List of figures

Chapter I: Introduction

The Introduction chapter should provide an overview of the MCQ Generator project and its importance in the context of educational technology. It should highlight the key objectives of the project, which are to develop a web-based platform that leverages the Google Gemini API to generate customizable multiple-choice questions across various subject domains.

The introduction should emphasize how the MCQ Generator addresses the growing need for efficient tools that can create diverse assessment materials for educators, trainers, and educational institutions. It should explain how the project aims to simplify the question creation process by harnessing the power of artificial intelligence and modern web technologies.

Additionally, the introduction should touch upon the potential impact of AI-powered question generation on improving personalized learning experiences and reducing the time spent on manual question creation. This sets the stage for the reader to understand the project's core purpose and the value it aims to deliver.

Chapter II: Literature Review

The Literature Review chapter should explore the key themes and research related to the technological foundations, user-centric design, and the impact of AI-powered question generation tools in the educational technology landscape.

The technological foundations section should delve into the core technologies used in the MCQ Generator project, including Python as the backend programming language, HTML and CSS for the frontend development, and the integration of the Google Gemini API for intelligent question generation. This section should also discuss the required libraries and frameworks, such as Pandas and ReportLab for PDF export functionality, and how they contribute to the overall system architecture.

The user-centric design aspect of the literature review should highlight the importance of creating an intuitive and responsive user interface that simplifies the question generation workflow. It should discuss design principles and best practices aimed at ensuring a clean, modern, and accessible user experience for educators and learners.

Finally, the literature review should examine the growing recognition of AI-powered question generation tools in the educational technology field. It should discuss how these tools can reduce the time and effort required for manual question creation, provide diverse and context-aware questions, and support personalized learning experiences. This section should reference relevant research papers and industry trends to substantiate the project's potential impact.

Chapter III: Feasibility Study

The Feasibility Study chapter should assess the viability of the MCQ Generator project from various perspectives, including technical, economic, operational, and legal/ethical considerations.

The technical feasibility section should evaluate the project's technology stack, including the compatibility and scalability of the chosen technologies (Python, HTML, CSS, JavaScript, Google Gemini API). It should also address any potential challenges or limitations related to API integration, required libraries, and the ability to handle increased user demand and data volume.

The economic feasibility aspect should provide a cost analysis of the project, considering the development expenses, potential API subscription fees, and the long-term maintenance costs. This section should demonstrate the project's financial viability and justify the investment required.

The operational feasibility should assess the ease of user interaction with the MCQ Generator's interface, the learning curve for end-users, and the potential for integration with existing educational platforms or systems. This ensures the project's seamless integration and adoption within the target user base.

Finally, the legal and ethical feasibility should address any concerns related to data privacy, security, and compliance with API usage terms. This section should outline the strategies employed to mitigate these risks and ensure the project's adherence to relevant regulations and guidelines.

Chapter IV: Requirement Analysis

The Requirement Analysis chapter should outline the functional and non-functional requirements for the MCQ Generator project. This section should provide a detailed breakdown of the key features and capabilities the project aims to deliver.

The functional requirements should include:
1. Question Generation: Describe the ability to generate MCQs using the Google Gemini API, support for multiple subject domains, and customization options for question difficulty.
2. Export Capabilities: Outline the requirements for exporting generated questions in PDF and text formats, ensuring the preservation of formatting and structure.
3. User Interface: Detail the responsive design, intuitive question generation workflow, and effective error handling mechanisms.

The non-functional requirements should address aspects such as performance, security, usability, and cross-browser compatibility. This ensures that the MCQ Generator project not only meets the functional needs but also delivers a high-quality, reliable, and user-friendly experience.

Chapter V: Methodology

The Methodology chapter should outline the step-by-step approach adopted for the development of the MCQ Generator project. This section should cover the key phases, including project planning, design, development, testing, and deployment.

The project planning phase should involve defining the project scope, setting milestones, and identifying the core technologies to be used. The design phase should focus on creating wireframes, designing the user interface, and planning the API integration strategy.

The development phase should detail the implementation of the backend with Python, the frontend with HTML and CSS, the integration of the Google Gemini API, and the development of the export functionality. This section should also touch upon the specific libraries, frameworks, and best practices employed during the coding process.

The testing phase should describe the various testing strategies, including unit testing, integration testing, and user acceptance testing. This ensures the project's functionality, reliability, and overall quality.

Finally, the deployment and documentation phase should outline the steps taken to make the MCQ Generator available to end-users, as well as the creation of comprehensive technical and user documentation.

Chapter VI: Technology Description

The Technology Description chapter should provide a detailed overview of the key technologies used in the MCQ Generator project.

The Python section should explain how the backend programming language is utilized to handle API requests, process the generated questions, and manage the export functionality. This section should also touch upon the specific Python libraries and modules employed in the project.

The HTML/CSS section should discuss the role of these technologies in creating the responsive and visually appealing user interface. It should highlight the design approach, including the use of modern CSS techniques and the adherence to web development best practices.

The Google Gemini API section should delve into the integration of this AI-powered question generation service. It should explain how the API is leveraged to create diverse and intelligent multiple-choice questions, and the benefits this brings to the overall user experience.

6.1 Python

The backend of the MCQ Generator is built using Python, a powerful and versatile programming language. The Python code, encapsulated in the `app.py` file, is responsible for the following key functionalities:

6.1.1 API Key Management

The project begins by setting the Google Gemini API key as an environment variable, which is then used to configure the `genai` module for interacting with the API.

Python code:

```
os.environ["GOOGLE_API_KEY"] = "AIzaSyAlcW6I205HjS-9eQMlaAB8Rzf_4XQKtqc"
genai.configure(api_key=os.environ["GOOGLE_API_KEY"])
model = genai.GenerativeModel("models/gemini-1.5-pro")
```

This ensures secure storage and usage of the API credentials, which are essential for communicating with the Google Gemini question generation service.

6.1.2 File Handling and Text Extraction

The `app.py` file includes utility functions for handling user-uploaded files and extracting the text content from them. The `allowed_file()` function checks the file extension against a list of approved formats (PDF, DOCX, and TXT), while the `extract_text_from_file()` function uses appropriate libraries (pdfplumber, python-docx) to extract the text from the uploaded file.

Python code:

```
def allowed_file(filename):
        return '.' in filename and filename.rsplit('.', 1)[1].lower() in
app.config['ALLOWED_EXTENSIONS']

def extract_text_from_file(file_path):
        # Text extraction logic using pdfplumber, python-docx, and file reading
        # ...
```

These functions ensure that the system can handle various file formats and retrieve the necessary text content for the MCQ generation process.

6.1.3 MCQ Generation

The core functionality of the MCQ Generator is implemented in the `Question_mcqs_generator()` function. This function takes the extracted text and the desired number of questions as input, and then leverages the Google Gemini API to generate the corresponding multiple-choice questions.

Python code:

```
def Question_mcqs_generator(input_text, num_questions):
        prompt = f"""
        You are an AI assistant helping the user generate multiple-choice questions (MCQs)
based on the following text:
        '{input_text}'
        Please generate {num_questions} MCQs from the text. Each question should have:
        - A clear question
        - Four answer options (labeled A, B, C, D)
        - The correct answer clearly indicated
        # MCQ
        # MCQ
        # MCQ
        """
        response = model.generate_content(prompt).text.strip()
        return response
```

The function constructs a prompt that instructs the Google Gemini API to generate the specified number of MCQs based on the provided text. The API's response is then returned as a formatted string containing the generated questions.

6.1.4 File Storage and Exports

To save the generated MCQs, the `app.py` file includes functions to write the questions to text files (`save_mcqs_to_file()`) and generate PDF documents (`create_pdf()`). These functions ensure that users can download the MCQs in their preferred format.

Python code:

```
def save_mcqs_to_file(mcqs, filename):
        # Save MCQs to a text file
        # ...

def create_pdf(mcqs, filename):
        # Generate a PDF document from the MCQs
        # ...
```

The saved files are stored in the configured `RESULTS_FOLDER` directory, allowing users to access and download the generated content.

6.1.5 Web Application Integration

Finally, the `app.py` file integrates the backend functionality with the Flask web framework to create a seamless user experience. The `index()` and `generate_mcqs()` routes handle the user interactions, allowing them to upload files, specify the number of questions, and view/download the generated MCQs.

Python code:

```
@app.route('/')
def index():
        return render_template('index.html')

@app.route('/generate', methods=['POST'])
def generate_mcqs():
        # MCQ generation and file handling logic
        # ...
```

The `download_file()` route enables users to download the generated MCQ files in their preferred format (text or PDF).

Overall, the `app.py` file showcases the Python-powered backend of the MCQ Generator, demonstrating its ability to interact with the Google Gemini API, handle user-uploaded content, generate high-quality multiple-choice questions, and provide flexible export options to the end-users.

6.2 HTML

The frontend of the MCQ Generator project utilizes HTML (Hypertext Markup Language) to create the user interface and structure the web application. The `index.html` file contains the main entry point for the application, defining the layout and interactive elements that users will interact with.

6.2.1 User Interface Structure

The HTML code sets up the basic structure of the web page, including a container div to hold the main content, a header with the project title, and a form for user input.

HTML Code:

```
<div class="container">
        <h1><i class="fas fa-pencil-alt"></i> Generate MCQs</h1>
        <form action="/generate" method="POST" enctype="multipart/form-data">
        <!-- Form inputs and button -->
        </form>
        <div class="footer">
        <!-- Footer content -->
        </div>
</div>
```

This structure provides a clean and organized layout, ensuring a visually appealing and intuitive user experience.

6.2.2 Form Inputs

The HTML form contains two input fields: one for the user to upload a document (PDF, TXT, or DOCX), and another for the user to specify the desired number of MCQ questions to generate.

HTML Code:

```
<div class="form-group">
        <label for="file">Upload your document (PDF, TXT, DOCX):</label>
        <input type="file" name="file" accept=".pdf, .txt, .docx" required>
</div>
<div class="form-group">
        <label for="num_questions">How many questions do you want?</label>
        <input type="number" name="num_questions" min="1" required>
</div>
```

These input fields allow users to provide the necessary information for the MCQ generation process, which is then handled by the backend Python code.

6.2.3 Styling and Responsiveness

The HTML code is accompanied by a CSS stylesheet that applies a consistent visual style to the web page. This includes a modern and clean design, with a gradient background, centered layout, and responsive behavior for different screen sizes.

CSS Code:

```
body {
        font-family: 'Arial', sans-serif;
        background: linear-gradient(to right, #f4f4f9, #e3f2fd);
        /* Additional styles */
}

@media (max-width: 600px) {
        h1 {
        font-size: 1.5rem;
        }
}
```

The CSS styles ensure that the MCQ Generator's user interface is visually appealing, accessible, and adapts well to various devices and screen sizes.

Overall, the `index.html` file sets the foundation for the MCQ Generator's frontend, providing a user-friendly and responsive interface for interacting with the application's core functionality.

6.3 HTML

In addition to the `index.html` file that defines the input interface for the MCQ Generator, the project also includes the `results.html` template to handle the display of the generated multiple-choice questions.

6.3.1 Structured and Visually Appealing Layout

The `results.html` file establishes a well-organized and aesthetically pleasing layout for presenting the generated MCQs. It utilizes a centralized container div to house the content, ensuring a clean and consistent visual presentation.

