AI ASSISTED CODING

LAB ASSIGNMENT – 20.3

Name: Likhitha Pothunuri

Roll no: 2403A510D1

Batch 05

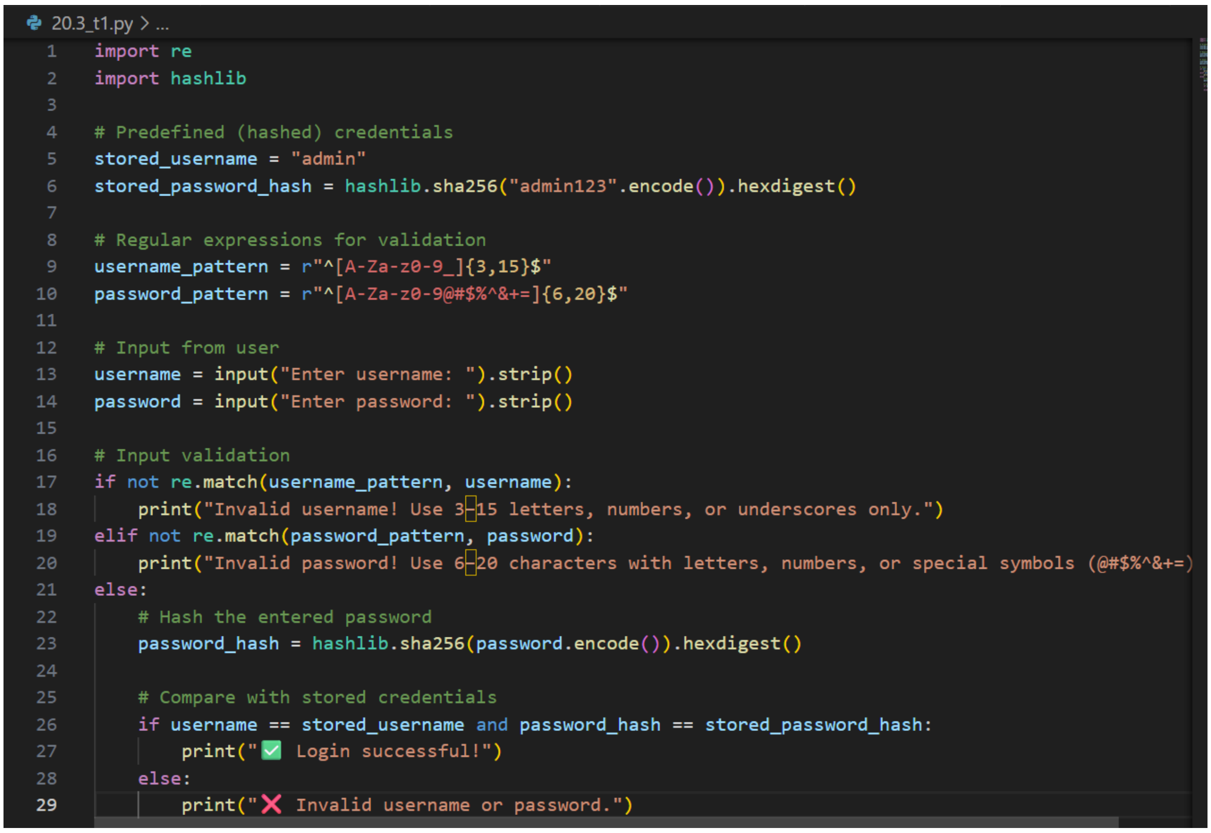
CSE 2nd year

Task 1 – Input Validation Check

Prompt:

"Generate a simple Python program that asks for a username and password and allows login if they match predefined credentials."

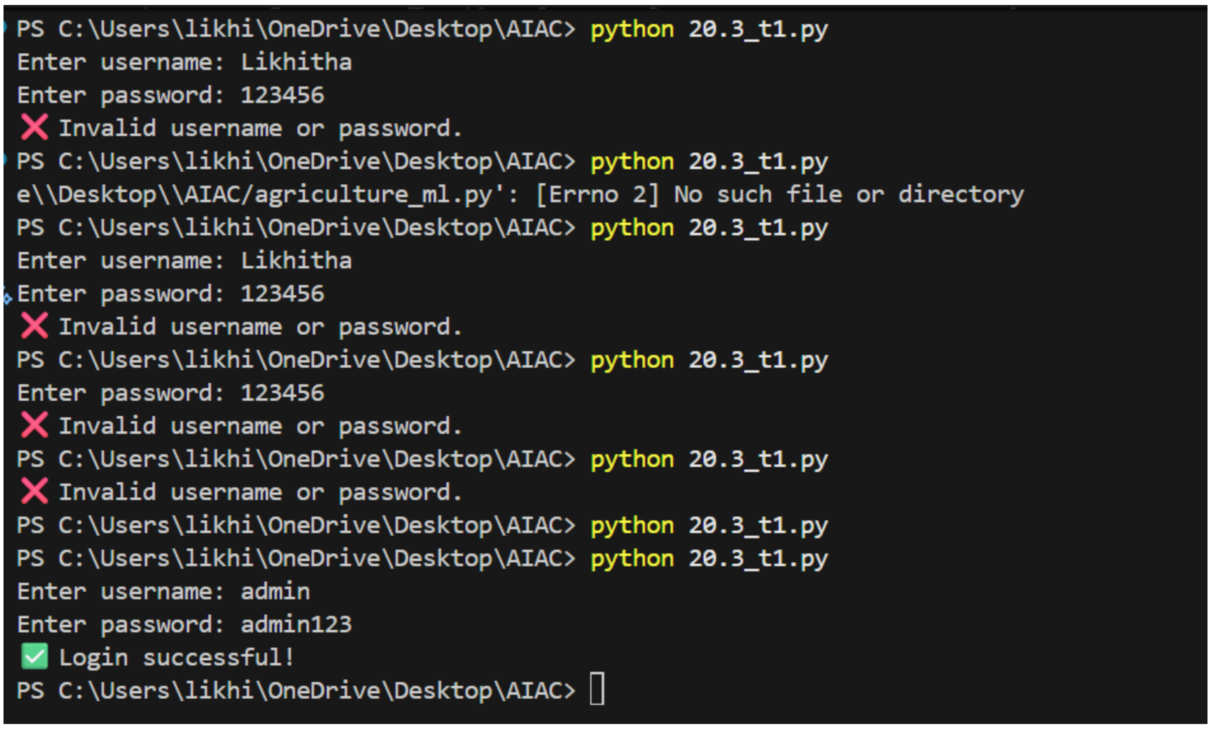
Code Generated:



20.3_t1.py > ...

```python
import re
import hashlib

# Predefined (hashed) credentials
stored_username = "admin"
stored_password_hash = hashlib.sha256("admin123".encode()).hexdigest()

# Regular expressions for validation
username_pattern = r"^[A-Za-z0-9_]{3,15}$"
password_pattern = r"^[A-Za-z0-9@#$%^&+=]{6,20}$"

# Input from user
username = input("Enter username: ").strip()
password = input("Enter password: ").strip()

# Input validation
if not re.match(username_pattern, username):
    print("Invalid username! Use 3–15 letters, numbers, or underscores only.")
elif not re.match(password_pattern, password):
    print("Invalid password! Use 6–20 characters with letters, numbers, or special symbols (@#$%^&+=)
else:
    # Hash the entered password
    password_hash = hashlib.sha256(password.encode()).hexdigest()

    # Compare with stored credentials
    if username == stored_username and password_hash == stored_password_hash:
        print("✅ Login successful!")
    else:
        print("❌ Invalid username or password.")
```

Output:



```
PS C:\Users\likhi\OneDrive\Desktop\AIAC> python 20.3_t1.py
Enter username: Likhitha
Enter password: 123456
❌ Invalid username or password.
PS C:\Users\likhi\OneDrive\Desktop\AIAC> python 20.3_t1.py
e\\Desktop\\AIAC/agriculture_ml.py': [Errno 2] No such file or directory
PS C:\Users\likhi\OneDrive\Desktop\AIAC> python 20.3_t1.py
Enter username: Likhitha
Enter password: 123456
❌ Invalid username or password.
PS C:\Users\likhi\OneDrive\Desktop\AIAC> python 20.3_t1.py
Enter password: 123456
❌ Invalid username or password.
PS C:\Users\likhi\OneDrive\Desktop\AIAC> python 20.3_t1.py
❌ Invalid username or password.
PS C:\Users\likhi\OneDrive\Desktop\AIAC> python 20.3_t1.py
PS C:\Users\likhi\OneDrive\Desktop\AIAC> python 20.3_t1.py
Enter username: admin
Enter password: admin123
✅ Login successful!
PS C:\Users\likhi\OneDrive\Desktop\AIAC> 
```

