Good morning, everyone.

I'm (Name)

I was the team leader for the EduTutor AI: Personalized Learning with Generative AI and LMS Integration along with my team members {Other Team Member names }

I'm here today to introduce you to

.**EduTutor AI**, a project that aims to bridge the gap between AI-powered learning and traditional Learning Management Systems. Our goal is to empower students and educators by providing an accessible, AI-powered assistant that can answer questions about course material, provide personalized explanations, generate practice problems, and integrate seamlessly with existing LMS platforms.

An explanation of the EduTutor AI project's sudo code, flowchart, and front-end design, along with a description of the Python backend.

Before that we see the system requirement of this project before jump in...

System Requirement

Hardware Requirements

- **GPU:** A dedicated GPU with sufficient VRAM is essential for running the model efficiently. While a CPU-only setup is supported, it would be extremely slow. IBM recommends using a GPU with at least **48 GB of VRAM**, such as an **NVIDIA L40S**, for deploying custom foundation models. The provided backend code uses `torch.bfloat16` for memory efficiency, so a consumer-grade GPU with less VRAM might be able to handle it but with significant performance limitations.
- **CPU:** A modern multi-core processor, like an **Intel Core i5/i7 or AMD Ryzen 5/7**, is recommended for handling data preprocessing and other tasks.
- **RAM:** A minimum of **8 GB RAM** is required for basic tasks, but **16 GB or more** is highly recommended, especially when loading and running large models.
- **Storage:** An **SSD** with at least **256 GB** of storage is recommended for faster data loading, with **512 GB or more** being ideal for large models and datasets.

Software Requirements

- **Operating System:** PyTorch, a key dependency for the model, is compatible with

Windows, **macOS**, and **Linux**. Linux distributions like **Ubuntu** are often preferred for their better CUDA support.
- **Python:** The `transformers` library, used to load the model, requires **Python 3.9+**.
- **Libraries:**
    - **PyTorch**: The backend code relies on PyTorch, which requires a compatible version (e.g., PyTorch 2.1+).
    - **Transformers**: This library, used for the model, requires Python 3.9+.
    - **Werkzeug Security**: This library is used for password hashing.
    - **Flask**: The main web framework for the backend application.
    - **GPU drivers:** If a GPU is used, the appropriate NVIDIA drivers and CUDA Toolkit must be installed.

Sudo Code & Flowchart

The sudo code and flowchart represent the core logic of the chat functionality. They outline the steps from a user submitting a message to receiving a response from the AI.

Pseudocode and Flowchart (Leader's Perspective)

As a project leader, I will explain the core logic of our EduTutor AI system through its pseudocode and flowchart. This will provide a high-level understanding of how our system processes information and makes decisions, without delving into the specific implementation details.

Pseudocode Overview for Edututor

The application, named "AI Quiz Generator," follows a standard user-centric flow, starting with authentication, moving to quiz creation and completion, and ending with result saving and history viewing. The frontend handles user interaction and UI updates, while the backend manages user data, AI model calls, and database operations.

1. The "AI Quiz Generator" application by Edututor uses a pseudocode overview for its user-centric flow, including user authentication, quiz generation, quiz completion, saving results, and viewing historical data.
2. **Frontend (HTML, JavaScript):** Manages user interaction and UI updates.
    - Functions: Initial page loading, user authentication (login/signup), quiz generation from user input, quiz administration, display of historical data, and logout.
3. **Backend (Python, Flask):** Manages user data, AI model calls, and database

operations.
    - Functions: Server initialization, handling /login and /signup endpoints, regulating quiz generation and result management via protected endpoints like /generate_quiz, /save_result, and /quiz_history.
4. **AI Model (ibm-granite):** Loaded upon server startup for quiz generation.
5. **Data Storage:** User data in `users.json`, quiz results in `quiz_results.json`.