HTML Code:

```
<div class="container">
        <h1>Generated MCQs</h1>

        <!-- MCQ content -->

        <a href="/download/{{ txt_filename }}">Download as TXT</a>
        <a href="/download/{{ pdf_filename }}">Download as PDF</a>
</div>
```

The use of a gradient background and subtle hover effects on the container element creates a visually engaging and professional-looking user experience.

6.3.2 Dynamic MCQ Rendering

The `results.html` template leverages Jinja2 templating syntax to dynamically render the generated multiple-choice questions. It iterates through the `mcqs` variable, which contains the MCQ content, and displays each question, answer options, and the correct answer.

HTML Code:

```
{% for mcq in mcqs.split("## MCQ") %}
        {% if mcq.strip() %}
        <div class="mcq" role="article">
        <div class="question">{{ mcq.split('A)')[0].strip() }}</div>
        <div class="options">
        <!-- Render answer options -->
        </div>
        <div class="correct-answer">
        Correct Answer: {{ mcq.split('Correct Answer:')[1].strip() }}
        </div>
        </div>
        {% endif %}
{% endfor %}
```

This dynamic rendering approach ensures that the application can seamlessly display the generated MCQs, regardless of the number or content of the questions.

6.3.4 Download Functionality

In addition to the MCQ display, the `results.html` file includes links that allow users to download the generated content in both text (TXT) and PDF formats. These links leverage the Flask routing and file download functionality implemented in the backend `app.py` code.

HTML Code:

```
<a href="/download/{{ txt_filename }}">Download as TXT</a>
<a href="/download/{{ pdf_filename }}">Download as PDF</a>
```

This comprehensive output display, coupled with the convenient download options, provides users with a complete and user-friendly experience for accessing the generated multiple-choice questions.

Overall, the `results.html` template plays a crucial role in the MCQ Generator project, delivering a visually appealing, interactive, and functional interface for presenting the AI-generated content to end-users.

Chapter VII: Testing

The Testing chapter should outline the various testing strategies and methodologies applied to the MCQ Generator project to ensure its quality and functionality.

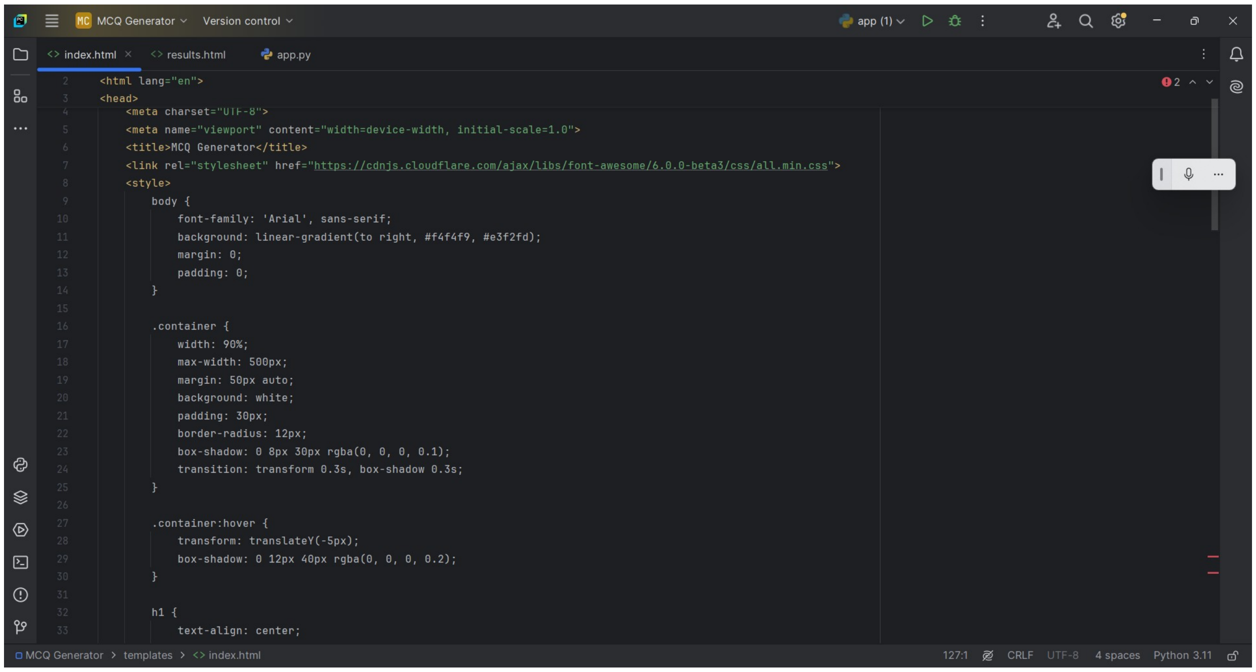
The Unit Testing section should describe the approach used to test individual components, such as the API integration, question generation logic, and export functionality. This ensures the reliable operation of each module within the system.

The Integration Testing section should focus on verifying the seamless communication between the different components of the project, including the API integration, export mechanisms, and the overall workflow.

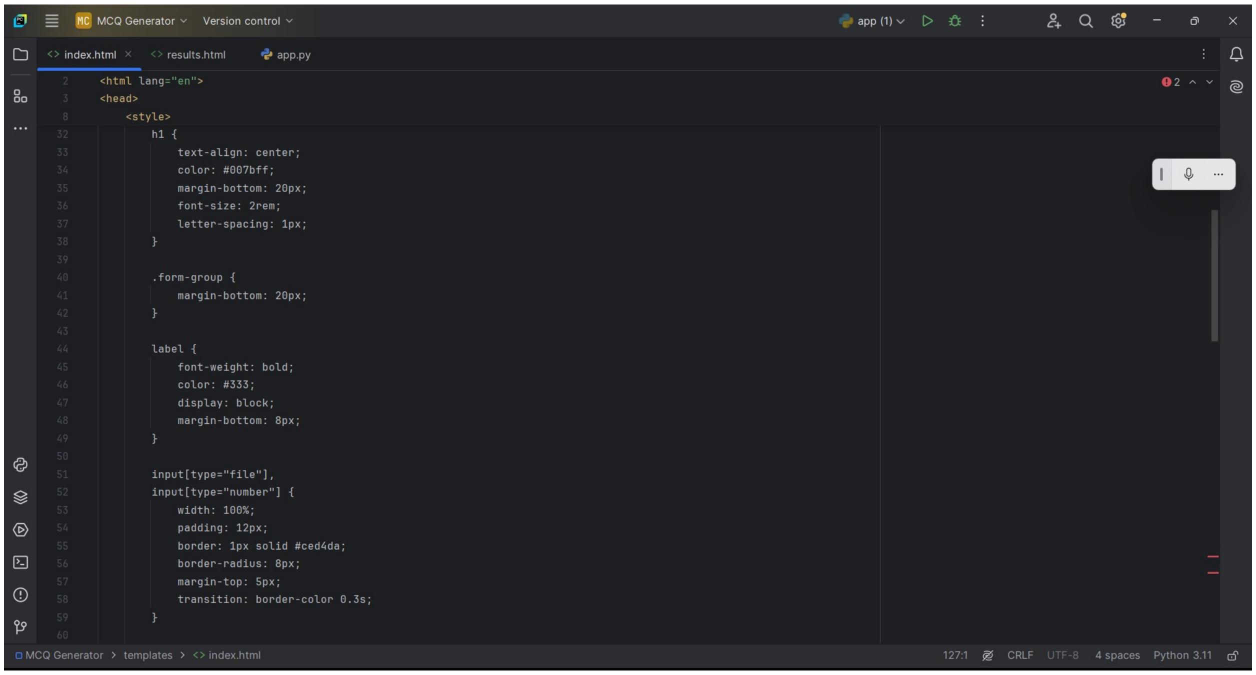
The User Interface Testing section should address the evaluation of the responsive design, cross-browser compatibility, and the usability assessment to validate the project's adherence to best practices and end-user requirements.

This comprehensive testing approach helps to identify and address any issues or bugs early in the development process, ultimately delivering a high-quality MCQ Generator solution.