Observation:

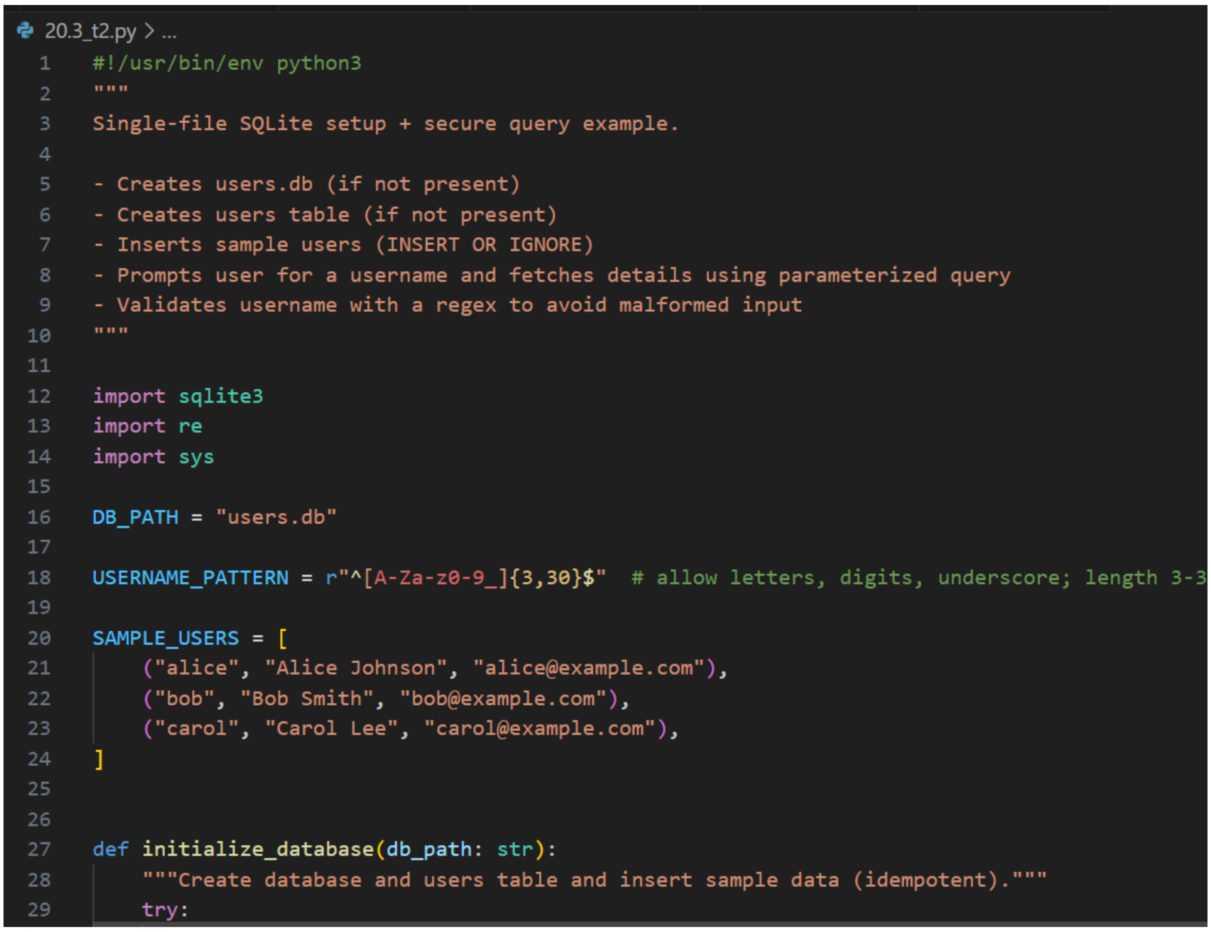
1. The initial AI-generated login script did not include any input validation or sanitization, making it insecure.
2. The improved version adds **regex-based input validation**, **password hashing**, and **safe comparison**, ensuring that only properly formatted and secure inputs are accepted.

Task 2 – SQL Injection Prevention

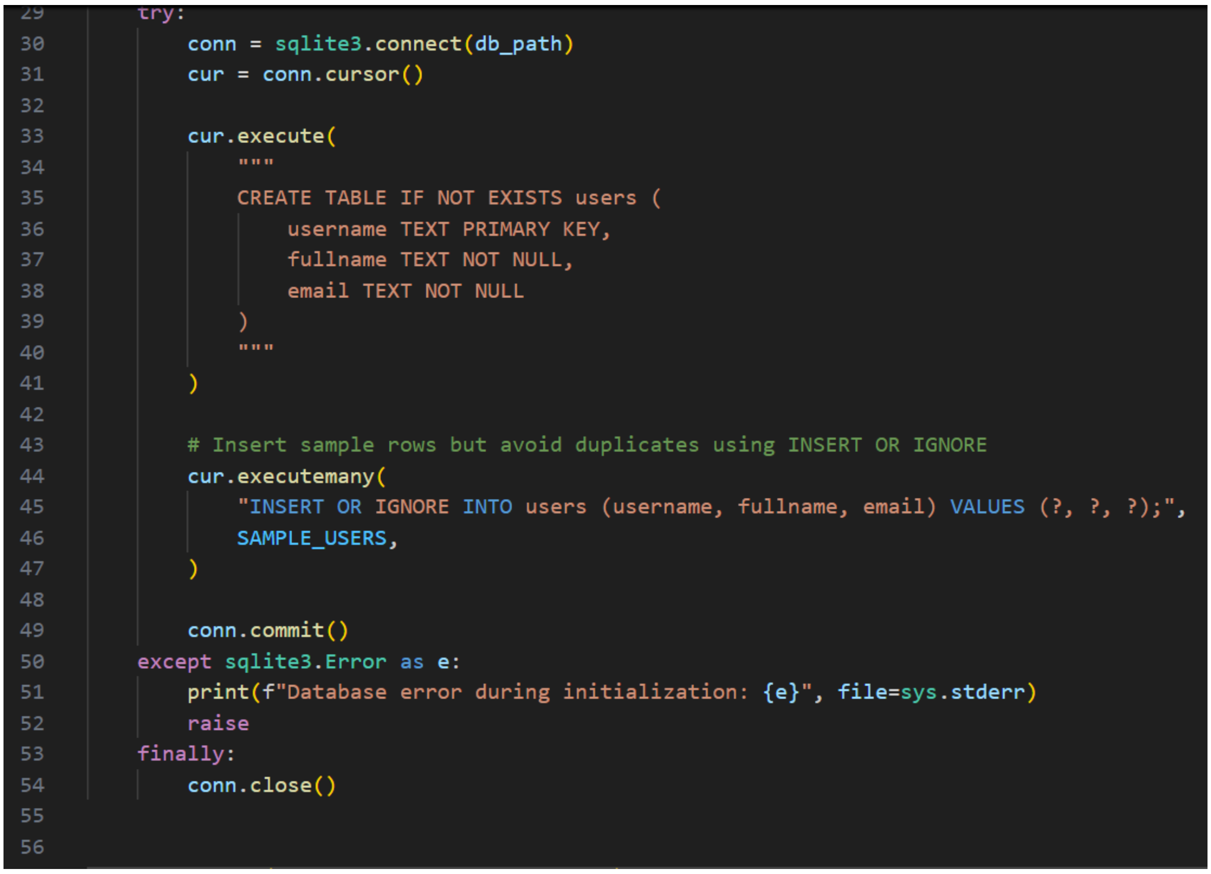
Prompt:

"Generate a Python script using SQLite to fetch user details from a database based on a username entered by the user."

