log in, and upload relevant meta models.
- **Steps**:
    - User provides credentials and a meta model JSON file.
    - Backend validates credentials and stores meta models in MongoDB database.
    - Only authenticated users gain access to restricted parts of the application backend.

---

### 2. Dashboard Access and Overview

- **Actors**: User
- **Description**: Users can access a chatbot for facilitating generation of report summary and requirement elicitation.
- **Steps**:
    - Backend fetches user-specific meta model from the database.
    - Data is processed and sent to the frontend.
    - Frontend collects the requirements by communicating with the user and produces a summary of requirements.
    - Requests the Backend to store the current conversational requirements to the database.

---

### 3. Profile Management

- **Actors**: User, System
- **Description**: Relevant users have their own profile which enables them to upload their own custom meta models to the backend.
- **Steps**:
    - User requests the backend admin to create an account
    - Backend validates and stores the new user in the database.
    - Updated data is used by frontend to add new meta models.

---

**4. Data Query and Search**

- **Actors**: User
- **Description**: Allow users to search for specific data within the system.
- **Steps**:
    - User inputs a query into the search bar.
    - Backend processes the query and retrieves matching records from the database.
    - Frontend displays the results in a user-friendly format.

---

**5. Data Visualization and Reporting**

- **Actors**: User
- **Description**: Provide graphical representations of data (e.g., charts, graphs) to users for easy analysis.
- **Steps**:
    - Backend aggregates data and formats it for visualization.
    - Frontend renders the data using visualization libraries.
    - Users can download or interact with the reports.

---

**6. Notifications and Alerts**

- **Actors**: System, User
- **Description**: Notify users about critical actions or errors (e.g., login failures, updates).
- **Steps**:
    - System detects a relevant event (e.g., an error or success).
    - Notification is sent to the user via the frontend.
    - Users acknowledge or take action based on the notification.

---

**7. API Interaction and Integration**

- **Actors**: System
- **Description**: APIs allow external systems to interact with the project (e.g., exporting data or integrating third-party services).
- **Steps**:
    - External system sends a request to the backend API.
    - Backend processes the request and responds with the necessary data.

**8. Error Handling and Recovery**

- **Actors**: System, User
- **Description**: Ensure smooth recovery from system failures or invalid user actions.
- **Steps**:
    - Backend detects errors (e.g., invalid input, database issues).
    - Error details are logged for debugging.
    - A user-friendly error message is displayed on the frontend.

---

**9. Deployment and Maintenance**

- **Actors**: System Administrators
- **Description**: Automate deployment processes and monitor the system for performance and updates.
- **Steps**:
    - Developers push changes to a Git repository.
    - CI/CD pipelines deploy the latest code to production.
    - Logs and analytics are monitored for maintenance.

# Explanation of code:

## Directory structure:

The project codebase is well-structured and organized into two primary components: **Backend** and **Frontend**, with each component designed to ensure modularity and scalability.

Below is an overview of the directory structure and its key elements:

### ➢ Backend:

- `.env`: Configuration file for environment variables.
- `app.js`: Main application entry point.
- `models`: Likely contains database models.
- `routes`: Handles the API endpoints.
- `package.json` **and** `package-lock.json`: Manage dependencies and scripts.

### ➢ Frontend:

- `.gitignore`: Files to ignore in version control.
- `meta-model-json`: May store model metadata (possibly related to the UI or API structure).
- `package.json` **and** `package-lock.json`: Manage frontend dependencies and scripts.
- `public`: Likely contains static assets (e.g., images, HTML).
- `src`: Contains the main source code for the frontend.
- `README.md`: Documentation.