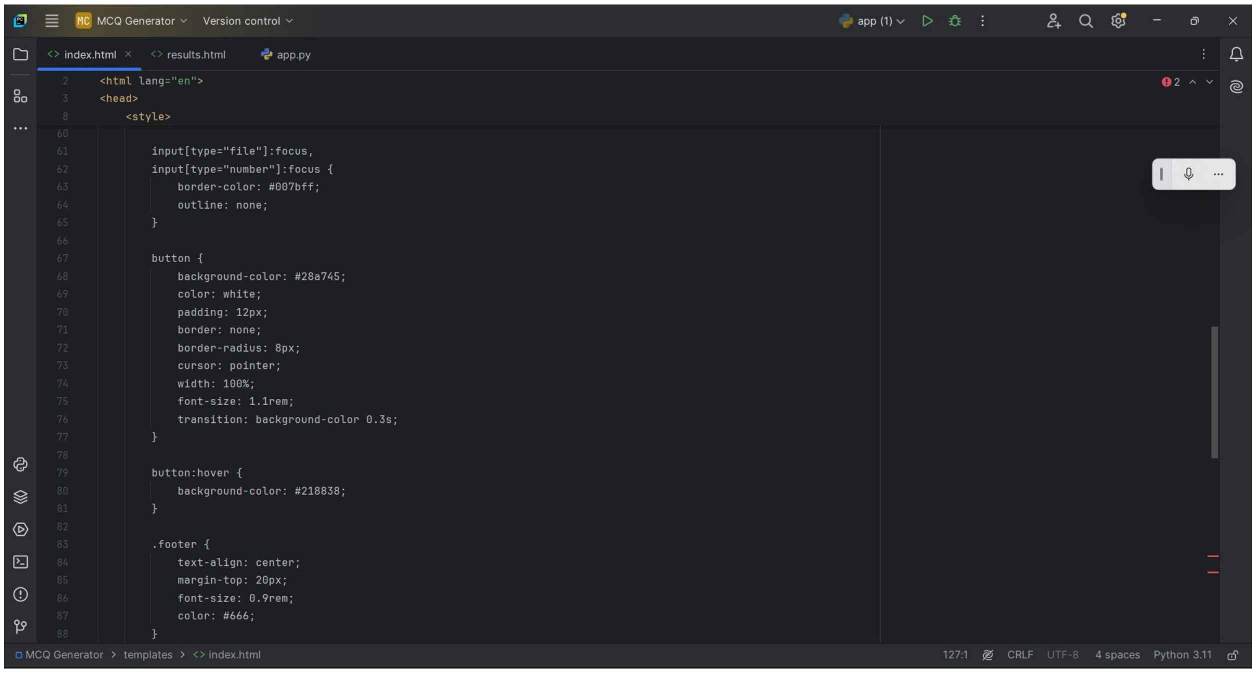
index.html code



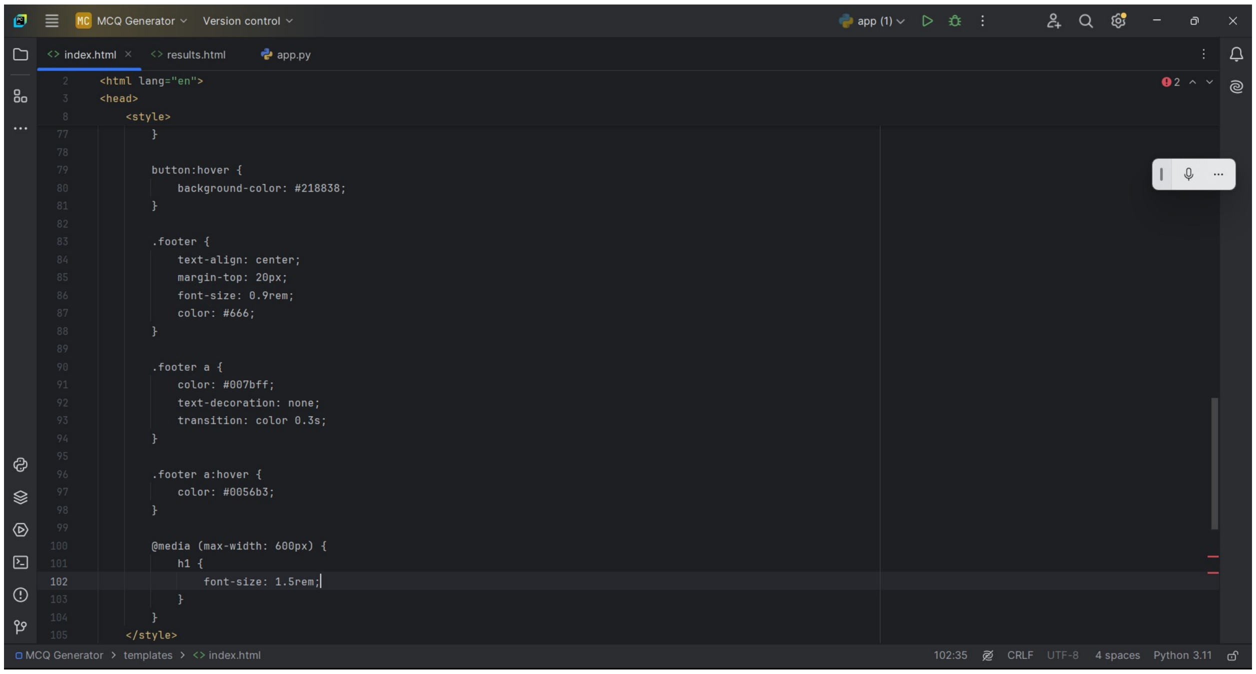
```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>MCQ Generator</title>
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.0.0-beta3/css/all.min.css">
    <style>
        body {
            font-family: 'Arial', sans-serif;
            background: linear-gradient(to right, #f4f4f9, #e3f2fd);
            margin: 0;
            padding: 0;
        }

        .container {
            width: 90%;
            max-width: 500px;
            margin: 50px auto;
            background: white;
            padding: 30px;
            border-radius: 12px;
            box-shadow: 0 8px 30px rgba(0, 0, 0, 0.1);
            transition: transform 0.3s, box-shadow 0.3s;
        }

        .container:hover {
            transform: translateY(-5px);
            box-shadow: 0 12px 40px rgba(0, 0, 0, 0.2);
        }

        h1 {
            text-align: center;
```



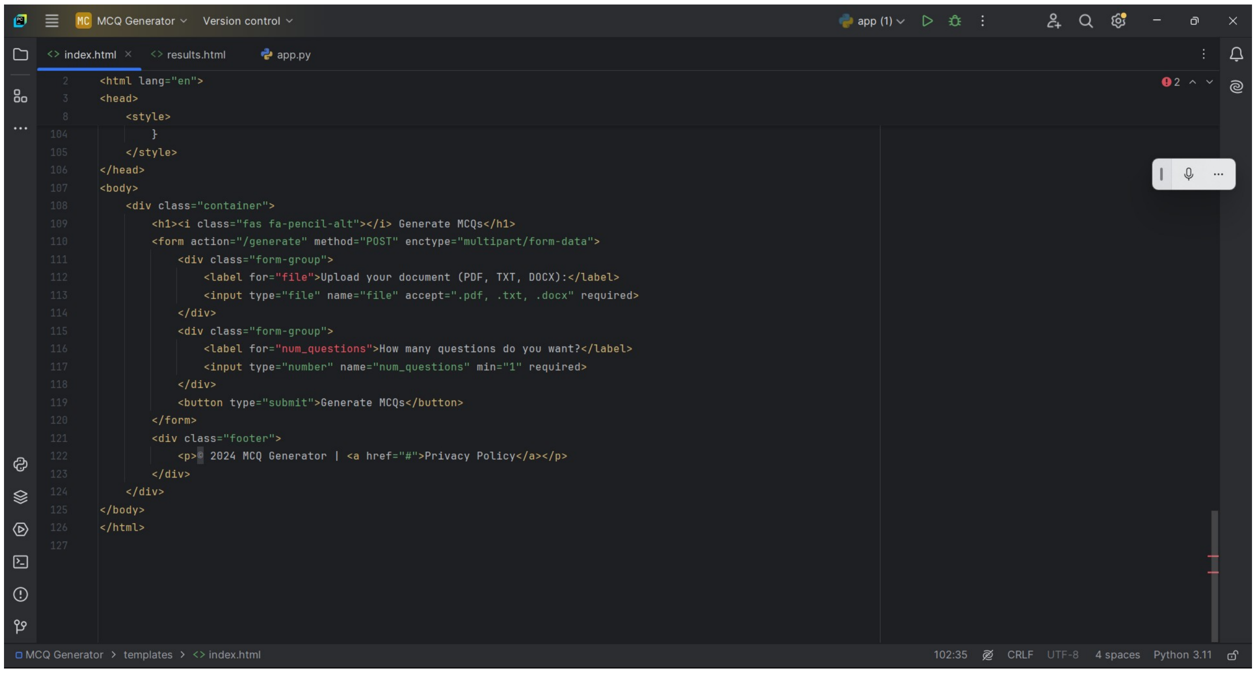
```
<html lang="en">
<head>
    <style>
        h1 {
            text-align: center;
            color: #007bff;
            margin-bottom: 20px;
            font-size: 2rem;
            letter-spacing: 1px;
        }

        .form-group {
            margin-bottom: 20px;
        }

        label {
            font-weight: bold;
            color: #333;
            display: block;
            margin-bottom: 8px;
        }

        input[type="file"],
        input[type="number"] {
            width: 100%;
            padding: 12px;
            border: 1px solid #ced4da;
            border-radius: 8px;
            margin-top: 5px;
            transition: border-color 0.3s;
        }
```


MCQ Generator
Version control
app (1)
index.html
results.html
app.py
MCQ Generator > templates > index.html
127:1 CRLF UTF-8 4 spaces Python 3.11

```
<html lang="en">
<head>
    <style>

        input[type="file"]:focus,
        input[type="number"]:focus {
            border-color: #007bff;
            outline: none;
        }

        button {
            background-color: #28a745;
            color: white;
            padding: 12px;
            border: none;
            border-radius: 8px;
            cursor: pointer;
            width: 100%;
            font-size: 1.1rem;
            transition: background-color 0.3s;
        }

        button:hover {
            background-color: #218838;
        }

        .footer {
            text-align: center;
            margin-top: 20px;
            font-size: 0.9rem;
            color: #666;
        }
```

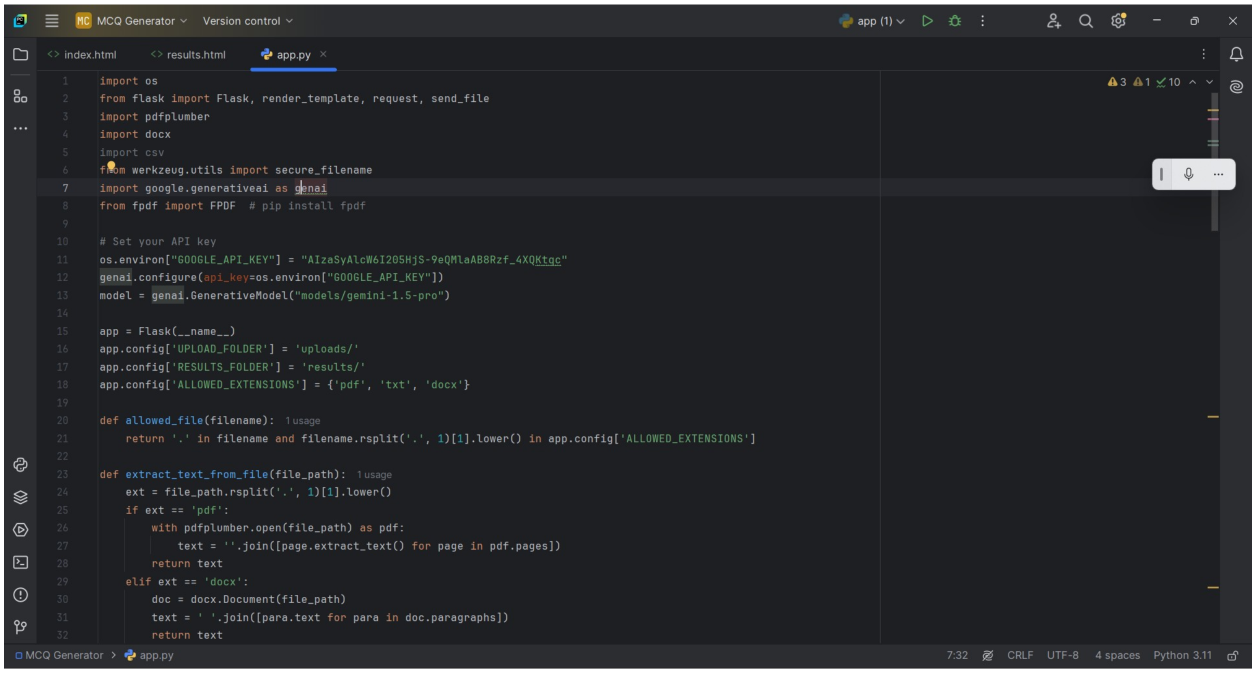

MCQ Generator
Version control
app (1)
index.html
results.html
app.py
MCQ Generator > templates > index.html
102:35 CRLF UTF-8 4 spaces Python 3.11

```
<html lang="en">
<head>
    <style>
        }

        button:hover {
            background-color: #218838;
        }

        .footer {
            text-align: center;
            margin-top: 20px;
            font-size: 0.9rem;
            color: #666;
        }

        .footer a {
            color: #007bff;
            text-decoration: none;
            transition: color 0.3s;
        }

        .footer a:hover {
            color: #0056b3;
        }

        @media (max-width: 600px) {
            h1 {
                font-size: 1.5rem;
            }
        }
    </style>
```