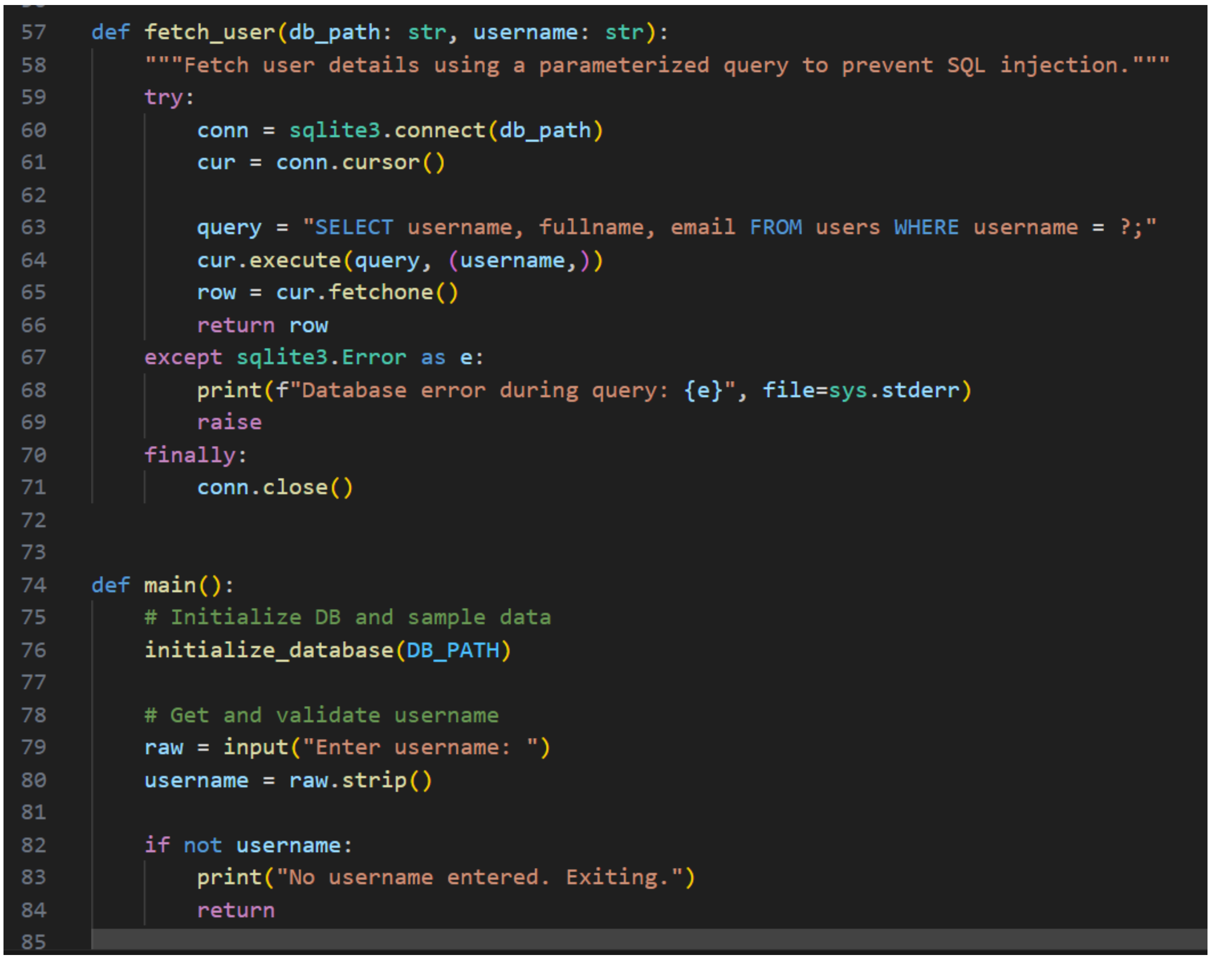
Code Generated:


20.3_t2.py > ...

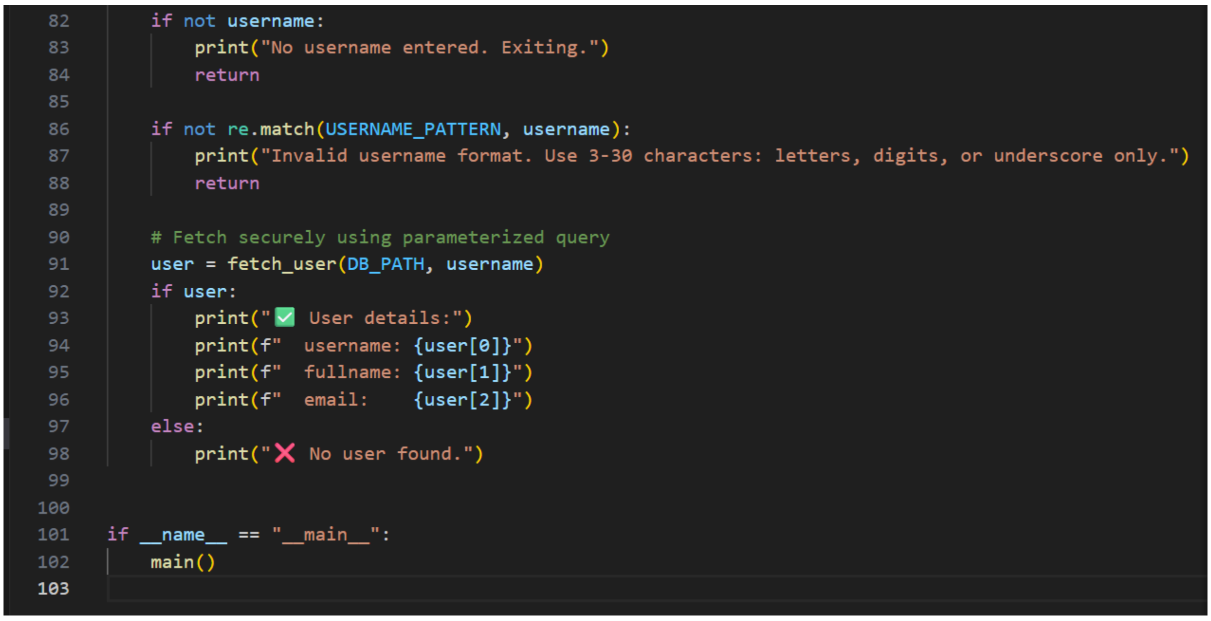
```python
#!/usr/bin/env python3
"""
Single-file SQLite setup + secure query example.

- Creates users.db (if not present)
- Creates users table (if not present)
- Inserts sample users (INSERT OR IGNORE)
- Prompts user for a username and fetches details using parameterized query
- Validates username with a regex to avoid malformed input
"""

import sqlite3
import re
import sys

DB_PATH = "users.db"

USERNAME_PATTERN = r"^[A-Za-z0-9_]{3,30}$"  # allow letters, digits, underscore; length 3-3

SAMPLE_USERS = [
    ("alice", "Alice Johnson", "alice@example.com"),
    ("bob", "Bob Smith", "bob@example.com"),
    ("carol", "Carol Lee", "carol@example.com"),
]


def initialize_database(db_path: str):
    """Create database and users table and insert sample data (idempotent)."""
    try:
```



```python
    try:
        conn = sqlite3.connect(db_path)
        cur = conn.cursor()

        cur.execute(
            """
            CREATE TABLE IF NOT EXISTS users (
                username TEXT PRIMARY KEY,
                fullname TEXT NOT NULL,
                email TEXT NOT NULL
            )
            """
        )

        # Insert sample rows but avoid duplicates using INSERT OR IGNORE
        cur.executemany(
            "INSERT OR IGNORE INTO users (username, fullname, email) VALUES (?, ?, ?);",
            SAMPLE_USERS,
        )

        conn.commit()
    except sqlite3.Error as e:
        print(f"Database error during initialization: {e}", file=sys.stderr)
        raise
    finally:
        conn.close()