Now,
We'll see the sudo code of this project that i created

Sudo Code

```
START

// Initialize Flask app and configuration
SET app = new Flask app
SET app.secret_key = new random key
SET app.config['USER_DB'] = 'users.json'
SET app.config['QUIZ_DB'] = 'quiz_results.json'

// Initialize AI Model
SET MODEL_NAME = "ibm-granite/granite-3.3-2b-instruct"
TRY:
  SET tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)
  SET model = AutoModelForCausalLM.from_pretrained(MODEL_NAME)
  [cite_start]// Check for CUDA availability and move model to GPU if present [cite: 62]
  IF CUDA IS available:
    [cite_start]model.to("cuda") [cite: 62]
  ELSE:
    model.to("cpu")
CATCH Exception AS e:
  [cite_start]// Log error and set model/tokenizer to null [cite: 63]
  [cite_start]PRINT error message [cite: 63]
  SET tokenizer = null
  SET model = null

// Function to generate AI response
FUNCTION generate_ai_response(user_message):
  IF model IS null OR tokenizer IS null:
```

```
    [cite_start]RETURN null, "AI model is not loaded." [cite: 63]

  // Set up the message prompt for the AI
  SET messages = [
    [cite_start]{"role": "system", "content": "You are a quiz-creating assistant. Provide responses in
JSON format only."}, [cite: 64]
    {"role": "user", "content": user_message}
  ]

  TRY:
    // Apply chat template and generate output from the model
    [cite_start]SET inputs = tokenizer.apply_chat_template(messages) [cite: 65]
    [cite_start]SET outputs = model.generate(inputs) [cite: 65]

    [cite_start]// Decode and extract the JSON string from the raw output [cite: 66, 67]
    [cite_start]SET raw_output = tokenizer.decode(outputs[0]) [cite: 66]
    [cite_start]SET json_start = raw_output.find('[') [cite: 67]
    [cite_start]SET json_end = raw_output.rfind(']') [cite: 67]
    IF json_start != -1 AND json_end != -1:
      [cite_start]SET json_string = raw_output from json_start to json_end [cite: 67]
      [cite_start]RETURN JSON.loads(json_string), null [cite: 67]

    // If no valid JSON is found
    RETURN null, "Model response did not contain a valid JSON format."

  CATCH Exception AS e:
    PRINT error message
    RETURN null, "An error occurred during generation."

// Database Helper Functions (file-based)
FUNCTION init_db(db_file):
  IF db_file DOES NOT exist:
    [cite_start]CREATE empty JSON file [cite: 68]
FUNCTION get_data(db_file):
  init_db(db_file)
  RETURN data from JSON file
FUNCTION save_data(db_file, data):
  WRITE data to JSON file
FUNCTION find_user(email):
  ITERATE through users in 'users.json'
  IF user['email'] == email:
    RETURN user
  RETURN null
FUNCTION register_user(email, password):
  IF find_user(email):
```

```
    [cite_start]RETURN false, "Email already registered" [cite: 69]
  users = get_data('users.json')
  [cite_start]APPEND new user with hashed password to users list [cite: 69]
  save_data('users.json', users)
  RETURN true, ""
FUNCTION verify_user(email, password):
  user = find_user(email)
  IF user IS null:
    RETURN false, "User not found"
  IF check_password_hash(user['password'], password) IS false:
    [cite_start]RETURN false, "Incorrect password" [cite: 70]
  RETURN true, user
FUNCTION save_quiz_result(email, topic, score, total):
  results = get_data('quiz_results.json')
  [cite_start]APPEND new result with user email and quiz data to results list [cite: 70]
  save_data('quiz_results.json', results)
FUNCTION get_quiz_history(email):
  results = get_data('quiz_results.json')
  [cite_start]RETURN filtered list where result['user_email'] == email [cite: 71]

// Routes
ON POST to '/login':
  // Get email and password from request body
  success, result = verify_user(email, password)
  IF success:
    SET session['user']['email'] = email
    RETURN success JSON response
  ELSE:
    [cite_start]RETURN error JSON response, status 401 [cite: 72]
ON POST to '/signup':
  // Get email and password from request body
  [cite_start]success, message = register_user(email, password) [cite: 72]
  IF success:
    SET session['user']['email'] = email
    RETURN success JSON response
  ELSE:
    [cite_start]RETURN error JSON response, status 400 [cite: 72]
ON POST to '/logout':
  REMOVE 'user' from session
  RETURN success JSON response
ON POST to '/generate_quiz':
  IF 'user' NOT IN session:
    [cite_start]RETURN "Not logged in" error [cite: 76]
  // Get topic and num_questions from request body
  [cite_start]prompt = "Generate X multiple-choice questions about Y." [cite: 74, 75]
```

```
  questions, error = generate_ai_response(prompt)
  IF questions IS NOT null:
    RETURN JSON response with questions
  ELSE:
    [cite_start]RETURN error JSON response, status 500 [cite: 76]
ON POST to '/save_result':
  IF 'user' NOT IN session:
    [cite_start]RETURN "Not logged in" error [cite: 76]
  // Get topic, score, total from request body
  email = session['user']['email']
  save_quiz_result(email, topic, score, total)
  RETURN success JSON response
ON GET to '/quiz_history':
  IF 'user' NOT IN session:
    [cite_start]RETURN "Not logged in" error [cite: 77]
  email = session['user']['email']
  history = get_quiz_history(email)
  RETURN JSON response with history

// Run the app
RUN app on port 5001 with debug=True

END
```

Flowchart

The flowchart visually represents the same logic as the sudo code. It starts with the user input, moves through a decision block to check if the input is valid, and then proceeds with the UI updates, API call, and response handling. Error handling and retry logic are also included in the flow, showing a clear path for when the API call fails.

Start

|

|--Check for existing user session on page load (checkLogin function)

| |

| |--Is user logged in?

| | |

| | |--Yes: Call onLoginSuccess()

| | | |--Hide login/signup forms

| | | |--Show navigation bar

| | | |--Show "Generate Quiz" section

| | |

| |--No: Show "Auth Container" (login/signup forms)

|

|--User chooses to Log In

| |

| |--Submit login form

| | |

| | |--Send POST request to '/login' with email and password

| | |--Server receives request

| | | |--Finds user in `users.json`

| | | |--Checks password hash

| | | |--Success: Stores user email in session, returns 'success'

| | | |--Fail: Returns 'error' message

| | |

| |--If successful, call onLoginSuccess()

|

|--User chooses to Sign Up

| |

| |--Submit signup form

| | |

| | |--Send POST request to '/signup' with email and password

| | |--Server receives request

| | | |--Checks if email is already registered in `users.json`

| | | |--Success: Hashes password, saves new user, stores email in session, returns 'success'

| | | |--Fail: Returns 'error' message

| | |

| |--If successful, call onLoginSuccess()

|

|--Logged-in user wants to Generate Quiz

| |

| |--Click "Generate Quiz" nav button

| |--Show "Quiz Generator" section

| |--Submit quiz generation form with topic and number of questions

| | |

| | |--Send POST request to '/generate_quiz' with topic and number

| | |--Server receives request

| | | |--Calls `generate_ai_response` function

| | | |--Constructs a prompt for the AI model

| | | |--Feeds prompt to the `ibm-granite` model

| | | |--Extracts JSON quiz data from AI response

| | | |--Success: Returns JSON array of questions

| | | |--Fail: Returns 'error' message

| | |

| |--If successful, call renderQuiz() and show "Quiz Container" section

|

|--User takes the quiz

| |

| |--Answer questions

| |--Click "Submit Answers"