# Source Code:

Conversation Route code:

```
// backend/routes/conversationRoutes.js
const express = require('express');
const router = express.Router();
const Conversation = require('../models/Conversation');
const MetaModel = require('../models/MetaModel');

// Start Conversation
router.post('/start', async (req, res) => {
  const { metaModel } = req.body;
  try {
    const model = await MetaModel.findOne({ "data.name": metaModel });
    if (!model) {
      return res.status(404).json({ error: 'Meta-model not found.' });
    }

    res.status(200).json({ metaModel: model.data });
  } catch (error) {
    console.error('Error starting conversation:', error);
    res.status(500).json({ error: 'Failed to start conversation.' });
  }
});


// Save Conversation
router.post('/save', async (req, res) => {
  try {
    const conversation = new Conversation({
      userResponses: req.body.userResponses,
      summary: req.body.summary,
    });
    await conversation.save();
    res.status(200).json({ message: 'Conversation saved successfully.'
});
  } catch (error) {
    res.status(500).json({ error: 'Failed to save conversation.' });
  }
});
```

```
module.exports = router;
```

Meta Model Route Code:

```
// backend/routes/metaModelRoutes.js
const express = require('express');
const router = express.Router();
const MetaModel = require('../models/MetaModel');
const User = require('../models/User');
const bcrypt = require('bcrypt');

// Authentication Middleware
const authenticateUser = async (req, res, next) => {
  const { email, password } = req.body;
  if (!email || !password) {
    return res.status(400).json({ error: 'Email and password are
required.' });
  }

  try {
    const user = await User.findOne({ email });
    if (!user || !(await bcrypt.compare(password, user.password))) {
      return res.status(401).json({ error: 'Unauthorized. Invalid
credentials.' });
    }
    next();
  } catch (error) {
    res.status(500).json({ error: 'Authentication failed.' });
  }
};

// Upload MetaModel
router.post('/upload', authenticateUser, async (req, res) => {
  try {
    const metaModel = new MetaModel({ data: req.body.fileContent });
    await metaModel.save();
    res.status(200).json({ message: 'Meta-model uploaded successfully.'
});
  } catch (error) {
    res.status(500).json({ error: 'Failed to upload meta-model.' });
```

```
  }
});

// Get Latest MetaModel
router.get('/', async (req, res) => {
  try {
    const metaModel = await MetaModel.findOne().sort({ _id: -1 });
    res.status(200).json(metaModel);
  } catch (error) {
    res.status(500).json({ error: 'Failed to fetch meta-model.' });
  }
});

// Get List of MetaModels
router.get('/list', async (req, res) => {
  try {
    const metaModels = await MetaModel.find({}, { _id: 0, "data.name":
1 });
    const metaModelNames = metaModels
      .map((model) => model?.data?.name)
      .filter((name) => name !== undefined);
    res.status(200).json({ metaModels: metaModelNames });
  } catch (error) {
    console.error('Error fetching meta-model list:', error);
    res.status(500).json({ error: 'Failed to fetch meta-model list.'
});
  }
});


module.exports = router;
```