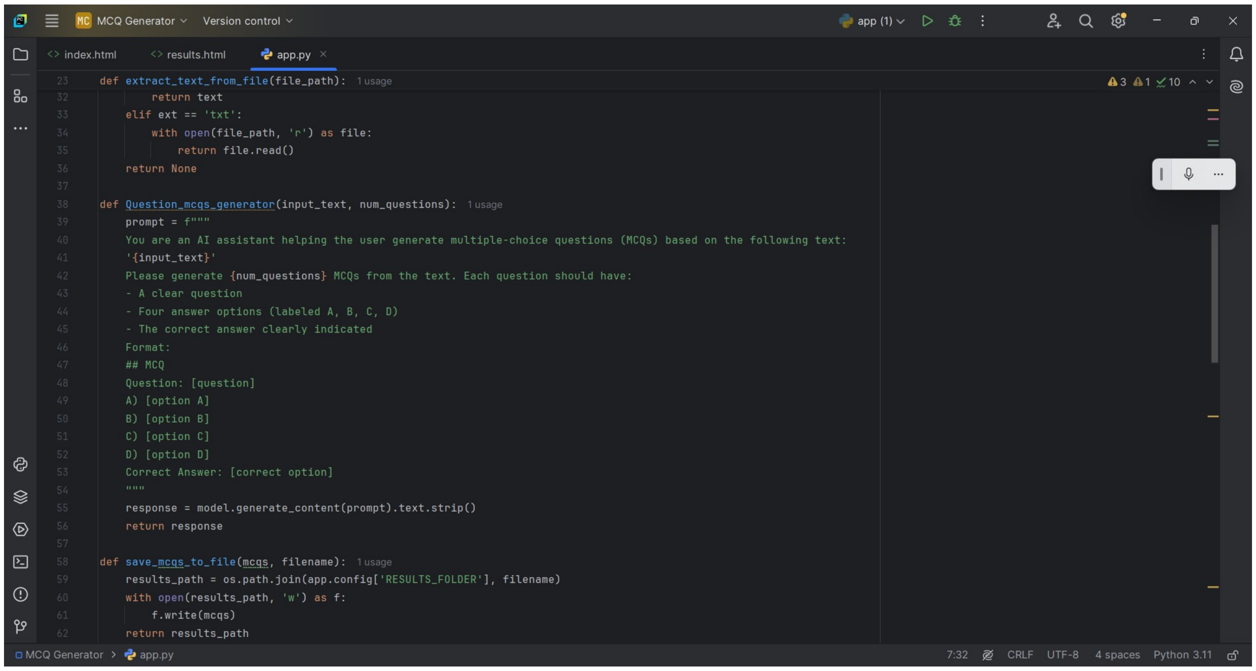


```html
<html lang="en">
<head>
    <style>
        }
    </style>
</head>
<body>
    <div class="container">
        <h1><i class="fas fa-pencil-alt"></i> Generate MCQs</h1>
        <form action="/generate" method="POST" enctype="multipart/form-data">
            <div class="form-group">
                <label for="file">Upload your document (PDF, TXT, DOCX):</label>
                <input type="file" name="file" accept=".pdf, .txt, .docx" required>
            </div>
            <div class="form-group">
                <label for="num_questions">How many questions do you want?</label>
                <input type="number" name="num_questions" min="1" required>
            </div>
            <button type="submit">Generate MCQs</button>
        </form>
        <div class="footer">
            <p>© 2024 MCQ Generator | <a href="#">Privacy Policy</a></p>
        </div>
    </div>
</body>
</html>
```

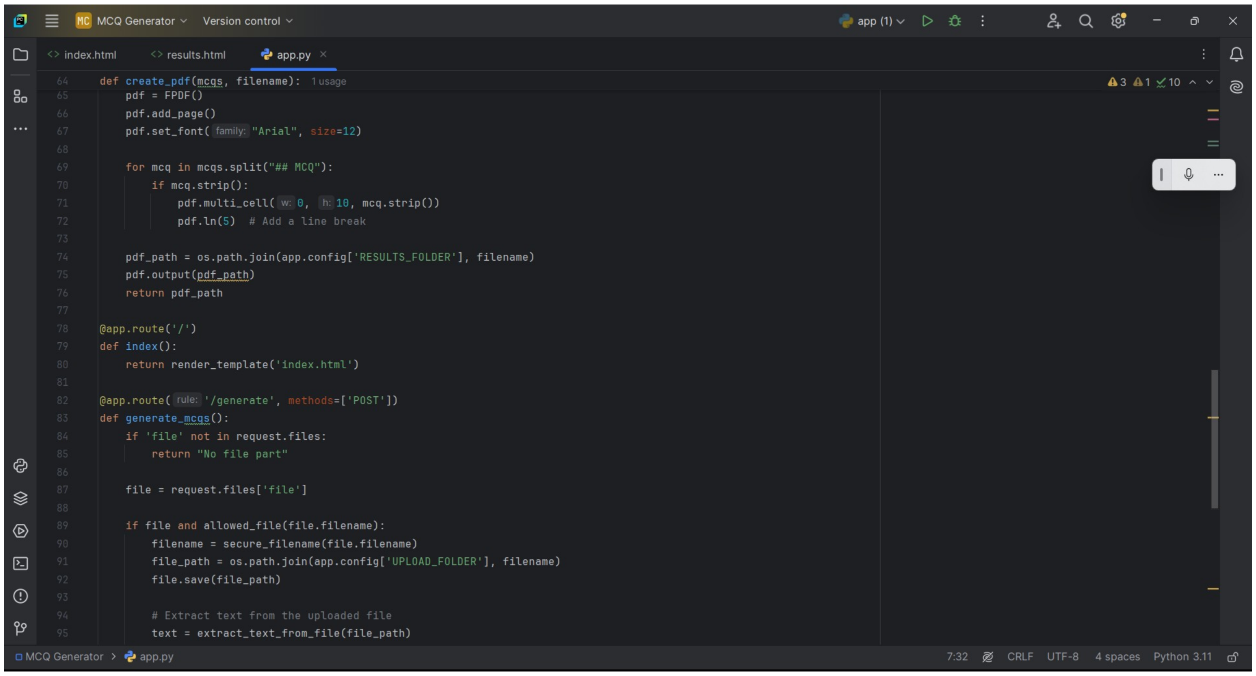
app.py code



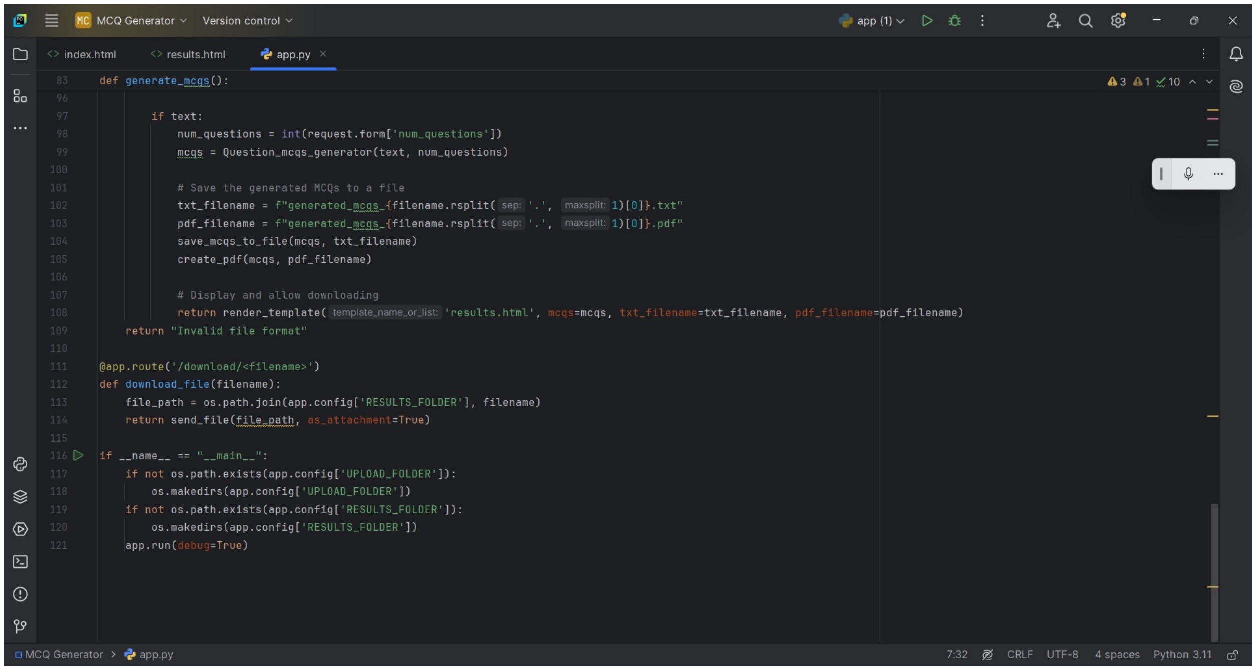
```python
import os
from flask import Flask, render_template, request, send_file
import pdfplumber
import docx
import csv
from werkzeug.utils import secure_filename
import google.generativeai as genai
from fpdf import FPDF  # pip install fpdf

# Set your API key
os.environ["GOOGLE_API_KEY"] = "AIzaSyAlcW6I205HjS-9eQMlaAB8Rzf_4XQKtqc"
genai.configure(api_key=os.environ["GOOGLE_API_KEY"])
model = genai.GenerativeModel("models/gemini-1.5-pro")

app = Flask(__name__)
app.config['UPLOAD_FOLDER'] = 'uploads/'
app.config['RESULTS_FOLDER'] = 'results/'
app.config['ALLOWED_EXTENSIONS'] = {'pdf', 'txt', 'docx'}

def allowed_file(filename):
    return '.' in filename and filename.rsplit('.', 1)[1].lower() in app.config['ALLOWED_EXTENSIONS']

def extract_text_from_file(file_path):
    ext = file_path.rsplit('.', 1)[1].lower()
    if ext == 'pdf':
        with pdfplumber.open(file_path) as pdf:
            text = ''.join([page.extract_text() for page in pdf.pages])
        return text
    elif ext == 'docx':
        doc = docx.Document(file_path)
        text = ' '.join([para.text for para in doc.paragraphs])
        return text
```



```python
def extract_text_from_file(file_path):
        return text
    elif ext == 'txt':
        with open(file_path, 'r') as file:
            return file.read()
    return None

def Question_mcqs_generator(input_text, num_questions):
    prompt = f"""
    You are an AI assistant helping the user generate multiple-choice questions (MCQs) based on the following text:
    '{input_text}'
    Please generate {num_questions} MCQs from the text. Each question should have:
    - A clear question
    - Four answer options (labeled A, B, C, D)
    - The correct answer clearly indicated
    Format:
    ## MCQ
    Question: [question]
    A) [option A]
    B) [option B]
    C) [option C]
    D) [option D]
    Correct Answer: [correct option]
    """
    response = model.generate_content(prompt).text.strip()
    return response

def save_mcqs_to_file(mcqs, filename):
    results_path = os.path.join(app.config['RESULTS_FOLDER'], filename)
    with open(results_path, 'w') as f:
        f.write(mcqs)
    return results_path
```