```

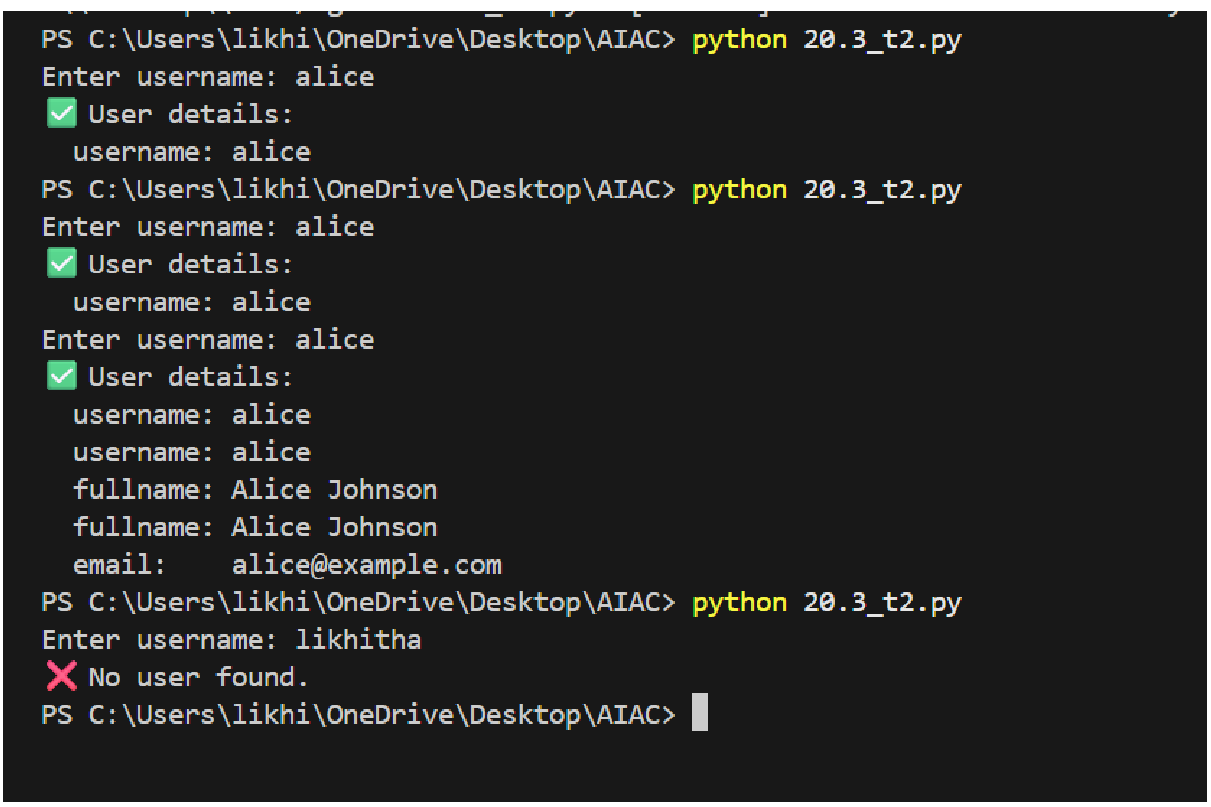


```python
def fetch_user(db_path: str, username: str):
    """Fetch user details using a parameterized query to prevent SQL injection."""
    try:
        conn = sqlite3.connect(db_path)
        cur = conn.cursor()

        query = "SELECT username, fullname, email FROM users WHERE username = ?;"
        cur.execute(query, (username,))
        row = cur.fetchone()
        return row
    except sqlite3.Error as e:
        print(f"Database error during query: {e}", file=sys.stderr)
        raise
    finally:
        conn.close()


def main():
    # Initialize DB and sample data
    initialize_database(DB_PATH)

    # Get and validate username
    raw = input("Enter username: ")
    username = raw.strip()

    if not username:
        print("No username entered. Exiting.")
        return

```



```python
    if not username:
        print("No username entered. Exiting.")
        return

    if not re.match(USERNAME_PATTERN, username):
        print("Invalid username format. Use 3-30 characters: letters, digits, or underscore only.")
        return

    # Fetch securely using parameterized query
    user = fetch_user(DB_PATH, username)
    if user:
        print("✅ User details:")
        print(f"  username: {user[0]}")
        print(f"  fullname: {user[1]}")
        print(f"  email:    {user[2]}")
    else:
        print("❌ No user found.")


if __name__ == "__main__":
    main()

```

Output:



```
PS C:\Users\likhi\OneDrive\Desktop\AIAC> python 20.3_t2.py
Enter username: alice
✅ User details:
  username: alice
PS C:\Users\likhi\OneDrive\Desktop\AIAC> python 20.3_t2.py
Enter username: alice
✅ User details:
  username: alice
Enter username: alice
✅ User details:
  username: alice
  username: alice
  fullname: Alice Johnson
  fullname: Alice Johnson
  email:    alice@example.com
PS C:\Users\likhi\OneDrive\Desktop\AIAC> python 20.3_t2.py
Enter username: likhitha
❌ No user found.
PS C:\Users\likhi\OneDrive\Desktop\AIAC> 
```

Observation:

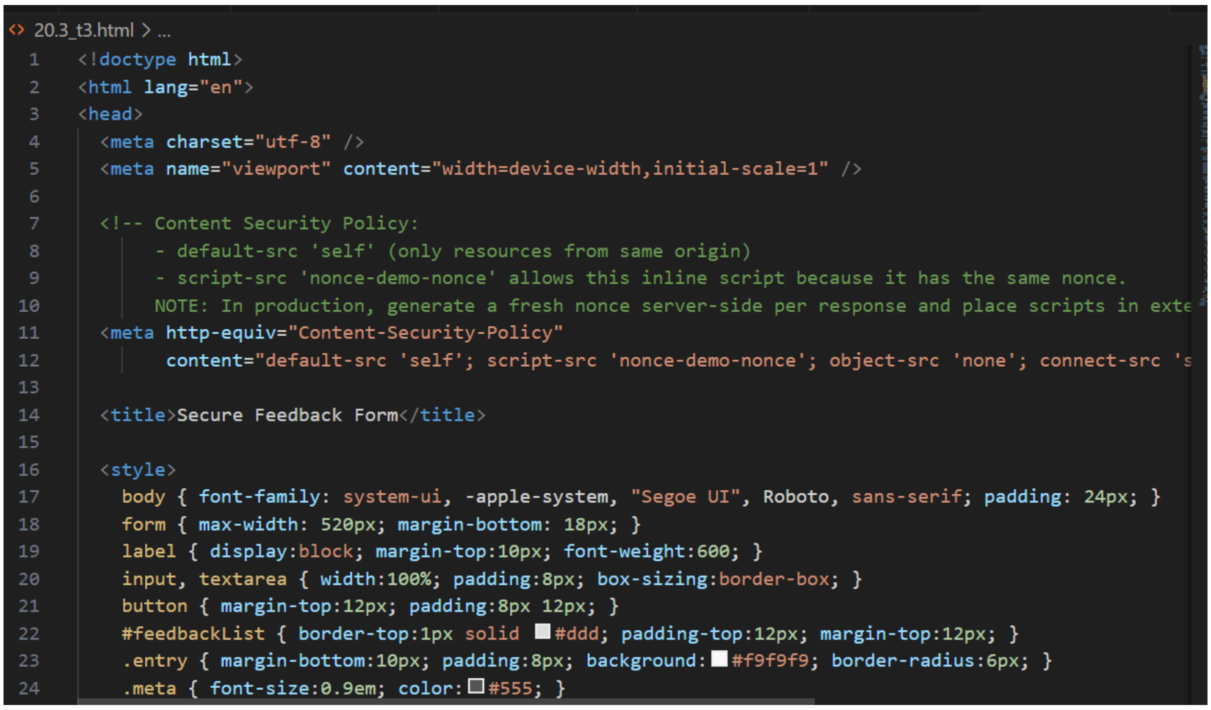
1. The original AI-generated SQL script was **vulnerable to SQL injection** because it used **string concatenation** to build queries directly from user input.In the improved version, **parameterized queries** (? placeholders) are used, which prevent malicious SQL code from being executed.
2. Additionally, the script includes **input validation using regex** to ensure usernames follow a safe pattern, and it **creates the database and sample table automatically** if they don't exist.
3. This makes the program **secure, self-contained, and reliable**, protecting both the database and user data from injection or accidental corruption.