Frontend Chat Interface:

```
// frontend/src/components/ChatInterface.js
import React, { useState, useEffect } from 'react';
import axios from 'axios';
import ChatBubble from './ChatBubble';
import Summary from './Summary';
import './chatInterface.css'
```

```
function ChatInterface() {
  const [messages, setMessages] = useState([]);
  const [metaModel, setMetaModel] = useState(null);
  const [userInput, setUserInput] = useState('');
  const [conversationOver, setConversationOver] = useState(false);
  const [summary, setSummary] = useState('');
  const [metaModelsList, setMetaModelsList] = useState([]);
  const [validResponses, setValidResponses] = useState([]);

  useEffect(() => {
    axios
      .get('http://localhost:5000/api/meta-model/list')
      .then((res) => {
        const metaModels = res.data.metaModels;
        setMetaModelsList(metaModels);

        const initialMessage = `Hello! Please select a meta-model to
start: ${metaModels.join(', ')}`;
        setMessages([{ sender: 'bot', text: initialMessage }]);
      })
      .catch((err) => {
        console.error('Error fetching meta-model list:', err);
        setMessages([{ sender: 'bot', text: 'Error: Could not fetch
meta-model list. Please try again later.' }]);
      });
  }, []);

  const startConversation = (selectedMetaModel) => {
    axios
      .post('http://localhost:5000/api/conversation/start', {
metaModel: selectedMetaModel })
      .then((res) => {
        if (!res.data.metaModel) {
          throw new Error('MetaModel data is null or undefined');
        }
        setMetaModel(res.data.metaModel);
        const initialQuestion = `Hello! What type of report do you
need? Here are the options: ${res.data.metaModel.report_types.map(rt =>
rt.label).join(', ')}.`;
        setMessages((prev) => [...prev, { sender: 'bot', text:
initialQuestion }]);
      })
```

```
      .catch((err) => {
        console.error('Error starting conversation:', err);
        setMessages((prev) => [...prev, { sender: 'bot', text: 'Error:
Could not start the conversation. Please try again.' }]);
      });
  };

  const handleUserInput = (e) => {
    e.preventDefault();
    if (!userInput.trim()) return;

    const newMessages = [...messages, { sender: 'user', text: userInput
}];
    setMessages(newMessages);

    processUserResponse(userInput, newMessages);
    setUserInput('');
  };

  const processUserResponse = (input, updatedMessages) => {
    const step = updatedMessages.filter((msg) => msg.sender === 'bot'
&& !msg.text.startsWith('Invalid input')).length;

    if (step === 1) {
      if (!metaModelsList.includes(input)) {
        setMessages((prev) => [
          ...prev,
          { sender: 'bot', text: 'Invalid meta-model selected. Please
choose from the available options.' },
        ]);
        return;
      }
      startConversation(input);
      return;
    }

    let botResponse = '';
    let validOptions = [];
    let isValid = true;

    switch (step) {
      case 2:
```

```
        validOptions = metaModel?.report_types || [];
        botResponse = `Great choice! What visualization type would you
prefer? Options: ${metaModel.visualization_types.map(vt =>
vt.label).join(', ')}.`;
        break;
      case 3:
        validOptions = metaModel?.visualization_types || [];
        botResponse = 'Do you need any filters applied? (e.g., date
range, departments).';
        break;
      case 4:
        if (input.toLowerCase() !== 'yes' && input.toLowerCase() !==
'no') {
          setMessages((prev) => [
            ...prev,
            { sender: 'bot', text: 'Invalid input. Please respond with
"Yes" or "No".' },
          ]);
          isValid = false;
        } else {
          botResponse = `Any sub-reports needed? Options:
${metaModel.sub_reports.map(sr => sr.label).join(', ')}.`;
        }
        break;
      case 5:
        validOptions = metaModel?.sub_reports || [];
        botResponse = `What data fields do you require? Options:
${metaModel.data_fields.map(df => df.label).join(', ')}.`;
        break;
      case 6:
        validOptions = metaModel?.data_fields || [];
        botResponse = `In which format would you like the report
delivered? Options: ${metaModel.delivery_formats.map(df =>
df.label).join(', ')}.`;
        break;
      case 7:
        validOptions = metaModel?.delivery_formats || [];
        botResponse = `How frequently do you need this report? Options:
${metaModel.frequency.map(f => f.label).join(', ')}.`;
        break;
      case 8:
        validOptions = metaModel?.frequency || [];
```

```
        botResponse = 'Thank you! Type anything in the chat to generate
a Summary.';
        break;
      case 9:
        if (isValid) {
          setValidResponses((prev) => [...prev, input]);
        }
        setMessages((prev) => [...prev, { sender: 'bot', text:
botResponse }]);
        generateSummary();
        setConversationOver(true);
        return;
      default:
        botResponse = 'Invalid input.';
        isValid = false;
    }

    if (!conversationOver && validOptions.length > 0) {
      const validLabels = validOptions.map(option => option.label);
      if (!validLabels.includes(input)) {
        setMessages((prev) => [
          ...prev,
          { sender: 'bot', text: 'Invalid input. Please choose from the
available options.' },
        ]);
        isValid = false;
      }
    }

    if (isValid) {
      setValidResponses((prev) => [...prev, input]);
    }

    if (!conversationOver && isValid) {
      setTimeout(() => {
        setMessages((prevMessages) => [
          ...prevMessages,
          { sender: 'bot', text: botResponse },
        ]);
      }, 500);
    }
  };
```

```
  const generateSummary = () => {
    const summaryText = `
    Report Type: ${validResponses[0] || 'N/A'}
    Visualization: ${validResponses[1] || 'N/A'}
    Filters: ${validResponses[2]?.toLowerCase() === 'yes' ? 'Yes' :
'No'}
    Sub-Reports: ${validResponses[3] || 'N/A'}
    Data Fields: ${validResponses[4] || 'N/A'}
    Delivery Format: ${validResponses[5] || 'N/A'}
    Frequency: ${validResponses[6] || 'N/A'}
    `;

    setSummary(summaryText);

    axios
      .post('http://localhost:5000/api/conversation/save', {
        userResponses: validResponses,
        summary: summaryText,
      })
      .then((res) => console.log(res.data.message))
      .catch((err) => console.error(err));
  };

  return (
    <div className="chat-interface">
      <h2>Report Requirement Chat</h2>
      <div className="chat-window">
        {messages.map((msg, index) => (
          <ChatBubble key={index} message={msg} />
        ))}
      </div>
      {!conversationOver ? (
        <form onSubmit={handleUserInput} className="chat-input">
          <input
            type="text"
            value={userInput}
            onChange={(e) => setUserInput(e.target.value)}
            placeholder="Type your response..."
          />
          <button type="submit">Send</button>
        </form>
```

```
      ) : (
        <Summary summary={summary} />
      )}
    </div>
  );
}

export default ChatInterface;
```