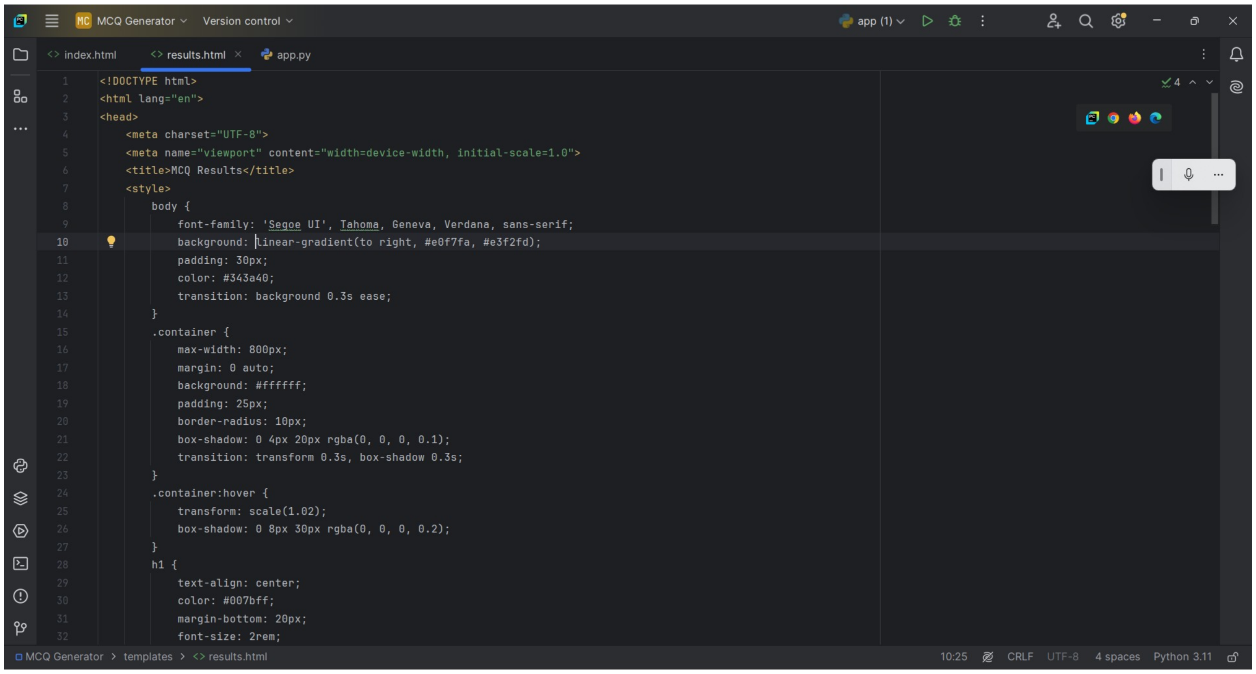


```python
def create_pdf(mcqs, filename):
    pdf = FPDF()
    pdf.add_page()
    pdf.set_font("Arial", size=12)

    for mcq in mcqs.split("## MCQ"):
        if mcq.strip():
            pdf.multi_cell(0, 10, mcq.strip())
            pdf.ln(5)  # Add a line break

    pdf_path = os.path.join(app.config['RESULTS_FOLDER'], filename)
    pdf.output(pdf_path)
    return pdf_path

@app.route('/')
def index():
    return render_template('index.html')

@app.route('/generate', methods=['POST'])
def generate_mcqs():
    if 'file' not in request.files:
        return "No file part"

    file = request.files['file']

    if file and allowed_file(file.filename):
        filename = secure_filename(file.filename)
        file_path = os.path.join(app.config['UPLOAD_FOLDER'], filename)
        file.save(file_path)

        # Extract text from the uploaded file
        text = extract_text_from_file(file_path)
```

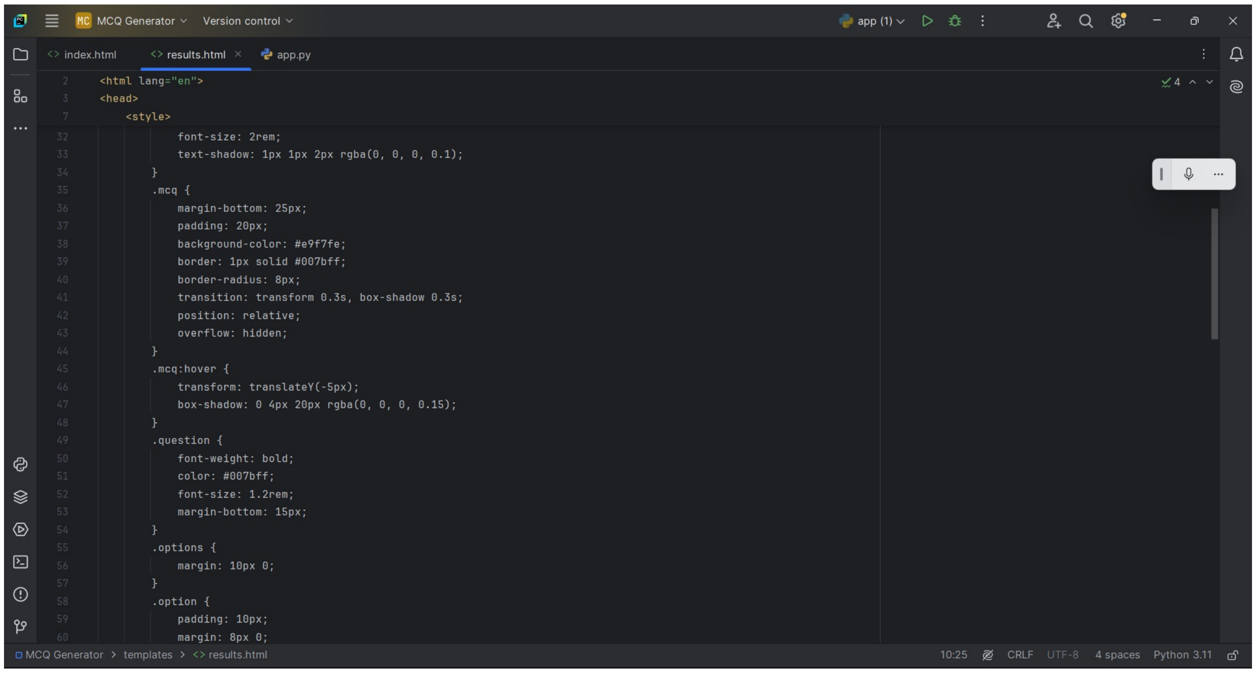


```python
def generate_mcqs():

        if text:
            num_questions = int(request.form['num_questions'])
            mcqs = Question_mcqs_generator(text, num_questions)

            # Save the generated MCQs to a file
            txt_filename = f"generated_mcqs_{filename.rsplit('.', 1)[0]}.txt"
            pdf_filename = f"generated_mcqs_{filename.rsplit('.', 1)[0]}.pdf"
            save_mcqs_to_file(mcqs, txt_filename)
            create_pdf(mcqs, pdf_filename)

            # Display and allow downloading
            return render_template('results.html', mcqs=mcqs, txt_filename=txt_filename, pdf_filename=pdf_filename)
    return "Invalid file format"

@app.route('/download/<filename>')
def download_file(filename):
    file_path = os.path.join(app.config['RESULTS_FOLDER'], filename)
    return send_file(file_path, as_attachment=True)

if __name__ == "__main__":
    if not os.path.exists(app.config['UPLOAD_FOLDER']):
        os.makedirs(app.config['UPLOAD_FOLDER'])
    if not os.path.exists(app.config['RESULTS_FOLDER']):
        os.makedirs(app.config['RESULTS_FOLDER'])
    app.run(debug=True)
```

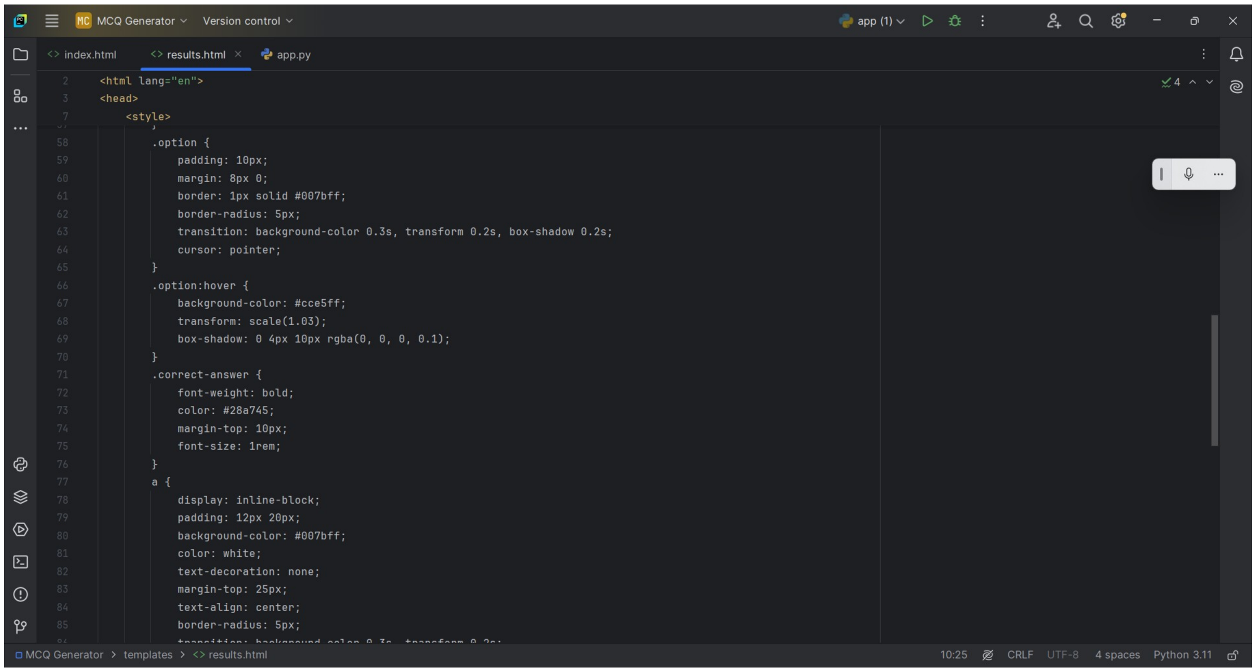
results.html code



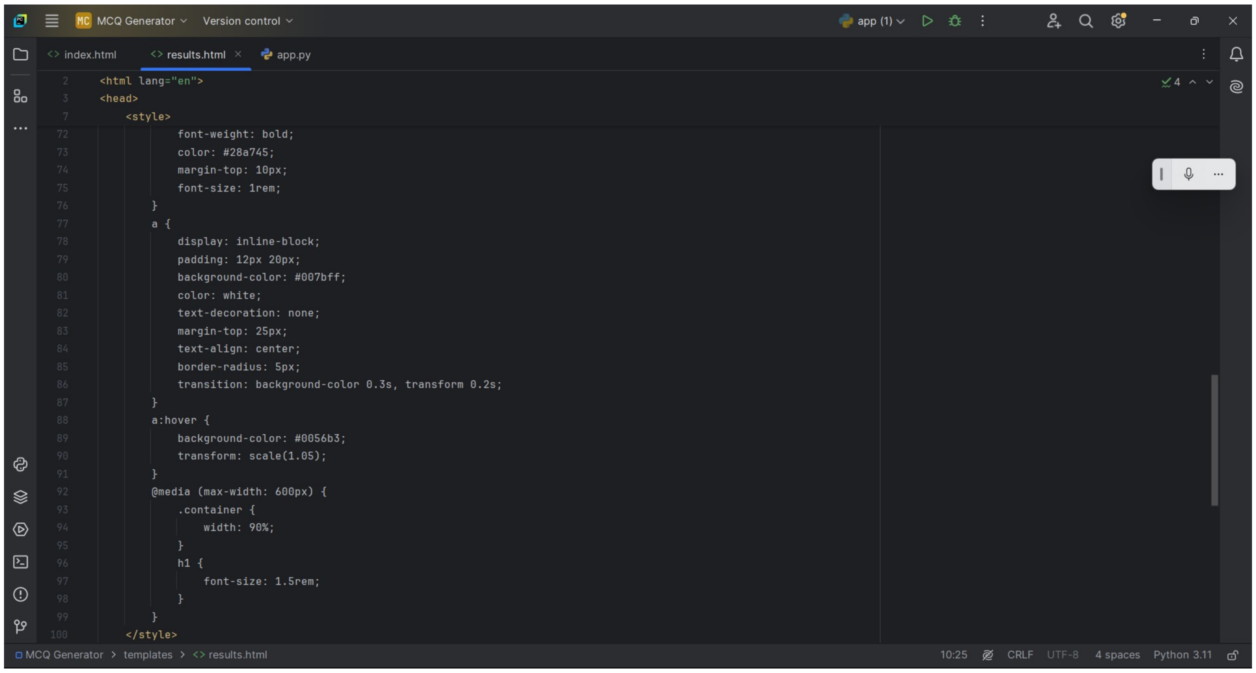
```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>MCQ Results</title>
    <style>
        body {
            font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
            background: linear-gradient(to right, #e0f7fa, #e3f2fd);
            padding: 30px;
            color: #343a40;
            transition: background 0.3s ease;
        }
        .container {
            max-width: 800px;
            margin: 0 auto;
            background: #ffffff;
            padding: 25px;
            border-radius: 10px;
            box-shadow: 0 4px 20px rgba(0, 0, 0, 0.1);
            transition: transform 0.3s, box-shadow 0.3s;
        }
        .container:hover {
            transform: scale(1.02);
            box-shadow: 0 8px 30px rgba(0, 0, 0, 0.2);
        }
        h1 {
            text-align: center;
            color: #007bff;
            margin-bottom: 20px;
            font-size: 2rem;
```