| | |

| | |--Frontend calculates score by comparing user answers to correct answers (stored locally)

| | |--Displays score to user

| | |--Send POST request to '/save_result' with topic, score, and total

| | |

| | |--Server receives request

| | | |--Finds user email from session

| | | |--Saves result to `quiz_results.json`

| | | |--Returns 'success'

|

|--Logged-in user wants to view History

| |

| |--Click "Quiz History" nav button

| |--Call loadHistory()

| | |

| | |--Send GET request to '/quiz_history'

| | |--Server receives request

| | | |--Finds user email from session

| | | |--Filters `quiz_results.json` for that user's history

| | | |--Returns a JSON array of past quiz results

| | |

| |--Frontend receives data and renders a list of historical quiz results

|

|--User logs out

| |

| |--Click "Logout" nav button

| |--Send POST request to '/logout'

| |--Server receives request

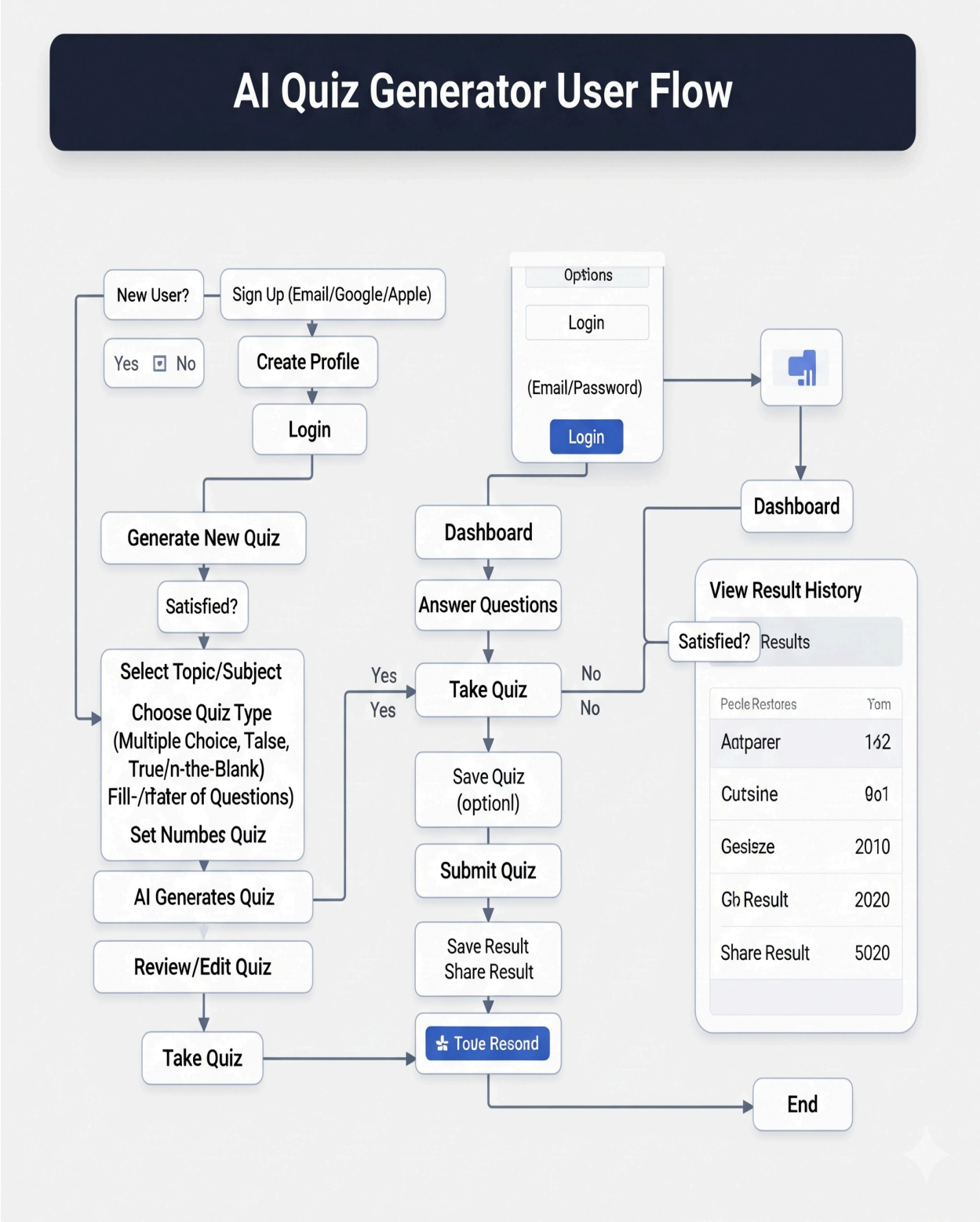
| | |

| | |--Removes user data from session

| |--Reload the page, returning the user to the initial login state

|

End


AI Quiz Generator User Flow
New User?
Sign Up (Email/Google/Apple)
Yes
No
Create Profile
Login
Generate New Quiz
Satisfied?
Select Topic/Subject
Choose Quiz Type
(Multiple Choice, Talse,
True/n-the-Blank)
Fill-/rater of Questions)
Set Numbes Quiz
AI Generates Quiz
Review/Edit Quiz
Take Quiz
Options
Login
(Email/Password)
Login
Dashboard
Answer Questions
Yes
Yes
Take Quiz
No
No
Save Quiz
(optionl)
Submit Quiz
Save Result
Share Result
Toue Resond
Dashboard
View Result History
Satisfied?
Results
Pecle Restores
Aatparer
Cutsine
Gesieze
Gb Result
Share Result
End

Python Backend Explanation

Good morning,
I'm {Back End developer name} of the **EduTutor AI: Personalized Learning with Generative AI and LMS Integration**

Today, I'm excited to unveil the backend architecture of EduTutor AI, a project uniquely positioned at the convergence of education and responsible AI. While the user-facing application offers an intuitive and seamless learning experience, my presentation will focus on the powerful and secure engine driving it: the robust backend system we've developed using Python and the Flask framework.