Task 3 – Cross-Site Scripting (XSS) Check

Prompt:

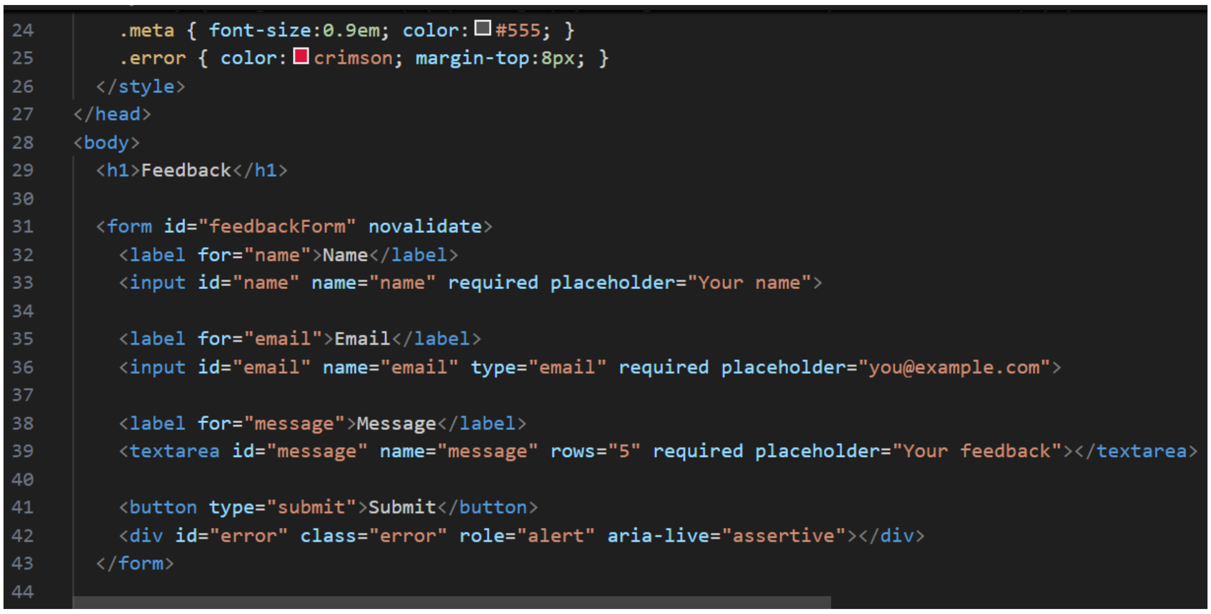
"Generate a simple HTML feedback form (name, email, message) that uses JavaScript to show the submitted feedback below the form."

Code Generated:

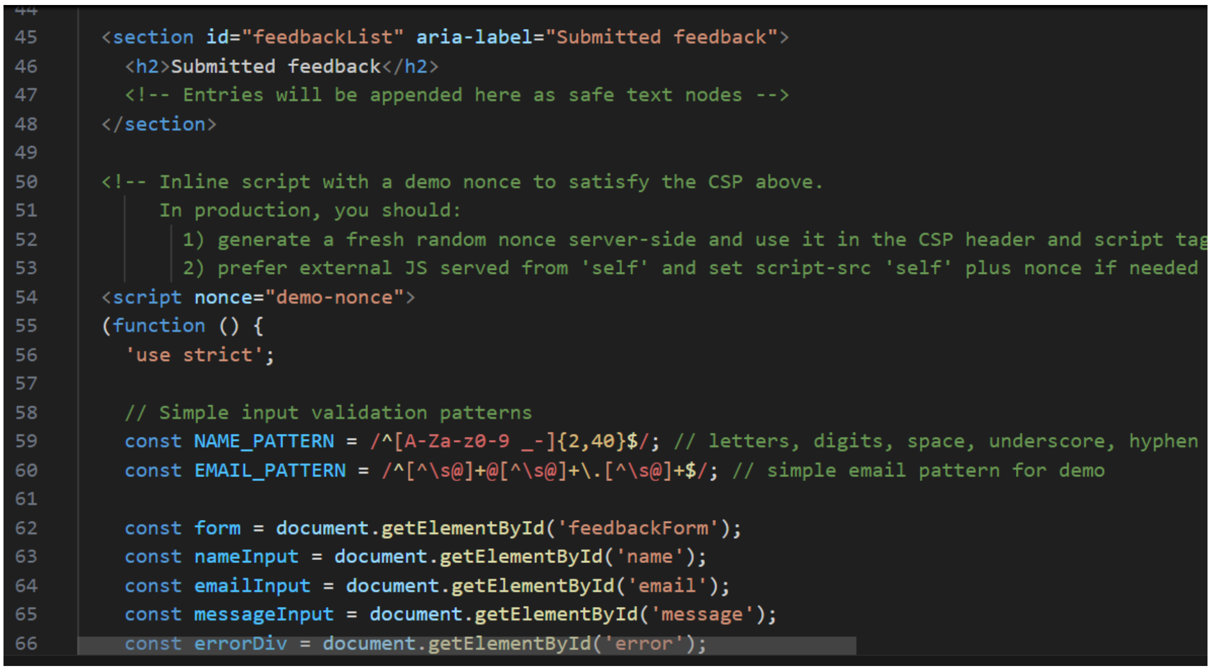


20.3_t3.html > ...

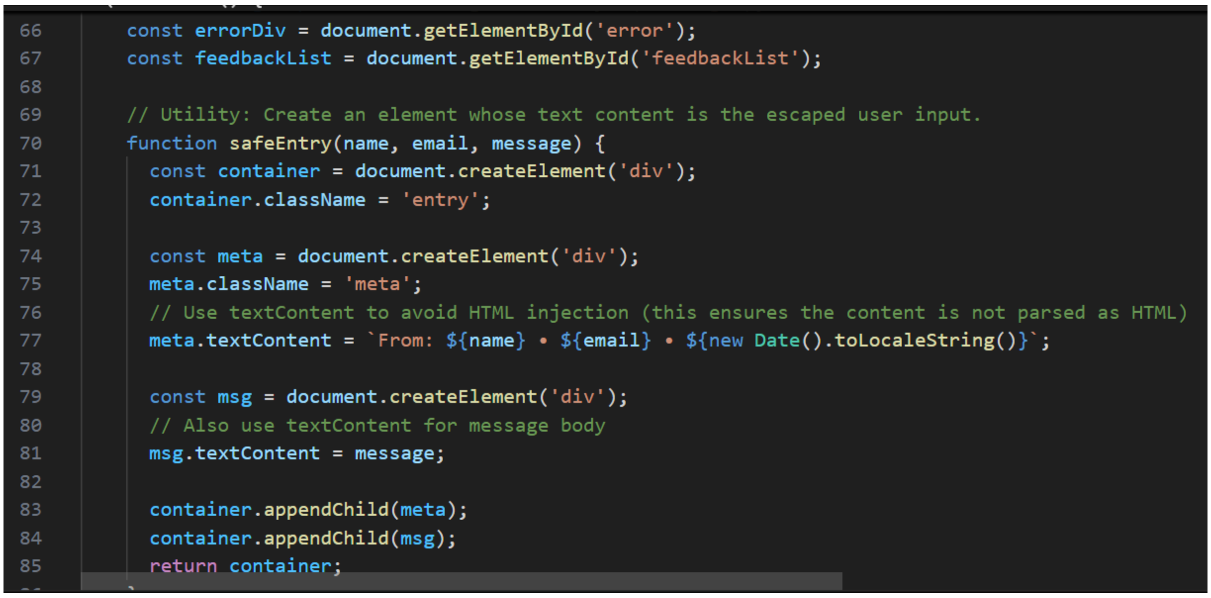
```html
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width,initial-scale=1" />

  <!-- Content Security Policy:
       - default-src 'self' (only resources from same origin)
       - script-src 'nonce-demo-nonce' allows this inline script because it has the same nonce.
       NOTE: In production, generate a fresh nonce server-side per response and place scripts in exte
  <meta http-equiv="Content-Security-Policy"
        content="default-src 'self'; script-src 'nonce-demo-nonce'; object-src 'none'; connect-src 's

  <title>Secure Feedback Form</title>

  <style>
    body { font-family: system-ui, -apple-system, "Segoe UI", Roboto, sans-serif; padding: 24px; }
    form { max-width: 520px; margin-bottom: 18px; }
    label { display:block; margin-top:10px; font-weight:600; }
    input, textarea { width:100%; padding:8px; box-sizing:border-box; }
    button { margin-top:12px; padding:8px 12px; }
    #feedbackList { border-top:1px solid #ddd; padding-top:12px; margin-top:12px; }
    .entry { margin-bottom:10px; padding:8px; background:#f9f9f9; border-radius:6px; }
    .meta { font-size:0.9em; color:#555; }
```



```html
    .meta { font-size:0.9em; color:#555; }
    .error { color:crimson; margin-top:8px; }
  </style>
</head>
<body>
  <h1>Feedback</h1>

  <form id="feedbackForm" novalidate>
    <label for="name">Name</label>
    <input id="name" name="name" required placeholder="Your name">

    <label for="email">Email</label>
    <input id="email" name="email" type="email" required placeholder="you@example.com">

    <label for="message">Message</label>
    <textarea id="message" name="message" rows="5" required placeholder="Your feedback"></textarea>

    <button type="submit">Submit</button>
    <div id="error" class="error" role="alert" aria-live="assertive"></div>
  </form>