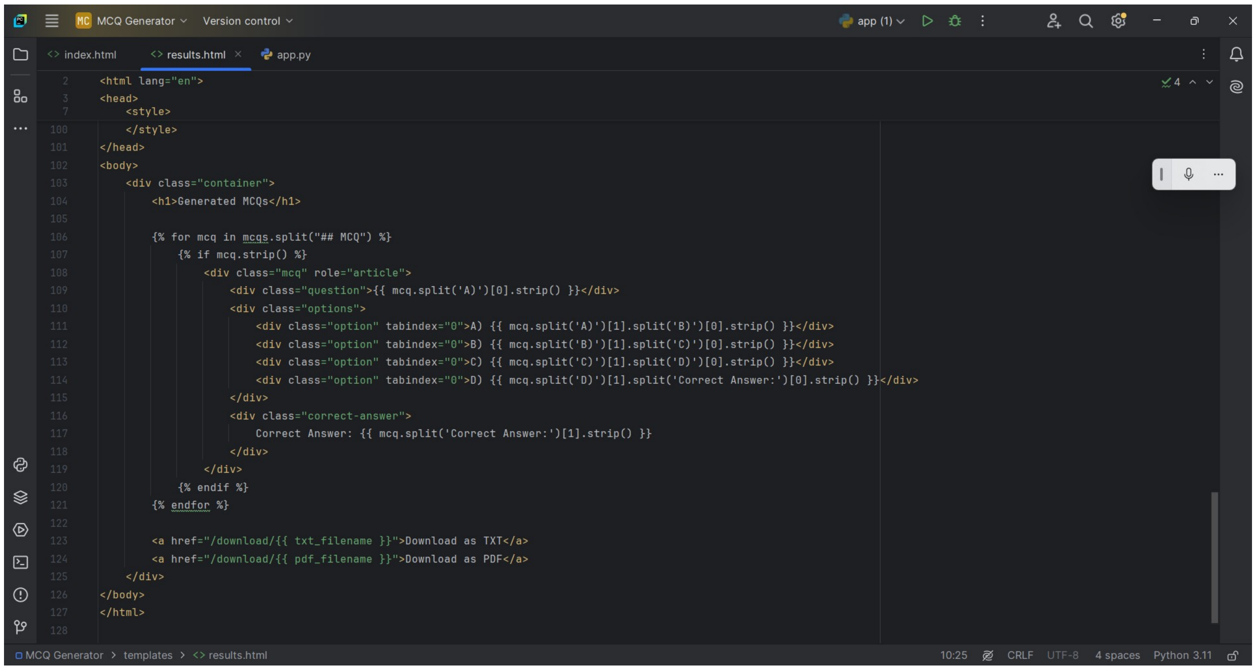
```
<html lang="en">
<head>
    <style>
            font-size: 2rem;
            text-shadow: 1px 1px 2px rgba(0, 0, 0, 0.1);
        }
        .mcq {
            margin-bottom: 25px;
            padding: 20px;
            background-color: #e9f7fe;
            border: 1px solid #007bff;
            border-radius: 8px;
            transition: transform 0.3s, box-shadow 0.3s;
            position: relative;
            overflow: hidden;
        }
        .mcq:hover {
            transform: translateY(-5px);
            box-shadow: 0 4px 20px rgba(0, 0, 0, 0.15);
        }
        .question {
            font-weight: bold;
            color: #007bff;
            font-size: 1.2rem;
            margin-bottom: 15px;
        }
        .options {
            margin: 10px 0;
        }
        .option {
            padding: 10px;
            margin: 8px 0;
```



```
<html lang="en">
<head>
    <style>
        .option {
            padding: 10px;
            margin: 8px 0;
            border: 1px solid #007bff;
            border-radius: 5px;
            transition: background-color 0.3s, transform 0.2s, box-shadow 0.2s;
            cursor: pointer;
        }
        .option:hover {
            background-color: #cce5ff;
            transform: scale(1.03);
            box-shadow: 0 4px 10px rgba(0, 0, 0, 0.1);
        }
        .correct-answer {
            font-weight: bold;
            color: #28a745;
            margin-top: 10px;
            font-size: 1rem;
        }
        a {
            display: inline-block;
            padding: 12px 20px;
            background-color: #007bff;
            color: white;
            text-decoration: none;
            margin-top: 25px;
            text-align: center;
            border-radius: 5px;
            transition: background-color 0.3s, transform 0.2s;
```

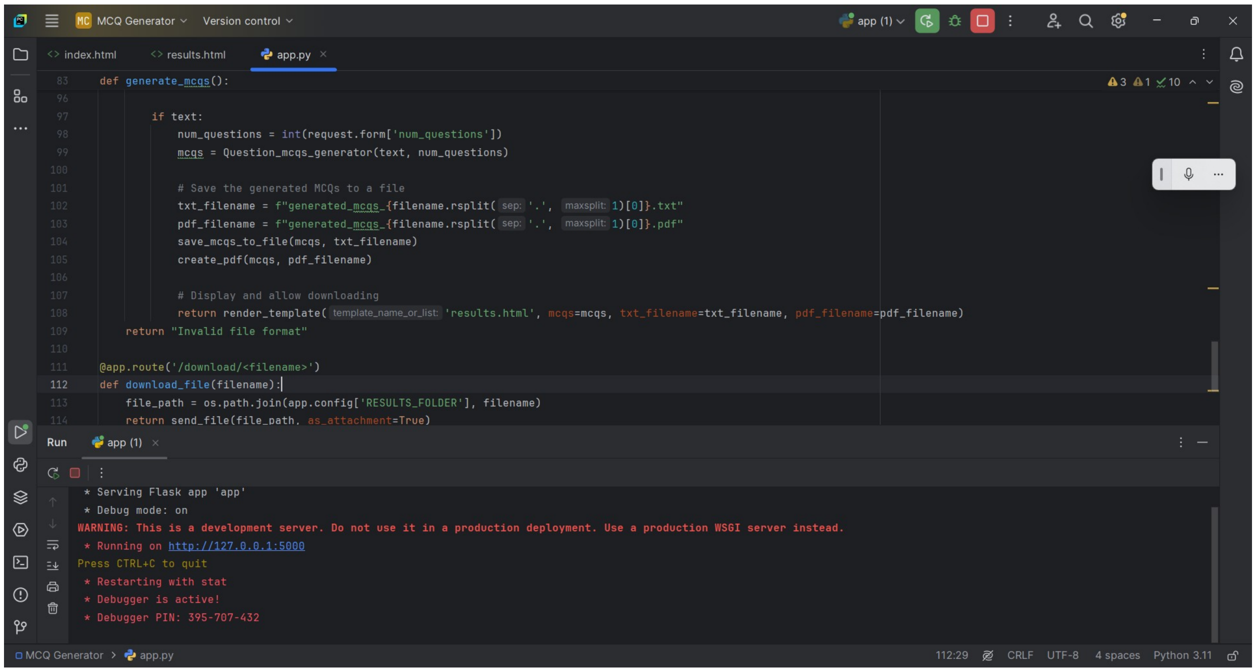


```
<html lang="en">
<head>
    <style>
            font-weight: bold;
            color: #28a745;
            margin-top: 10px;
            font-size: 1rem;
        }
        a {
            display: inline-block;
            padding: 12px 20px;
            background-color: #007bff;
            color: white;
            text-decoration: none;
            margin-top: 25px;
            text-align: center;
            border-radius: 5px;
            transition: background-color 0.3s, transform 0.2s;
        }
        a:hover {
            background-color: #0056b3;
            transform: scale(1.05);
        }
        @media (max-width: 600px) {
            .container {
                width: 90%;
            }
            h1 {
                font-size: 1.5rem;
            }
        }
    </style>
```



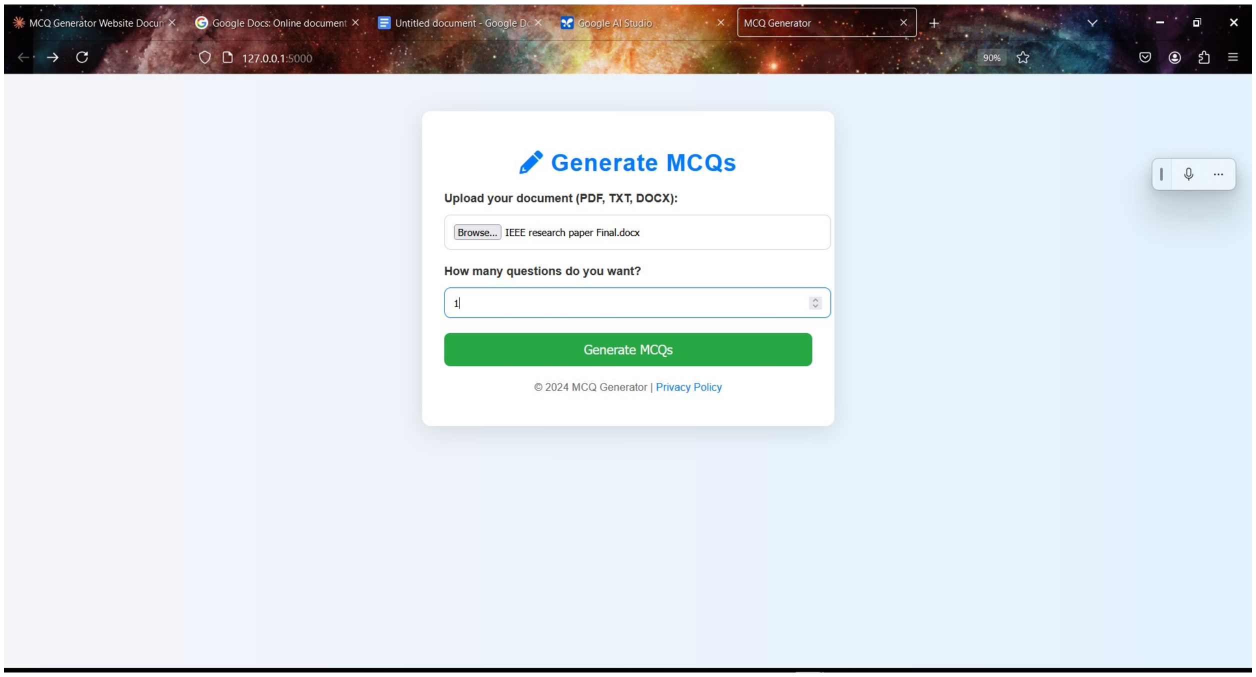
```
<html lang="en">
<head>
    <style>
    </style>
</head>
<body>
    <div class="container">
        <h1>Generated MCQs</h1>

        {% for mcq in mcqs.split("## MCQ") %}
            {% if mcq.strip() %}
                <div class="mcq" role="article">
                    <div class="question">{{ mcq.split('A)')[0].strip() }}</div>
                    <div class="options">
                        <div class="option" tabindex="0">A) {{ mcq.split('A)')[1].split('B)')[0].strip() }}</div>
                        <div class="option" tabindex="0">B) {{ mcq.split('B)')[1].split('C)')[0].strip() }}</div>
                        <div class="option" tabindex="0">C) {{ mcq.split('C)')[1].split('D)')[0].strip() }}</div>
                        <div class="option" tabindex="0">D) {{ mcq.split('D)')[1].split('Correct Answer:')[0].strip() }}</div>
                    </div>
                    <div class="correct-answer">
                        Correct Answer: {{ mcq.split('Correct Answer:')[1].strip() }}
                    </div>
                </div>
            {% endif %}
        {% endfor %}

        <a href="/download/{{ txt_filename }}">Download as TXT</a>
        <a href="/download/{{ pdf_filename }}">Download as PDF</a>
    </div>
</body>
</html>
```