As the back-end user, you are responsible for the server-side logic that handles the communication between the front end and the AI model. Here's what the Python code would handle:

- The AI Quiz Generator project's backend is a Flask application utilizing file-based databases and a pre-trained AI model for server-side logic, including user authentication, quiz generation, and data storage.
- **Core Components:**
    - **Flask Framework:** The backend is built with Flask, managing routes and handling HTTP requests. `app.secret_key` is set for secure session management.
    - **AI Model Integration:** It uses the `ibm-granite/granite-3.3-2b-instruct` model from Hugging Face for quiz generation.
        - The model and tokenizer load into memory upon server start, detecting and utilizing CUDA-enabled GPUs for performance.
        - The `generate_ai_response` function interfaces with the model, formatting user requests into structured prompts and extracting valid JSON data from the AI's output.
    - **File-Based Databases:** Two JSON files persist data:
        - `users.json`: Stores user information (email, hashed password) using `werkzeug.security` for password hashing and verification.
        - `quiz_results.json`: Stores quiz history (user email, topic, score, total questions, date).
- **Backend Routes (API Endpoints):**
    - `/login` (POST): Authenticates users against `users.json` and creates a session.
    - `/signup` (POST): Registers new users, checking for existing emails before hashing passwords and saving details to `users.json`.

    - `/logout` (POST): Clears the current user's session.
    - `/generate_quiz` (POST): Takes a topic and question count, uses the AI model to generate a quiz, and returns it as JSON. Requires user login.
    - `/save_result` (POST): Receives quiz score and topic, saving the result to `quiz_results.json` using the user's session email. Requires user login.
    - `/quiz_history` (GET): Retrieves the quiz history for the logged-in user by filtering `quiz_results.json`. Protected endpoint.

- **AI Model Initialization**
    - Uses **ibm-granite/granite-3.3-2b-instruct** model loaded directly into server memory, not an external API.
    - **Hardware Check**: Prioritizes GPU (using `torch.cuda.is_available()` and `.to("cuda")`) for faster inference; defaults to CPU if no GPU.
    - **Tokenizer**: Loads `AutoTokenizer.from_pretrained()` to convert text into numerical tokens.
    - **Model Loading**: Loads `AutoModelForCausalLM.from_pretrained()` with `torch_dtype=torch.bfloat16` for memory efficiency.
    - **generate_ai_response(user_message)**: Core AI integration function.
        - Takes user prompt, wraps it in a structured conversation format with a system prompt (acting as a "helpful and responsible AI assistant that specializes in creating quizzes" and responding "in a structured JSON format only").
        - Extracts valid JSON string containing quiz questions and options from raw model output, handling extra conversational text.
- **User DB Helpers**
    - Manages user accounts and data storage using simple JSON files as a lightweight database.
    - **File Initialization**: `init_db(db_file)` ensures `users.json` and `quiz_results.json` exist; creates empty JSON arrays if files are not found.
    - **Read/Write Operations**: `get_data(db_file)` and `save_data(db_file, data)` handle reading from and writing to JSON files.
    - **User Management**:
        - `find_user(email)`: Locates a user by email in `users.json`.
        - `register_user(email, password)`: Creates new user accounts.
            - Checks if email is already in use.
            - Hashes password using `werkzeug.security.generate_password_hash` for security.
        - `verify_user(email, password)`: Authenticates user by comparing provided password with stored hashed password using

`werkzeug.security.check_password_hash`.

- **Routes**
  - Flask application routes for various functionalities and API endpoints.
  - `/`: Renders the main `index.html` page.
  - `/login` (**POST**): Processes login attempts, calls `verify_user()`, and adds user's email to Flask session upon success.
  - `/signup` (**POST**): Handles new user registration, calls `register_user()`, and creates a new session upon success.
  - `/logout` (**POST**): Removes user's information from the session, logging them out.
  - `/generate_quiz` (**POST**): Protected API endpoint (requires login). Takes `topic` and `num_questions`, uses `generate_ai_response()` to create the quiz.
  - `/save_result` (**POST**): Protected endpoint for saving quiz scores. Retrieves logged-in user's email and saves results to `quiz_results.json`.
  - `/quiz_history` (**GET**): Protected endpoint that fetches and returns a user's past quiz results, filtering data from `quiz_results.json` for the current user.