```

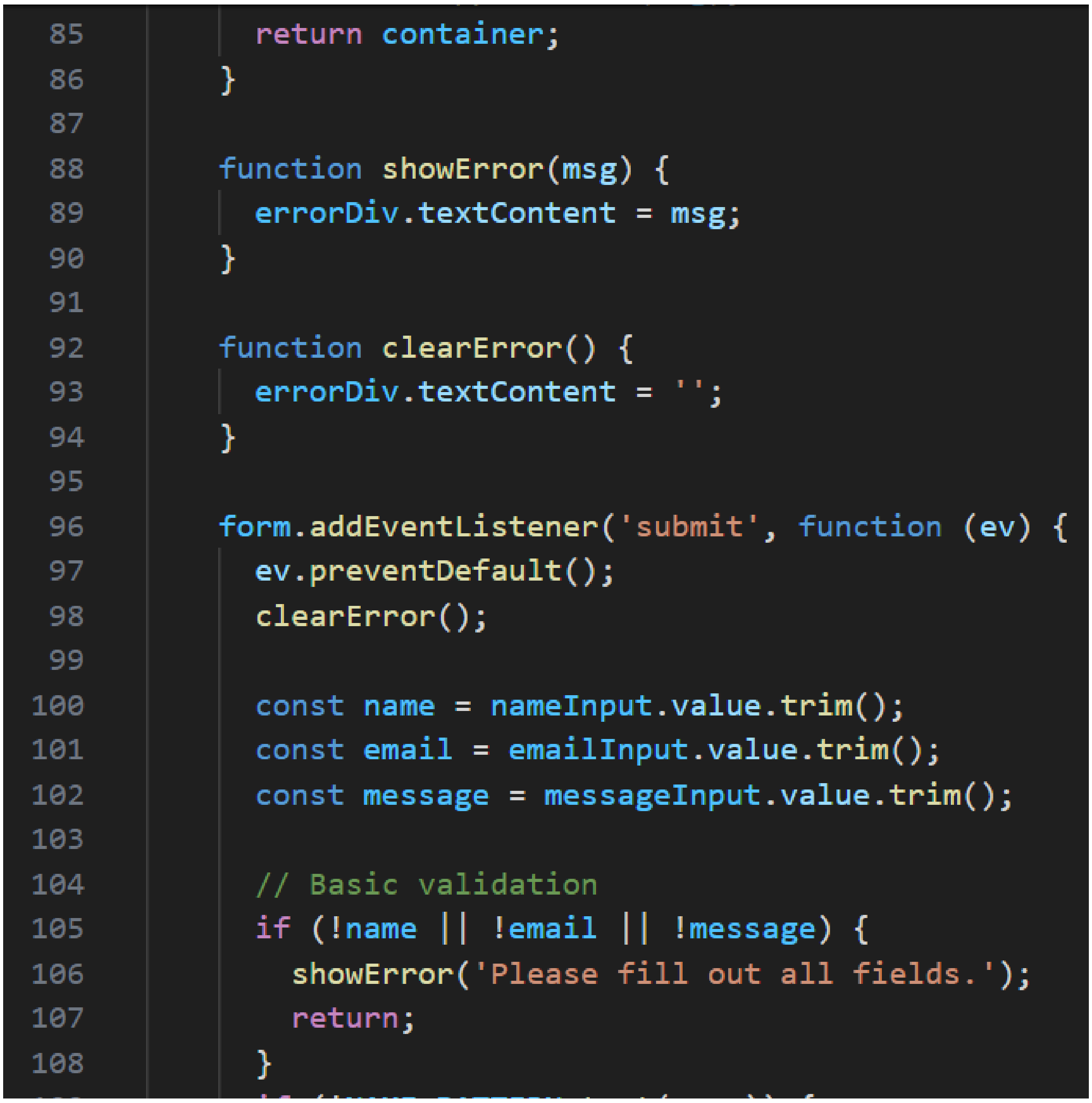


```
<section id="feedbackList" aria-label="Submitted feedback">
  <h2>Submitted feedback</h2>
  <!-- Entries will be appended here as safe text nodes -->
</section>

<!-- Inline script with a demo nonce to satisfy the CSP above.
     In production, you should:
       1) generate a fresh random nonce server-side and use it in the CSP header and script tag
       2) prefer external JS served from 'self' and set script-src 'self' plus nonce if needed
<script nonce="demo-nonce">
(function () {
  'use strict';

  // Simple input validation patterns
  const NAME_PATTERN = /^[A-Za-z0-9 _-]{2,40}$/; // letters, digits, space, underscore, hyphen
  const EMAIL_PATTERN = /^[^\s@]+@[^\s@]+\.[^\s@]+$/; // simple email pattern for demo

  const form = document.getElementById('feedbackForm');
  const nameInput = document.getElementById('name');
  const emailInput = document.getElementById('email');
  const messageInput = document.getElementById('message');
  const errorDiv = document.getElementById('error');
```

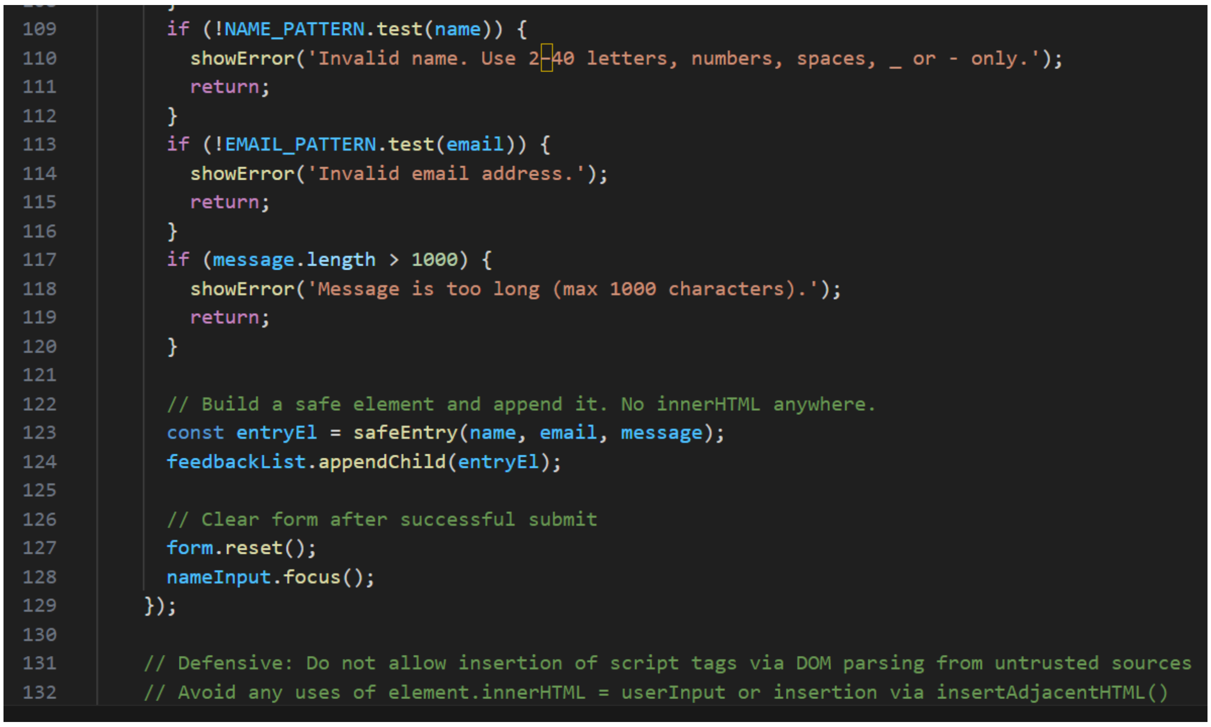


const errorDiv = document.getElementById('error');