Input interface



```
def generate_mcqs():

        if text:
            num_questions = int(request.form['num_questions'])
            mcqs = Question_mcqs_generator(text, num_questions)

            # Save the generated MCQs to a file
            txt_filename = f"generated_mcqs_{filename.rsplit('.', 1)[0]}.txt"
            pdf_filename = f"generated_mcqs_{filename.rsplit('.', 1)[0]}.pdf"
            save_mcqs_to_file(mcqs, txt_filename)
            create_pdf(mcqs, pdf_filename)

            # Display and allow downloading
            return render_template('results.html', mcqs=mcqs, txt_filename=txt_filename, pdf_filename=pdf_filename)
    return "Invalid file format"

@app.route('/download/<filename>')
def download_file(filename):
    file_path = os.path.join(app.config['RESULTS_FOLDER'], filename)
    return send_file(file_path, as_attachment=True)
```

```
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 395-707-432
```



Generate MCQs

Upload your document (PDF, TXT, DOCX):

Browse... IEEE research paper Final.docx

How many questions do you want?

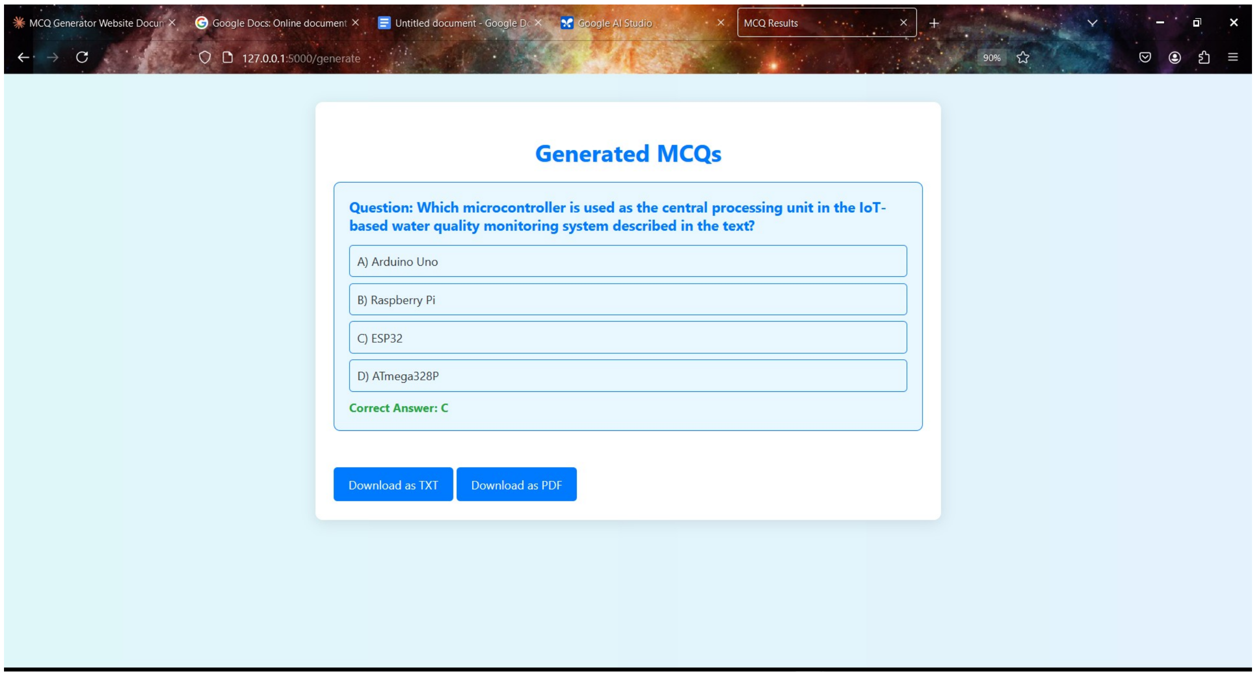
1

Generate MCQs

© 2024 MCQ Generator | Privacy Policy

MCQ Generator Website Docum
Google Docs: Online document
Untitled document - Google D
Google AI Studio
MCQ Generator
127.0.0.1:5000
90%

Output interface


MCQ Generator Website Docu
Google Docs: Online document
Untitled document - Google D
Google AI Studio
MCQ Results
127.0.0.1:5000/generate
90%

Generated MCQs

Question: Which microcontroller is used as the central processing unit in the IoT-based water quality monitoring system described in the text?

A) Arduino Uno

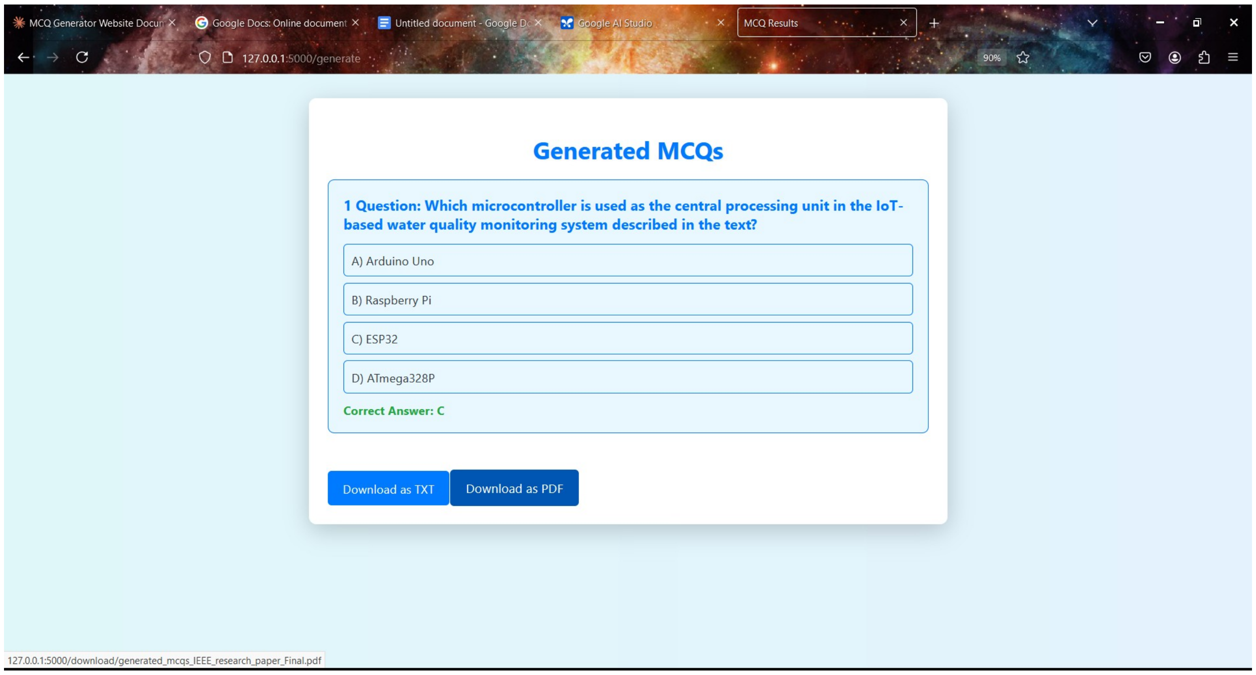
B) Raspberry Pi

C) ESP32

D) ATmega328P

Correct Answer: C

Download as pdf format


MCQ Generator Website Docu
Google Docs: Online document
Untitled document - Google D
Google AI Studio
MCQ Results
127.0.0.1:5000/generate
90%

Generated MCQs

1 Question: Which microcontroller is used as the central processing unit in the IoT-based water quality monitoring system described in the text?

A) Arduino Uno

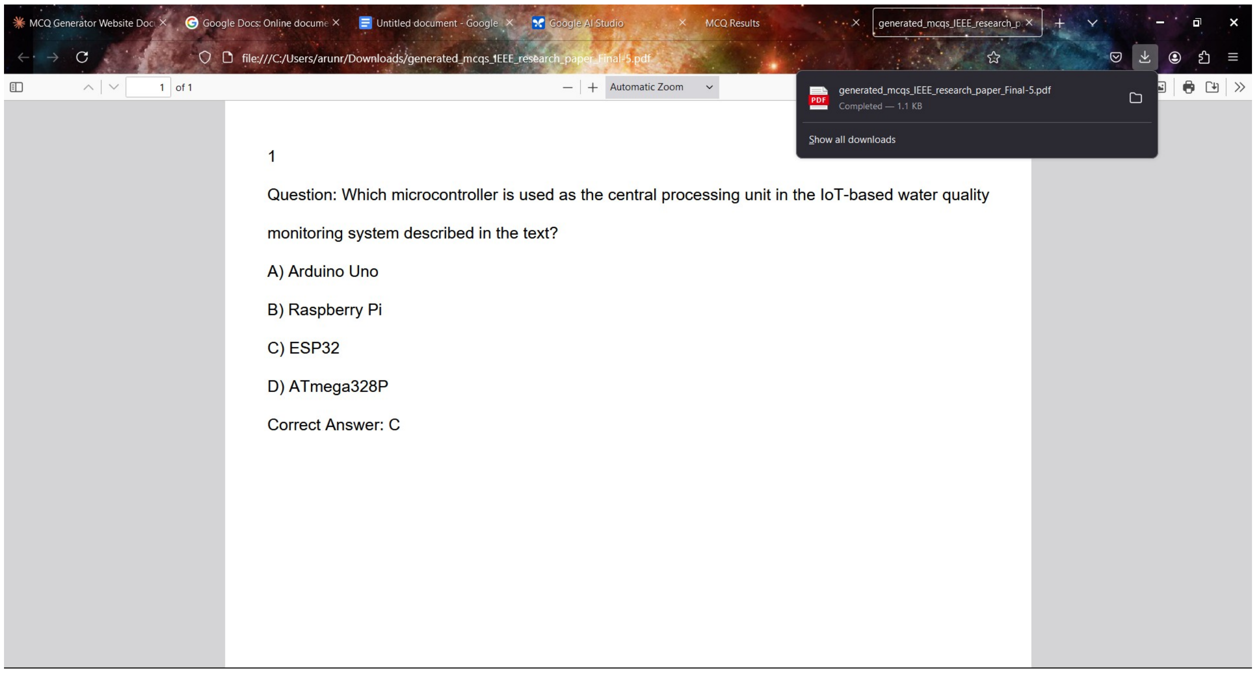
B) Raspberry Pi

C) ESP32

D) ATmega328P

Correct Answer: C

Download as TXT
Download as PDF
127.0.0.1:5000/download/generated_mcqs_IEEE_research_paper_Final.pdf


file:///C:/Users/arunr/Downloads/generated_mcqs_IEEE_research_paper_Final-5.pdf
generated_mcqs_IEEE_research_paper_Final-5.pdf
Completed — 1.1 KB
Show all downloads

1

Question: Which microcontroller is used as the central processing unit in the IoT-based water quality monitoring system described in the text?