Python program

```
from flask import Flask, render_template, request, session, jsonify
import os, json, secrets
from datetime import datetime
from werkzeug.security import generate_password_hash, check_password_hash
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM

app = Flask(__name__)
app.secret_key = secrets.token_hex(16)
app.config['USER_DB'] = 'users.json'
app.config['QUIZ_DB'] = 'quiz_results.json'

# ----------------------
# AI Model Initialization
# ----------------------
```

```
MODEL_NAME = "ibm-granite/granite-3.3-2b-instruct"
print(f"Loading model: {MODEL_NAME}...")

try:
    tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)
    model = AutoModelForCausalLM.from_pretrained(
        MODEL_NAME,
        torch_dtype=torch.bfloat16 if torch.cuda.is_available() else
torch.float32,
        device_map="auto" if torch.cuda.is_available() else None
    )
    if torch.cuda.is_available():
        model.to("cuda")
    else:
        model.to("cpu")
    print("Model loaded successfully.")

except Exception as e:
    print(f"Error loading model: {e}")
    tokenizer = None
    model = None


def generate_ai_response(user_message):
    """Generate AI chat response using the loaded Hugging Face model."""
    if not model or not tokenizer:
        return None, "AI model is not loaded. Please check the server
logs."

    messages = [
        {"role": "system", "content": "You are a helpful and responsible
AI assistant that specializes in creating quizzes. You will provide
responses in a structured JSON format only."},
        {"role": "user", "content": user_message}
    ]

    try:
```

```
        inputs = tokenizer.apply_chat_template(
            messages,
            add_generation_prompt=True,
            tokenize=True,
            return_tensors="pt",
            thinking=False
        ).to(model.device)

        outputs = model.generate(
            **inputs,
            max_new_tokens=512,
            temperature=0.7,
            top_p=0.9
        )

        raw_output = tokenizer.decode(
            outputs[0][inputs["input_ids"].shape[-1]:],
            skip_special_tokens=True
        )

        # The model might include some conversational text outside the
JSON.
        # We need to find and extract the valid JSON part.
        json_start = raw_output.find('[')
        json_end = raw_output.rfind(']')
        if json_start != -1 and json_end != -1:
            json_string = raw_output[json_start : json_end + 1]
            return json.loads(json_string), None

        return None, "Model response did not contain a valid JSON format."

    except Exception as e:
        print(f"Error during AI generation: {e}")
        return None, f"An error occurred while generating questions: {e}"


# ----------------------
```

```python
# User DB Helpers (file-based)
# ----------------------
def init_db(db_file):
    if not os.path.exists(db_file):
        with open(db_file, 'w') as f:
            json.dump([], f)

def get_data(db_file):
    init_db(db_file)
    with open(db_file, 'r') as f:
        return json.load(f)

def save_data(db_file, data):
    with open(db_file, 'w') as f:
        json.dump(data, f, indent=2)

def find_user(email):
    for user in get_data(app.config['USER_DB']):
        if user['email'] == email:
            return user
    return None

def register_user(email, password):
    if find_user(email):
        return False, "Email already registered"
    users = get_data(app.config['USER_DB'])
    users.append({
        'email': email,
        'password': generate_password_hash(password),
        'created_at': datetime.now().isoformat()
    })
    save_data(app.config['USER_DB'], users)
    return True, ""

def verify_user(email, password):
    user = find_user(email)
    if not user:
```

```
        return False, "User not found"
    if not check_password_hash(user['password'], password):
        return False, "Incorrect password"
    return True, user

def save_quiz_result(email, topic, score, total):
    results = get_data(app.config['QUIZ_DB'])
    results.append({
        'user_email': email,
        'topic': topic,
        'score': score,
        'total': total,
        'date': datetime.now().isoformat()
    })
    save_data(app.config['QUIZ_DB'], results)

def get_quiz_history(email):
    results = get_data(app.config['QUIZ_DB'])
    return [r for r in results if r['user_email'] == email]

# ----------------------
# Routes
# ----------------------
@app.route('/')
def home():
    return render_template('index.html')

@app.route('/login', methods=['POST'])
def login():
    data = request.json
    email = data.get('email')
    password = data.get('password')
    success, result = verify_user(email, password)
    if success:
        session['user'] = {'email': email}
        return jsonify({'status': 'success', 'user': {'email': email}})
    else:
```

```
        return jsonify({'status': 'error', 'message': result}), 401

@app.route('/signup', methods=['POST'])
def signup():
    data = request.json
    email = data.get('email')
    password = data.get('password')
    success, message = register_user(email, password)
    if success:
        session['user'] = {'email': email}
        return jsonify({'status': 'success', 'user': {'email': email}})
    else:
        return jsonify({'status': 'error', 'message': message}), 400

@app.route('/logout', methods=['POST'])
def logout():
    session.pop('user', None)
    return jsonify({'status': 'success'})

@app.route('/generate_quiz', methods=['POST'])
def generate_quiz():
    if 'user' not in session:
        return jsonify({"status": "error", "message": "Not logged in"}),
401

    data = request.json
    topic = data.get("topic", "General Knowledge")
    num_questions = data.get("num_questions", 5)

    prompt = f"Generate {num_questions} multiple-choice quiz questions
about {topic}. Each question should have 4 options. Indicate the correct
answer. Return the response as a valid JSON array. Do not include any text
outside of the JSON. The JSON should follow this structure: [{{'question':
'Your question here', 'options': ['Option 1', 'Option 2', 'Option 3',
'Option 4'], 'answer': 2}}] The 'answer' is the zero-based index of the
correct option."
```

```
    questions, error = generate_ai_response(prompt)
    if questions:
        return jsonify({"status": "success", "questions": questions})
    else:
        return jsonify({"status": "error", "message": error}), 500

@app.route('/save_result', methods=['POST'])
def save_result():
    if 'user' not in session:
        return jsonify({"status": "error", "message": "Not logged in"}),
401

    data = request.json
    email = session['user']['email']
    topic = data.get('topic')
    score = data.get('score')
    total = data.get('total')

    if not all([topic, score, total]):
        return jsonify({"status": "error", "message": "Missing quiz
data"}), 400

    save_quiz_result(email, topic, score, total)
    return jsonify({"status": "success"})

@app.route('/quiz_history', methods=['GET'])
def quiz_history():
    if 'user' not in session:
        return jsonify({"status": "error", "message": "Not logged in"}),
401

    email = session['user']['email']
    history = get_quiz_history(email)
    return jsonify({"status": "success", "history": history})

if __name__ == '__main__':
    app.run(debug=True, port=5001)
```

Front-End Design

Good morning,

I'm **{FrpntEnd Developer Name** } of the project **Citizen AI – Intelligent Citizen Engagement Platform**

I'm here today to introduce you to the front-end design of EduTutor AI, our project dedicated to making educational resources more accessible. As a front-end designer, my primary focus was on creating a user-centric interface that is not only visually appealing but also intuitive and functional. Our design philosophy was simple: to build a platform that feels welcoming and easy to navigate for anyone seeking help with their studies.