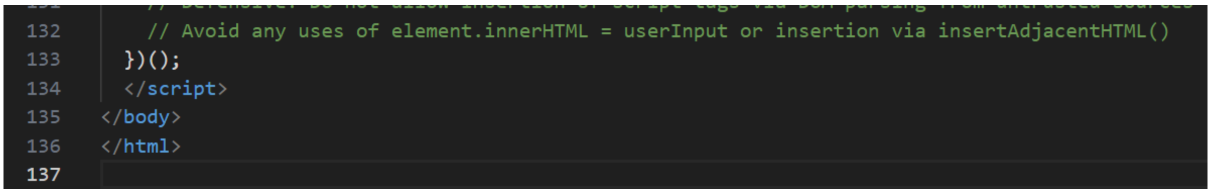
```
  const feedbackList = document.getElementById('feedbackList');

  // Utility: Create an element whose text content is the escaped user input.
  function safeEntry(name, email, message) {
    const container = document.createElement('div');
    container.className = 'entry';

    const meta = document.createElement('div');
    meta.className = 'meta';
    // Use textContent to avoid HTML injection (this ensures the content is not parsed as HTML)
    meta.textContent = `From: ${name} • ${email} • ${new Date().toLocaleString()}`;

    const msg = document.createElement('div');
    // Also use textContent for message body
    msg.textContent = message;

    container.appendChild(meta);
    container.appendChild(msg);
    return container;
```



```javascript
    return container;
  }

  function showError(msg) {
    errorDiv.textContent = msg;
  }

  function clearError() {
    errorDiv.textContent = '';
  }

  form.addEventListener('submit', function (ev) {
    ev.preventDefault();
    clearError();

    const name = nameInput.value.trim();
    const email = emailInput.value.trim();
    const message = messageInput.value.trim();

    // Basic validation
    if (!name || !email || !message) {
      showError('Please fill out all fields.');
      return;
    }
```



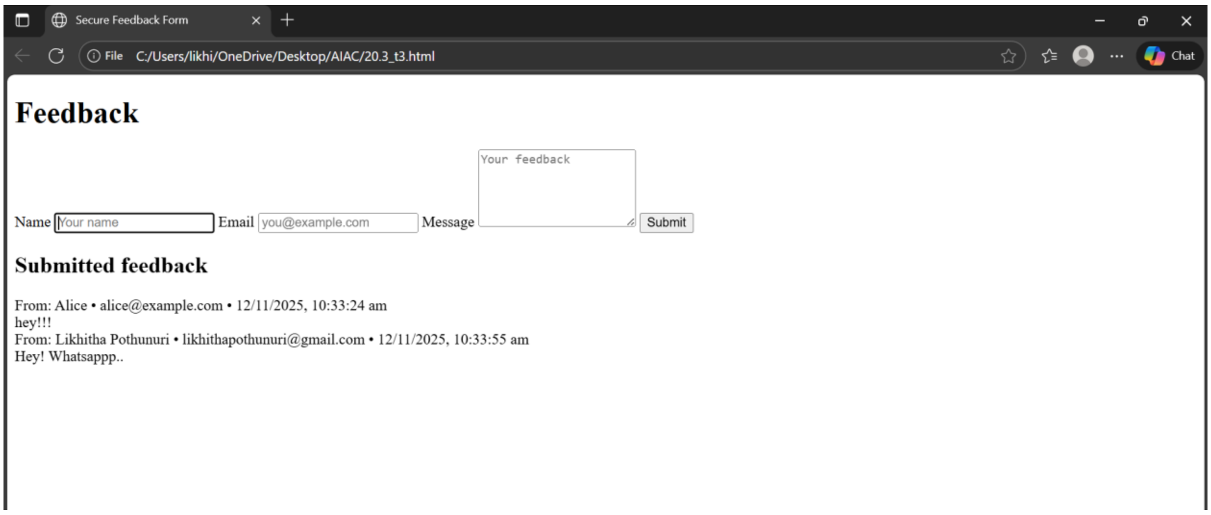
```
      if (!NAME_PATTERN.test(name)) {
        showError('Invalid name. Use 2-40 letters, numbers, spaces, _ or - only.');
        return;
      }
      if (!EMAIL_PATTERN.test(email)) {
        showError('Invalid email address.');
        return;
      }
      if (message.length > 1000) {
        showError('Message is too long (max 1000 characters).');
        return;
      }

      // Build a safe element and append it. No innerHTML anywhere.
      const entryEl = safeEntry(name, email, message);
      feedbackList.appendChild(entryEl);

      // Clear form after successful submit
      form.reset();
      nameInput.focus();
    });

    // Defensive: Do not allow insertion of script tags via DOM parsing from untrusted sources
    // Avoid any uses of element.innerHTML = userInput or insertion via insertAdjacentHTML()
```



```
    // Avoid any uses of element.innerHTML = userInput or insertion via insertAdjacentHTML()
```

```
  })();
  </script>
</body>
</html>
```

Output:



Feedback

Name [Your name] Email [you@example.com] Message [Your feedback] Submit

Submitted feedback

From: Alice • alice@example.com • 12/11/2025, 10:33:24 am
hey!!!
From: Likhitha Pothunuri • likhithapothunuri@gmail.com • 12/11/2025, 10:33:55 am
Hey! Whatsappp..

Observation:

1. The original AI-generated feedback form was **vulnerable to XSS** because it directly displayed user input using innerHTML.

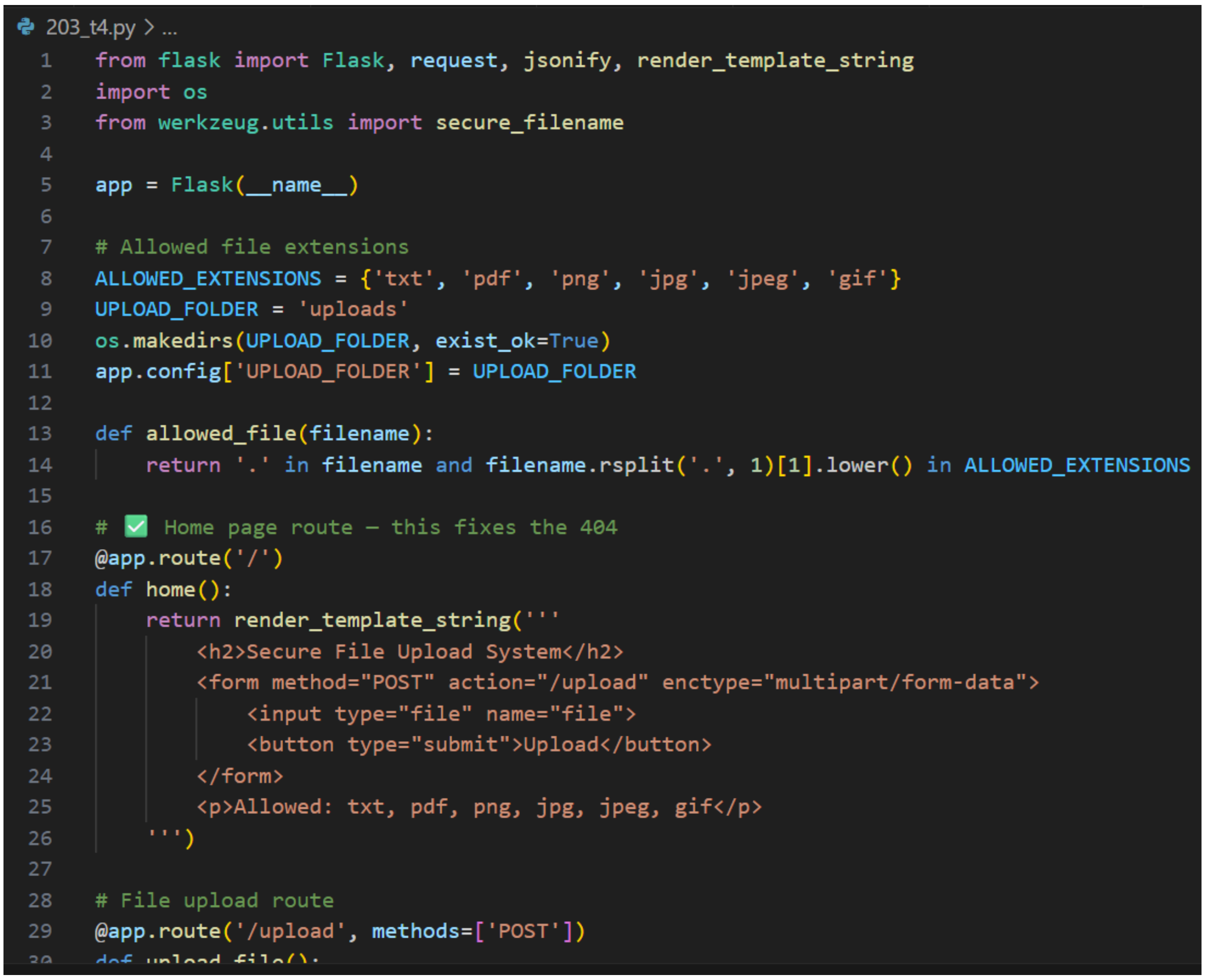
2. The secure version fixes this by using **textContent** to safely display input, **validating user data**, and adding a **Content Security Policy (CSP)**.
3. This ensures that any malicious scripts entered by a user will **not be executed**, keeping the webpage safe.