Frontend Meta Model Uploader:

```
// frontend/src/components/MetaModelUploader.js
import React, { useState } from 'react';
import axios from 'axios';
import './uploader.css'

function MetaModelUploader() {
  const [fileContent, setFileContent] = useState('');
  const [email, setEmail] = useState('');
  const [password, setPassword] = useState('');

  const handleFileRead = (e) => {
    const content = e.target.result;
    setFileContent(JSON.parse(content));
  };

  const handleFileChosen = (file) => {
    let fileReader = new FileReader();
    fileReader.onloadend = handleFileRead;
    fileReader.readAsText(file);
  };

  const handleUpload = () => {
    axios
      .post('http://localhost:5000/api/meta-model/upload', {
        fileContent,
        email,
        password,
      })
      .then((res) => alert(res.data.message))
```

```
      .catch((err) => alert(err.response.data.error));
  };

  return (
    <div className="uploader-container">
      {/* Left Section */}
      <div className="left-section">
        <h1>Upload Meta Model</h1>
      </div>

      {/* Right Section (Login Form) */}
      <div className="right-section">
        <div className="meta-model-uploader">
          <h2>Login to Upload</h2>
          <input
            type="text"
            placeholder="Email"
            value={email}
            onChange={(e) => setEmail(e.target.value)}
          />
          <input
            type="password"
            placeholder="Password"
            value={password}
            onChange={(e) => setPassword(e.target.value)}
          />
          <input
            type="file"
            accept=".json"
            onChange={(e) => handleFileChosen(e.target.files[0])}
          />
          <button onClick={handleUpload}>Upload</button>
        </div>
      </div>
    </div>
  );
}

export default MetaModelUploader;
```

# Video about final project:

Here is the link to a video of the project:
https://drive.google.com/file/d/1KIfYE0avGlyhli275KEBCijB5UXre3Dd/view?usp=drive_link

# Project Evolution:

## Phase 1: Initial Submission

During the initial phase of the project, the focus was primarily on conceptualizing and defining the requirements for the conversational requirement elicitation tool. The team developed a Software Requirement Specification (SRS) document, which outlined the system's scope, stakeholders, and expected functionalities. This phase also included the design of a basic meta-model to test communication logic and backend functionality.