The front end is the user interface. It is built using **HTML** for structure, **Tailwind CSS** for styling, and **JavaScript** for interactivity.

Here's a breakdown of the HTML structure and JavaScript logic:**HTML Structure**

- **Header**: Displays the title "AI Quiz Generator."

```
  <header class="bg-white shadow-md py-6">

    <h1 class="text-4xl font-extrabold text-center text-blue-700
tracking-wide select-none">AI Quiz Generator</h1>

  </header>
```

- **Navigation Bar**: Contains "Generate Quiz," "Quiz History," and "Logout" buttons, hidden until the user logs in.

```
  <nav id="nav" class="hidden bg-blue-600 text-white flex
justify-center space-x-6 py-4 shadow-md sticky top-0 z-50">

    <button id="nav-generate" class="hover:bg-blue-500 px-4 py-2
rounded-md font-semibold transition">Generate Quiz</button>

    <button id="nav-history" class="hover:bg-blue-500 px-4 py-2
rounded-md font-semibold transition">Quiz History</button>

    <button id="nav-logout" class="hover:bg-red-500 px-4 py-2
```

```
rounded-md font-semibold transition">Logout</button>

  </nav>
```

- **Authentication Container (`auth-container`)**: Initial view for new or logged-out users, containing login and signup forms.

```
<section id="auth-container" class="bg-white rounded-lg shadow-lg
p-8 max-w-md mx-auto">

      <!-- Login -->

      <h2 class="text-2xl font-bold text-center text-blue-700
mb-6">Login</h2>

      <form id="login-form" class="space-y-5">

        <div>

          <label for="login-email" class="block text-sm font-medium
text-gray-700">Email</label>

          <input type="email" id="login-email" required class="mt-1
block w-full rounded-md border border-gray-300 shadow-sm p-2
focus:ring-blue-500 focus:border-blue-500" />

        </div>

        <div>

          <label for="login-password" class="block text-sm
font-medium text-gray-700">Password</label>

          <input type="password" id="login-password" required
class="mt-1 block w-full rounded-md border border-gray-300 shadow-sm
p-2 focus:ring-blue-500 focus:border-blue-500" />

        </div>

        <button type="submit" class="w-full bg-blue-600
hover:bg-blue-700 text-white font-semibold py-2 rounded-md
transition">Login</button>

        <p class="text-center text-sm text-gray-600 mt-3">
```

```
          Don't have an account?

          <a href="#" id="show-signup" class="text-blue-600
hover:underline font-semibold cursor-pointer">Sign up here</a>

        </p>

        <p id="login-error" class="text-red-600 text-center
mt-2"></p>

      </form>



      <!-- Signup -->

      <h2 id="signup-title" class="text-2xl font-bold text-center
text-blue-700 mb-6 mt-10 hidden">Sign Up</h2>

      <form id="signup-form" class="space-y-5 hidden">

        <div>

          <label for="signup-email" class="block text-sm font-medium
text-gray-700">Email</label>

          <input type="email" id="signup-email" required class="mt-1
block w-full rounded-md border border-gray-300 shadow-sm p-2
focus:ring-blue-500 focus:border-blue-500" />

        </div>

        <div>

          <label for="signup-password" class="block text-sm
font-medium text-gray-700">Password</label>

          <input type="password" id="signup-password" required
class="mt-1 block w-full rounded-md border border-gray-300 shadow-sm
p-2 focus:ring-blue-500 focus:border-blue-500" />

        </div>

        <button type="submit" class="w-full bg-blue-600
hover:bg-blue-700 text-white font-semibold py-2 rounded-md
transition">Sign Up</button>
```

```
        <p class="text-center text-sm text-gray-600 mt-3">

          Already have an account?

          <a href="#" id="show-login" class="text-blue-600
hover:underline font-semibold cursor-pointer">Login here</a>

        </p>

        <p id="signup-error" class="text-red-600 text-center
mt-2"></p>

      </form>

    </section>
```

- **Quiz Generator (`quiz-generator`)**: Form for logged-in users to input a topic and desired number of questions.

```
<section id="quiz-generator" class="hidden bg-white rounded-lg
shadow-lg p-8 max-w-md mx-auto">

      <h2 class="text-2xl font-bold text-center text-blue-700
mb-6">Generate a Quiz</h2>

      <form id="generate-form" class="space-y-5">

        <div>

          <label for="topic" class="block text-sm font-medium
text-gray-700">Topic</label>

          <input type="text" id="topic" placeholder="e.g. Science,
History" required class="mt-1 block w-full rounded-md border
border-gray-300 shadow-sm p-2 focus:ring-blue-500
focus:border-blue-500" />

        </div>

        <div>

          <label for="num-questions" class="block text-sm
```

```
font-medium text-gray-700">Number of Questions</label>

          <input type="number" id="num-questions" min="1" max="20"
value="5" required class="mt-1 block w-full rounded-md border
border-gray-300 shadow-sm p-2 focus:ring-blue-500
focus:border-blue-500" />

        </div>

        <button type="submit" class="w-full bg-blue-600
hover:bg-blue-700 text-white font-semibold py-2 rounded-md
transition">Generate Quiz</button>

        <p id="generate-error" class="text-red-600 text-center
mt-2"></p>

      </form>

    </section>
```