A) Arduino Uno

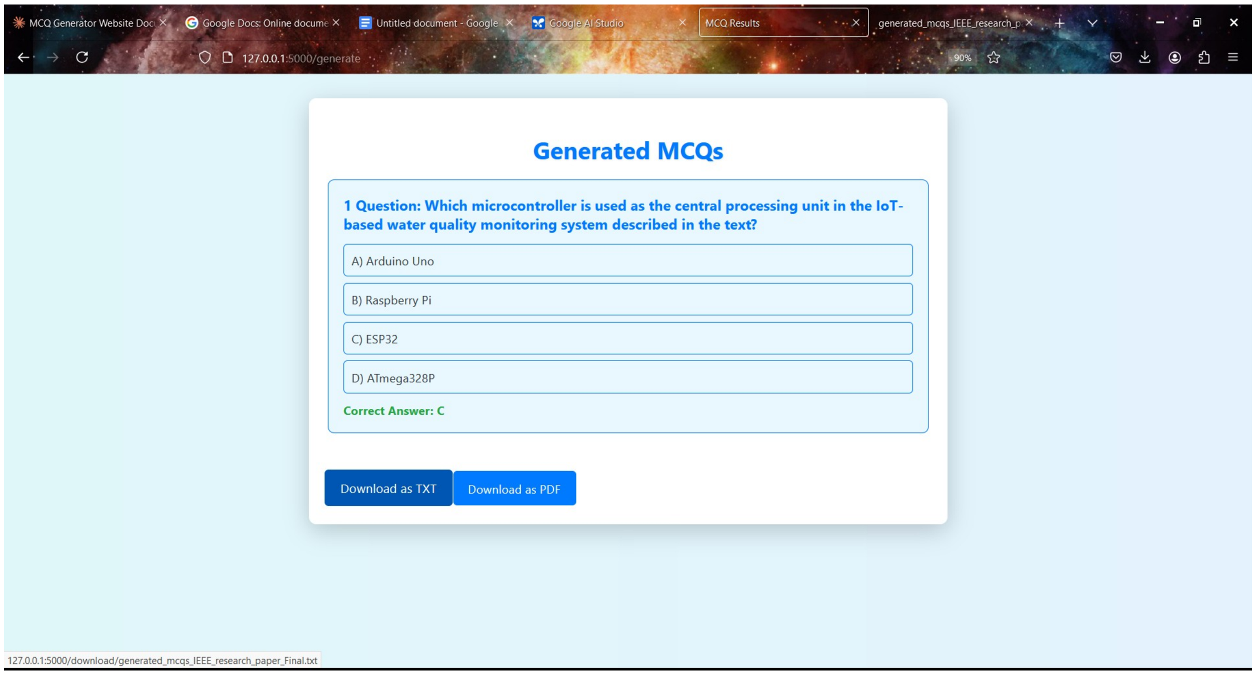
B) Raspberry Pi

C) ESP32

D) ATmega328P

Correct Answer: C

Download as text format



Generated MCQs

1 Question: Which microcontroller is used as the central processing unit in the IoT-based water quality monitoring system described in the text?

A) Arduino Uno

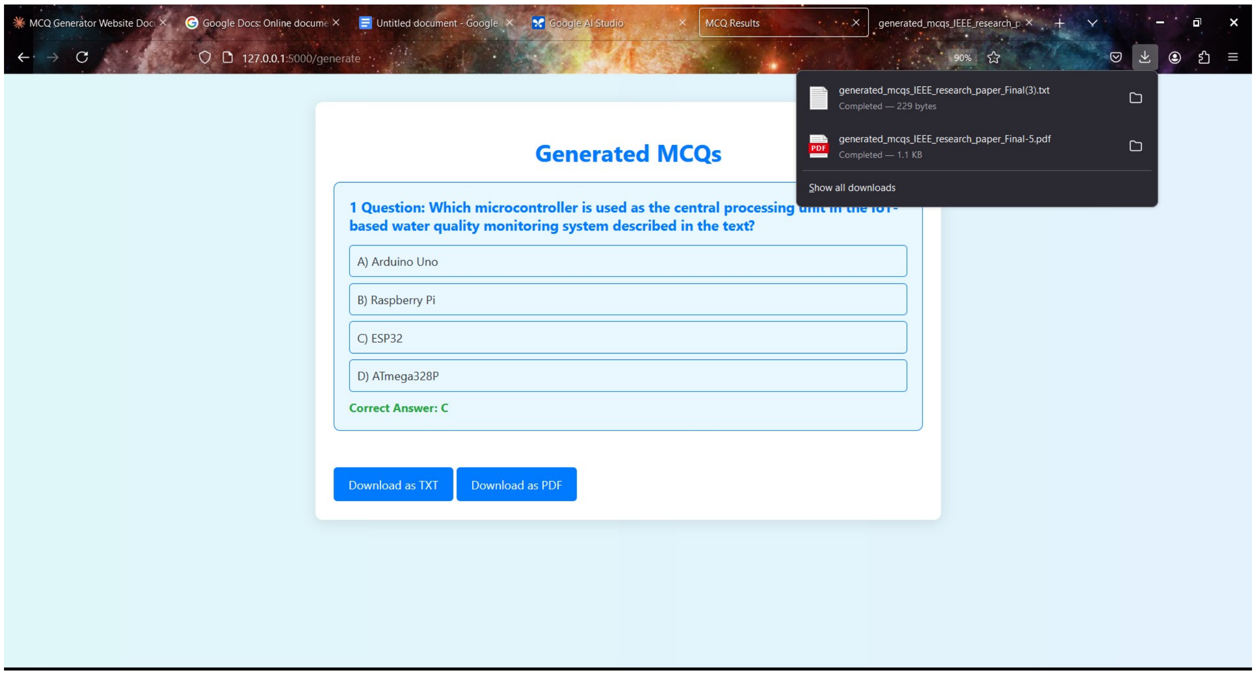
B) Raspberry Pi

C) ESP32

D) ATmega328P

Correct Answer: C

Download as TXT
Download as PDF
127.0.0.1:5000/download/generated_mcqs_IEEE_research_paper_Final.txt



Generated MCQs

1 Question: Which microcontroller is used as the central processing unit in the IoT-based water quality monitoring system described in the text?

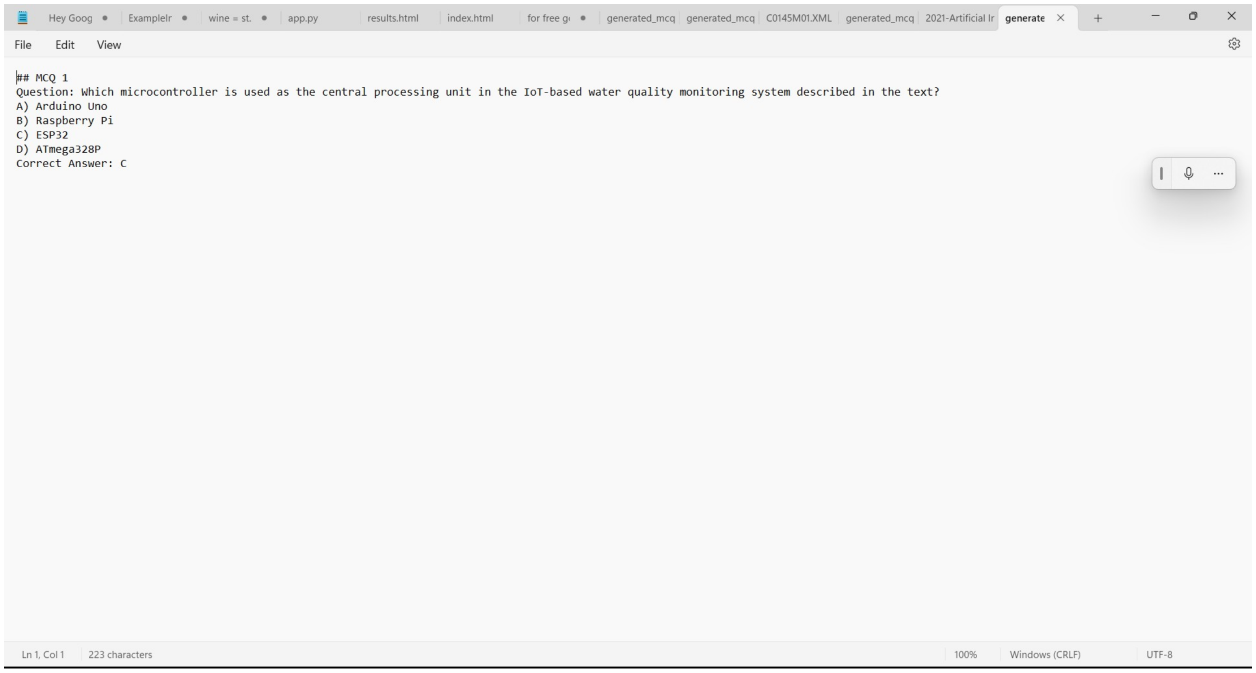
A) Arduino Uno

B) Raspberry Pi

C) ESP32

D) ATmega328P

Correct Answer: C



```
## MCQ 1
Question: Which microcontroller is used as the central processing unit in the IoT-based water quality monitoring system described in the text?
A) Arduino Uno
B) Raspberry Pi
C) ESP32
D) ATmega328P
Correct Answer: C
```

Chapter VIII: Results

The Results chapter should showcase the successful outcomes and key achievements of the MCQ Generator project. This section should highlight the project's ability to:

1. Reliably generate multiple-choice questions using the Google Gemini API.
2. Provide an intuitive and user-friendly interface for educators and learners.
3. Offer flexible export options, allowing users to generate questions in both PDF and text formats.
4. Demonstrate the potential for enhancing educational technology and assessment practices.

By summarizing the project's accomplishments, this chapter should demonstrate the value and impact the MCQ Generator can have in the educational technology landscape.

Chapter IX: Conclusion

The Conclusion chapter should provide a summary of the MCQ Generator project, emphasizing its significance and the potential for future enhancements. This section should reiterate the core objectives of the project and how the integration of Python, HTML, CSS, and the Google Gemini API has resulted in a powerful tool for streamlining question creation.

The conclusion should also outline potential areas for future development, such as support for additional question types, advanced customization options, and integration with learning management systems. This forward-looking perspective showcases the project's scalability and the opportunity for continuous improvement based on user feedback and evolving educational needs.

By clearly articulating the project's achievements and the possibilities for future growth, the Conclusion chapter should leave the reader with a comprehensive understanding of the MCQ Generator's impact and its potential to transform the way educators and learners approach assessment creation.

References:

1. Youtube Video Link: https://youtu.be/vIZGDMFSMBY?feature=shared
2. Pycharm Documentation Link: https://www.jetbrains.com/help/pycharm/creating-web-application-with-flask.html
3. Google AI Studio Link: https://aistudio.google.com/app/api