Key Achievements:

- Created a foundational meta-model in JSON/XML for initial testing.
- Developed the basic architecture for the frontend, middleware, and backend.
- Implemented preliminary database integration with MongoDB .

## Phase 2: Interim Submission

In the second phase, the focus shifted toward functional development and improving the interaction flow between components. The team implemented a functional frontend using React.js, connected it to the middleware, and ensured real-time communication with the backend and database.

Key Improvements:

- Built core React components such as `ChatInterface` and `ChatBubble`.
- Established API endpoints using Express.js.
- Established middleware processing logic for secure communication.
- Conducted integration testing between the frontend, middleware, and backend.

## Final Submission

The final phase represents the culmination of all previous work, focusing on enhancing the tool for deployment and ensuring production-readiness. This phase included rigorous testing, refinement of conversational logic, and the addition of features to improve user experience and system scalability.

Key Enhancements:

- Refined middleware logic for more efficient handling of complex user queries.
- Improved database schema to optimize query performance and scalability.
- Enhanced UI/UX elements, including dynamic visualizations and responsive design.
- Enhanced API integration using Axios.

- Conducted comprehensive testing to validate the tool's functionality, performance, and reliability.
- Prepared detailed documentation and user training materials for deployment readiness.

# Team member details and contributions:

**Team Number**: 21

The successful completion of this project was a collaborative effort by all team members, with each individual contributing their expertise in specific areas:

Members:

1. **Suraj Lalwani (2024201071):**
   Played a key role in designing and developing the **frontend user interface (UI/UX)**. Ensured the user interface was responsive, intuitive, and visually appealing. Additionally, conducted rigorous **testing** to validate the functionality and usability of the system.

2. **Fraz Uddin (2024201064):**
   Focused on the **backend development**, designing and implementing robust **API endpoints**. Integrated the backend with the **MongoDB database**, ensuring efficient data storage and retrieval to support dynamic user interactions.

3. **Amarnath Kumar (2024202024):**
   Worked on connecting the **frontend UI** with the backend, ensuring seamless communication and data exchange between the two layers. Contributed to the development of frontend components that interacted with the backend APIs.

4. **Abhirup Bhattacharya (2024201015):**
   Led the development of the **middleware**, ensuring secure and efficient communication between the frontend and backend. Designed middleware logic for input validation, authentication, and business rule enforcement.

This collective effort resulted in a well-rounded and robust system, with each team member bringing unique skills to different components of the project. Together, the team ensured the successful execution of the project objectives.

# References:

Here are the references used in the project:

1. **MongoDB Inc. (2023).** Introduction to NoSQL Databases and MongoDB. Retrieved from https://www.mongodb.com

2. **Node.js Foundation. (2023).** Node.js Documentation: Building Scalable Network Applications. Retrieved from https://nodejs.org

3. **Express.js. (2023).** Middleware and Routing with Express.js. Retrieved from https://expressjs.com

4. **React Documentation. (2023).** Building Interactive UIs with React. Retrieved from https://reactjs.org

5. **Axios.js. (2023).** Promise-Based HTTP Client for JavaScript. Retrieved from

6. **Mongoose Documentation. (2023).** Object Data Modeling (ODM) Library for MongoDB and Node.js. Retrieved from https://mongoosejs.com

7. **Fowler, M. (2019).** Patterns of Enterprise Application Architecture. Addison-Wesley.

8. **Shneiderman, B., & Plaisant, C. (2018).** Designing the User Interface: Strategies for Effective Human-Computer Interaction. Pearson.

_____**END**_____