- **Quiz Container (`quiz-container`)**: Dynamically displays generated quiz questions and options.

```
    <section id="quiz-container" class="hidden bg-white rounded-lg
shadow-lg p-8 max-w-3xl mx-auto">

      <h2 class="text-2xl font-bold text-center text-blue-700
mb-6">Take the Quiz</h2>

      <form id="quiz-form" class="space-y-8">

        <div id="questions-list" class="space-y-6 max-h-[60vh]
overflow-y-auto pr-4"></div>

        <button type="submit" class="w-full bg-green-600
hover:bg-green-700 text-white font-semibold py-3 rounded-md
transition">Submit Answers</button>
```

```
        <p id="quiz-error" class="text-red-600 text-center
mt-2"></p>

      </form>

      <div id="quiz-result" class="mt-6 text-center text-lg
font-semibold text-green-700"></div>

    </section>
```

- **History Container (`history-container`)**: Displays a list of past quiz scores and topics.

```
    <section id="history-container" class="hidden bg-white
rounded-lg shadow-lg p-8 max-w-3xl mx-auto">

      <h2 class="text-2xl font-bold text-center text-blue-700
mb-6">Your Quiz History</h2>

      <ul id="history-list" class="divide-y divide-gray-200
max-h-[60vh] overflow-y-auto"></ul>

    </section>
```

- **Footer**: Contains the copyright notice.

```
  <footer class="bg-white text-center py-4 text-gray-500
select-none">

    &copy; 2024 AI Quiz Generator. All rights reserved.

  </footer>
```

JavaScript Logic

- **Element Selection**: Variables reference specific HTML elements (forms, buttons, display areas).

```
const authContainer = document.getElementById('auth-container');

const loginForm = document.getElementById('login-form');

const signupForm = document.getElementById('signup-form');

const loginError = document.getElementById('login-error');

const signupError = document.getElementById('signup-error');

const showSignupLink = document.getElementById('show-signup');

const showLoginLink = document.getElementById('show-login');

const signupTitle = document.getElementById('signup-title');



const nav = document.getElementById('nav');

const navGenerate = document.getElementById('nav-generate');

const navHistory = document.getElementById('nav-history');

const navLogout = document.getElementById('nav-logout');



const quizGenerator = document.getElementById('quiz-generator');

const generateForm = document.getElementById('generate-form');

const generateError = document.getElementById('generate-error');
```

```
    const quizContainer = document.getElementById('quiz-container');

    const questionsList = document.getElementById('questions-list');

    const quizForm = document.getElementById('quiz-form');

    const quizError = document.getElementById('quiz-error');

    const quizResult = document.getElementById('quiz-result');

    const historyContainer =
document.getElementById('history-container');

    const historyList = document.getElementById('history-list');

    let currentQuestions = [];

    let currentTopic = '';
```

- **User Authentication**:
  - Event listeners handle login and signup form submissions.
  - Click events on "Sign up here" and "Login here" links toggle form visibility.
  - Form submissions send asynchronous fetch requests to `/login` or `/signup` endpoints.
  - `onLoginSuccess()` hides the authentication container and displays navigation/quiz generation upon successful login/signup.

```
  function onLoginSuccess() {

     authContainer.classList.add('hidden');

     nav.classList.remove('hidden');

     quizGenerator.classList.remove('hidden');

     signupForm.classList.add('hidden');

     signupTitle.classList.add('hidden');

     loginForm.classList.remove('hidden');
```

```
      loginError.textContent = '';
      signupError.textContent = '';
      window.scrollTo({top: 0, behavior: 'smooth'});
    }

    // On page load, check if user is logged in by trying to fetch quiz history
    async function checkLogin() {
      try {
        const res = await fetch('/quiz_history');
        if (res.status === 200) {
          const data = await res.json();
          if (data.status === 'success') {
            onLoginSuccess();
          }
        }
      } catch {
        // Not logged in or server error, show login form
      }
    }

    checkLogin();
```

- **Navigation and View Management**:
  - `showSection(section)` controls which main page section is visible.
  - Navigation buttons (`nav-generate`, `nav-history`, `nav-logout`) use `addEventListener` to change views or log out.

```
function showSection(section) {

      authContainer.classList.add('hidden');

      nav.classList.remove('hidden');

      quizGenerator.classList.add('hidden');

      quizContainer.classList.add('hidden');

      historyContainer.classList.add('hidden');



      if (section === 'generate') {

        quizGenerator.classList.remove('hidden');

      } else if (section === 'quiz') {

        quizContainer.classList.remove('hidden');

      } else if (section === 'history') {

        historyContainer.classList.remove('hidden');

      }

    }
```

- **Quiz Generation & Display**:
  - Quiz generation form submission sends a POST request to `/generate_quiz` with topic and number of questions.
  - `renderQuiz(questions)` dynamically creates HTML for questions and options from backend JSON data.

```
  generateForm.addEventListener('submit', async e => {

      e.preventDefault();

      generateError.textContent = '';

      quizResult.textContent = '';

      questionsList.innerHTML = '';

      quizContainer.classList.add('hidden');



      const topic =
document.getElementById('topic').value.trim();

      const numQuestions =
parseInt(document.getElementById('num-questions').value);



      if (!topic || numQuestions < 1) {

        generateError.textContent = 'Please enter a valid
topic and number of questions.';

        return;

      }



      try {

        const res = await fetch('/generate_quiz', {
```

```
          method: 'POST',
          headers: {'Content-Type': 'application/json'},
          body: JSON.stringify({topic, num_questions:
numQuestions})
        });
        const data = await res.json();
        if (data.status === 'success') {
          currentQuestions = data.questions;
          currentTopic = topic;
          renderQuiz(currentQuestions);
          showSection('quiz');
          window.scrollTo({top: 0, behavior: 'smooth'});
        } else {
          generateError.textContent = data.message || 'Failed
to generate quiz.';
        }
      } catch (err) {
        generateError.textContent = 'Server error. Please try
again later.';
      }
    });

    // Render quiz questions
```

```
    function renderQuiz(questions) {
      questionsList.innerHTML = '';
      questions.forEach((q, i) => {
        const div = document.createElement('div');
        div.className = 'quiz-question bg-blue-50 p-4
rounded-md shadow-sm';
        div.innerHTML = `
          <p class="font-semibold text-blue-800 mb-3">Q${i +
1}: ${q.question}</p>
          <div class="space-y-2">
            ${q.options.map((opt, idx) => `
              <label class="flex items-center space-x-3
cursor-pointer hover:bg-blue-100 rounded-md p-2 transition">
                <input type="radio" name="q${i}"
value="${idx}" required class="form-radio text-blue-600" />
                <span class="text-gray-800">${opt}</span>
              </label>
            `).join('')}
          </div>
        `;
        questionsList.appendChild(div);
      });
      quizError.textContent = '';
```

```
      quizResult.textContent = '';

    }
```