Task 4 – Real-Time Application: Security Audit of AI-Generated

Prompt:

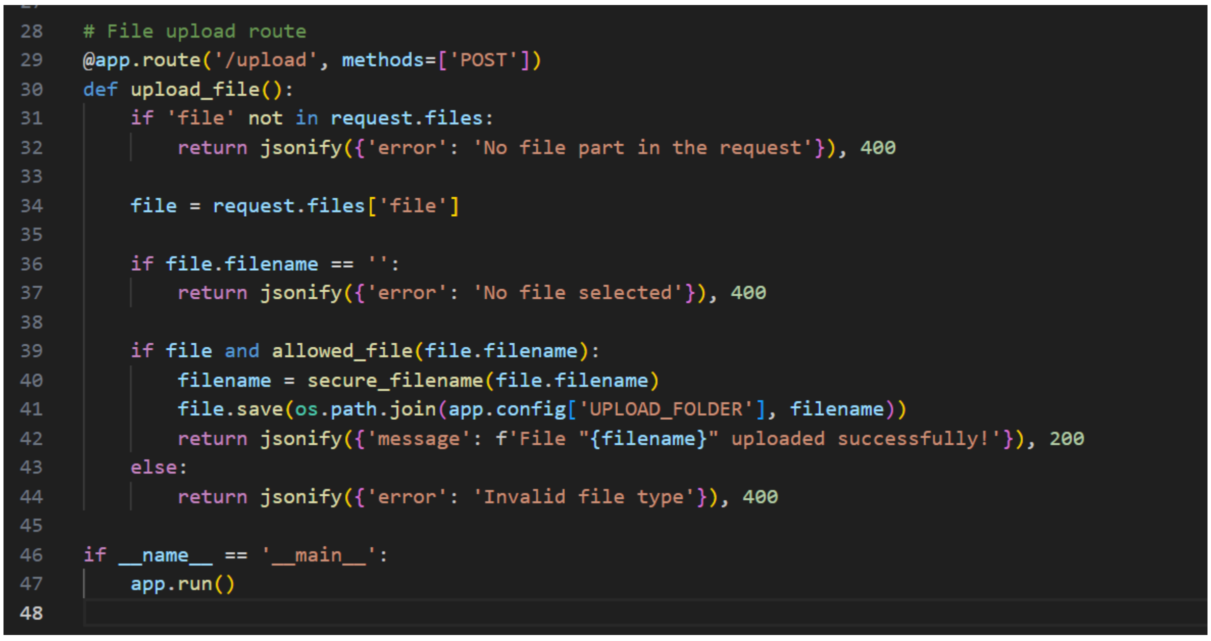
“Generate a simple Python Flask file upload program that saves uploaded files to a folder.”

Code Generated:



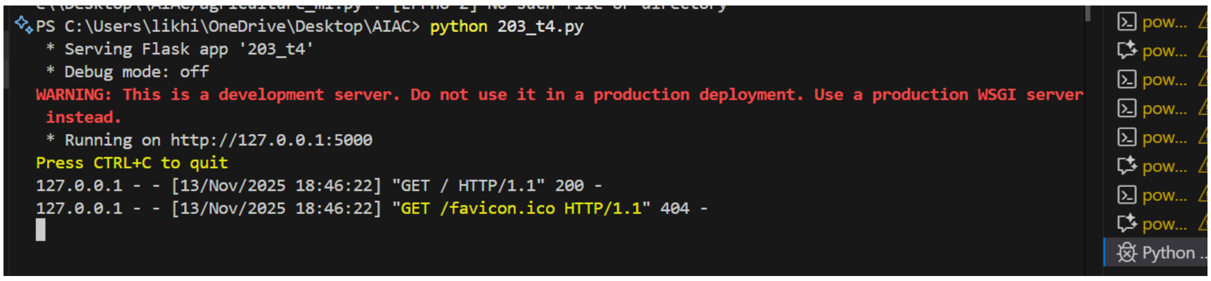
203_t4.py > ...

```python
from flask import Flask, request, jsonify, render_template_string
import os
from werkzeug.utils import secure_filename

app = Flask(__name__)

# Allowed file extensions
ALLOWED_EXTENSIONS = {'txt', 'pdf', 'png', 'jpg', 'jpeg', 'gif'}
UPLOAD_FOLDER = 'uploads'
os.makedirs(UPLOAD_FOLDER, exist_ok=True)
app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER

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

# ✅ Home page route — this fixes the 404
@app.route('/')
def home():
    return render_template_string('''
        <h2>Secure File Upload System</h2>
        <form method="POST" action="/upload" enctype="multipart/form-data">
            <input type="file" name="file">
            <button type="submit">Upload</button>
        </form>
        <p>Allowed: txt, pdf, png, jpg, jpeg, gif</p>
    ''')

# File upload route
@app.route('/upload', methods=['POST'])
```

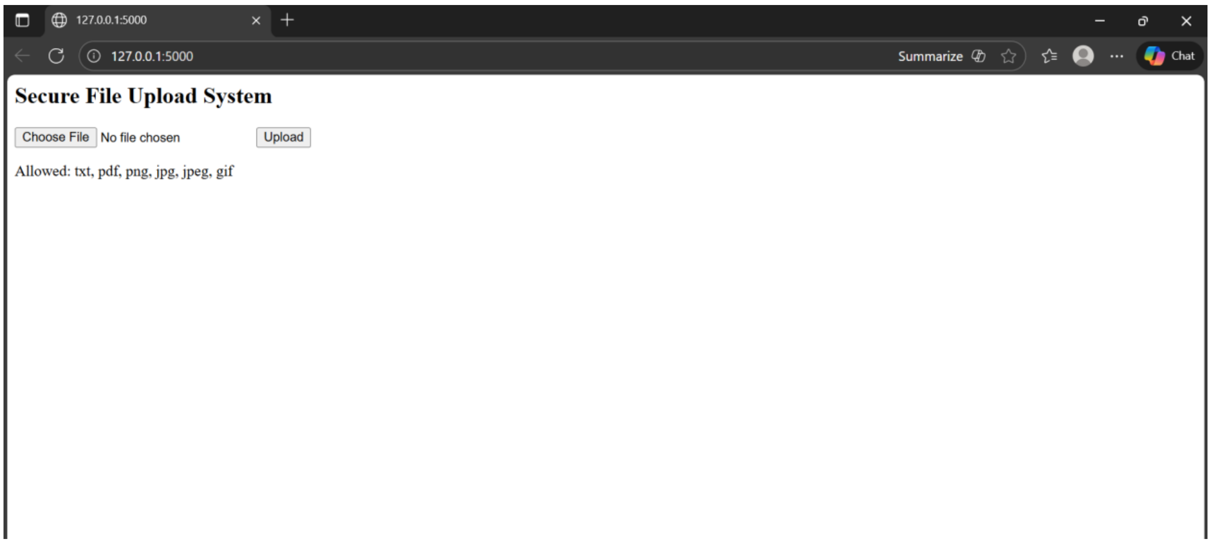


```python
# File upload route
@app.route('/upload', methods=['POST'])
def upload_file():
    if 'file' not in request.files:
        return jsonify({'error': 'No file part in the request'}), 400

    file = request.files['file']

    if file.filename == '':
        return jsonify({'error': 'No file selected'}), 400

    if file and allowed_file(file.filename):
        filename = secure_filename(file.filename)
        file.save(os.path.join(app.config['UPLOAD_FOLDER'], filename))
        return jsonify({'message': f'File "{filename}" uploaded successfully!'}), 200
    else:
        return jsonify({'error': 'Invalid file type'}), 400

if __name__ == '__main__':
    app.run()

```

Output:



```
PS C:\Users\likhi\OneDrive\Desktop\AIAC> python 203_t4.py
 * Serving Flask app '203_t4'
 * Debug mode: off
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
127.0.0.1 - - [13/Nov/2025 18:46:22] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [13/Nov/2025 18:46:22] "GET /favicon.ico HTTP/1.1" 404 -
```



Secure File Upload System

Allowed: txt, pdf, png, jpg, jpeg, gif

Observation:

1. The insecure version directly saved user-uploaded files without any validation, making it vulnerable to **path traversal** and **malicious file uploads**.
2. The secure version fixes these issues by using **secure_filename()**, validating file types and sizes, and disabling **debug mode**, ensuring that uploaded files are stored safely and securely.