- **Quiz Submission**:
  - The `quizForm` submit handler calculates the user's score by comparing selected answers to `currentQuestions` array.
  - A POST request is sent to `/save_result` to save the score to the backend.

```
quizForm.addEventListener('submit', async e => {

      e.preventDefault();

      quizError.textContent = '';

      quizResult.textContent = '';



      const formData = new FormData(quizForm);

      let score = 0;

      for (let i = 0; i < currentQuestions.length; i++) {

        const answer = formData.get(`q${i}`);

        if (answer === null) {

          quizError.textContent = 'Please answer all
questions.';

          return;

        }

        if (parseInt(answer) === currentQuestions[i].answer) {

          score++;
```

```
        }
      }

      quizResult.textContent = `You scored ${score} out of
${currentQuestions.length}. Saving result...`;

      try {
        const res = await fetch('/save_result', {
          method: 'POST',
          headers: {'Content-Type': 'application/json'},
          body: JSON.stringify({
            topic: currentTopic,
            score: score,
            total: currentQuestions.length
          })
        });
        const data = await res.json();
        if (data.status === 'success') {
          quizResult.textContent += ' Result saved!';
        } else {
          quizResult.textContent += ' But failed to save
result.';
```

```
        }

      } catch (err) {

        quizResult.textContent += ' But server error occurred
while saving.';

      }

    });
```

- **Quiz History**:
  - `loadHistory()` makes a GET request to `/quiz_history`.
  - Dynamically populates an unordered list (`<ul>`) with past quiz scores.

```
 // Load quiz history

    async function loadHistory() {

      historyList.innerHTML = '<li class="text-center py-4
text-gray-500">Loading...</li>';

      try {

        const res = await fetch('/quiz_history');

        const data = await res.json();

        if (data.status === 'success') {

          if (data.history.length === 0) {

            historyList.innerHTML = '<li class="text-center
py-4 text-gray-500">No quiz history found.</li>';

          } else {

            historyList.innerHTML = '';

            data.history.forEach(item => {
```

```
            const li = document.createElement('li');

            const date = new
Date(item.date).toLocaleString();

            li.className = 'py-4 px-6 hover:bg-blue-50
rounded-md cursor-default select-text';

            li.textContent = `${date} — Topic: ${item.topic}
— Score: ${item.score} / ${item.total}`;

            historyList.appendChild(li);

          });

        }

      } else {

        historyList.innerHTML = '<li class="text-center py-4
text-red-600">Failed to load history.</li>';

      }

    } catch (err) {

      historyList.innerHTML = '<li class="text-center py-4
text-red-600">Server error while loading history.</li>';

    }

  }
```

- **Page State Management**:
  - `checkLogin()` attempts to fetch quiz history on page load to determine if a user is logged in.
  - If successful, `onLoginSuccess()` is called to show the main application interface.

```
// Show sections based on state

function showSection(section) {

  authContainer.classList.add('hidden');

  nav.classList.remove('hidden');

  quizGenerator.classList.add('hidden');

  quizContainer.classList.add('hidden');

  historyContainer.classList.add('hidden');



  if (section === 'generate') {

    quizGenerator.classList.remove('hidden');

  } else if (section === 'quiz') {

    quizContainer.classList.remove('hidden');

  } else if (section === 'history') {

    historyContainer.classList.remove('hidden');

  }

}
```

Testing

Good morning.

I’m {testing developer name} of the EduTutor AI: Personalized Learning with Generative AI and LMS Integration

I'm here today to discuss the testing and quality assurance of **edututor**, a project where reliability and accuracy are paramount. My role as the testing lead is to ensure that our platform is robust, bug-free, and, most importantly, provides correct and helpful information to our users. We need to be confident that edututor is a trustworthy resource .

Our testing strategy covers the entire application, from the user-facing front end to the backend systems and the core AI model itself. We have a multi-faceted approach that includes:

- **User Interface (UI) and User Experience (UX) Testing:** We conduct rigorous testing to ensure the front end is responsive, all buttons and links are functional, and the chat flow is intuitive and seamless. This includes checking for bugs in the dynamic message display, the loading states, and the feedback forms.
- **Backend and API Testing:** We test all API endpoints to ensure they are secure and reliable. This includes verifying that messages are sent and received correctly, user authentication works as expected, and feedback data is logged properly. We also stress-test the system to ensure it can handle multiple users simultaneously without performance degradation.
- **AI Model Validation:** This is a crucial part of our process. We perform extensive validation of the AI model's responses to ensure the information provided is accurate, unbiased, and safe. We test the model against a wide range of civic queries, including complex legal and benefits-related questions, to identify any inaccuracies or potential for harmful outputs.

Throughout this presentation, I'll provide insights into our testing methodologies, the key metrics we track, and the steps we take to ensure that EdututorAI is a reliable and safe tool for public use.

Thank you.