AI-Enhanced Code Quality Analysis Tool: Development Plan

Prepared for Assignment 104
Date: May 21, 2025

1 Project Overview

This document outlines the development plan for an AI-powered Python code quality analysis tool that integrates traditional static analysis (`pylint`, `flake8`, `radon`) with LLM-based insights using OpenAI's APIs (`text-embedding-3-small`, `gpt-4o-mini`). The tool allows users to upload Python files, analyzes code for quality issues, predicts bugs, generates optimized code, and provides a conversational, chatbot-like interface for iterative modifications without resubmitting code. It features a responsive web interface (Flask, HTML, Tailwind CSS), SQLite for session and log persistence, and bonus features (readability scoring, complexity visualization). The project will be hosted on GitHub with a comprehensive `README.md`.

2 Project Structure

The project follows a modular structure:

```
ai-code-analyzer/
 app/
    static/
       css/              % Tailwind CSS (via CDN or compiled)
       js/               % JavaScript for Plotly and chat UI
    templates/            % HTML templates for UI
    routes.py             % Flask route definitions
    analyzer.py           % Static code analysis module
    ai_helper.py          % AI-based analysis and optimization
    readability.py        % AI-based maintainability scoring (
       bonus)
    visualize.py          % Complexity visualization (bonus)
 tests/                    % Unit tests for analyzer and AI
    modules
 uploads/                  % Temporary storage for uploaded files
 logs/                     % SQLite database for sessions and
    logs
 app.py                    % Flask application entry point
 requirements.txt          % Python dependencies
 README.md                 % Setup instructions and project info
 .env                      % Environment variables (
    OPENAI_API_KEY)
 .gitignore                % Excludes uploads/, __pycache__/, .
    env
```

3 Core Functionalities

3.1 File Upload and Conversational Interface (HTML + Tailwind + Flask)

The tool provides a web-based interface for uploading Python files and interacting conversationally:

- **Purpose**: Allow users to upload `.py` files, view analysis results, and iteratively modify optimized code via a chatbot-like interface without resubmitting code.
- **Implementation**: Use Flask to serve an HTML form styled with Tailwind CSS (via CDN). The form POSTs files to `/analyze`. A chat-style UI displays conversation history (original code, analysis, optimized code, user commands) and accepts commands via `/regenerate`. Example HTML:

```
<div class="chat-container flex flex-col h-screen p-4">
    <div class="chat-history flex-1 overflow-y-auto">
        <!-- Messages: original code, AI analysis, optimized code, user commands -->
    </div>
    <form action="/regenerate" method="POST" class="flex gap-2">
        <input type="hidden" name="session_id" value="{{ session_id }}">
        <input type="text" name="user_command" class="flex-1 p-2 rounded">
        <button class="bg-blue-500 text-white p-2 rounded">Submit</button>
    </form>
</div>
```

- **Responsiveness**: Use Tailwinds responsive classes (e.g., `md:grid-cols-2`, `sm:p-2`) for mobile and desktop compatibility. Test with browser developer tools.
- **Error Handling**: Validate file type (`.py`), size (<5MB), and content (valid Python). Display user-friendly errors. Example:

```
from flask import request, jsonify
import os
import uuid
import ast
def validate_python_file(file_path):
    try:
        with open(file_path, 'r') as f:
            ast.parse(f.read())
        return True
    except SyntaxError:
        return False
@app.route('/analyze', methods=['POST'])
def analyze():
    file = request.files.get('file')
    if not file or not file.filename.endswith('.py'):
```

```
        return jsonify({"error": "Invalid file: Only .py files allowed"}), 400
    if os.path.getsize(file.stream) > 5 * 1024 * 1024:
        return jsonify({"error": "File too large (max 5MB)"}), 400
    file_path = os.path.join('uploads', f"{uuid.uuid4()}.py")
    file.save(file_path)
    if not validate_python_file(file_path):
        return jsonify({"error": "Invalid Python code"}), 400
    session_id = str(uuid.uuid4())
    # Save session and run analysis
```

3.2 Static Code Analysis (analyzer.py)

The static analysis module evaluates code quality:

- **Tools**: `pylint` (style violations), `flake8` (PEP 8), `radon` (complexity, maintainability).
- **Functionality**: Parse files with `ast.parse()`. Extract metrics (complexity, line count, comment ratio, style issues). Output:

```
{
    "complexity": {
        "functions": [{"name": "foo", "complexity": 3}, ...],
        "module_complexity": 10
    },
    "style_issues": [{"line": 5, "message": "Missing docstring"}, ...],
    "warnings": [{"line": 10, "message": "Unused variable"}, ...]
}
```

- **Error Handling**:

```
import ast
def analyze_code(code):
    try:
        ast.parse(code)
    except SyntaxError as e:
        return {"error": f"Invalid Python code: {str(e)}"}
    # Run pylint, flake8, radon
```

3.3 AI Code Insight Engine (ai_helper.py)

The AI module uses OpenAIs APIs for analysis and iterative optimization:

- **API Key Usage**: Store in `.env`:

```
OPENAI_API_KEY=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
```

Load with:

```
import os
import openai
from dotenv import load_dotenv
load_dotenv()
openai.api_key = os.getenv("OPENAI_API_KEY")
```

- **Models**: `gpt-4o-mini` for chat completion (analysis, optimization); `text-embedding-3-small` for pattern analysis.
- **Conversational Analysis**: Analyze code and generate optimized code:

```
from tenacity import retry, stop_after_attempt,
    wait_exponential
@retry(stop=stop_after_attempt(3), wait=wait_exponential(
    multiplier=1, min=4, max=10))
def analyze_code_with_ai(code, session_id):
    if not validate_code_content(code):
        return {"error": "Code contains potentially harmful
            content"}
    response = openai.ChatCompletion.create(
        model="gpt-4o-mini",
        messages=[
            {
                "role": "system",
                "content": (
                    "Analyze the following Python code for
                        potential bugs, maintenance issues,
                        and refactoring opportunities. "
                    "Provide: 1. A list of predicted bugs (
                        with line numbers and explanations). "
                    "2. Refactoring suggestions with example
                        code snippets. "
                    "3. An optimized version of the code. "
                    "4. Rank issues by severity (high, medium
                        , low). "
                    "Do not invent issues. Code: ```{code}```
                        "
                )
            }
        ],
        max_tokens=1500,
        temperature=0.3
    )
    result = response.choices[0].message.content
    save_session(session_id, code, result, result)  % Save
        original and optimized code
    return result
```

- **Iterative Optimization**: Handle user commands for modifying optimized code:

```python
@retry(stop=stop_after_attempt(3), wait=wait_exponential(multiplier=1, min=4, max=10))
def regenerate_code(session_id, user_command):
    session = load_session(session_id)
    if not session:
        return {"error": "Session not found"}
    original_code = session["original_code"]
    optimized_code = session["optimized_code"]
    history = session["conversation_history"]
    prompt = (
        f"You are a Python code optimization assistant. Original code: ```{original_code}```. "
        f"Previous optimized code: ```{optimized_code}```. "
        f"Conversation history: {history}. "
        f"User request: '{user_command}'. "
        "Provide an updated optimized version of the code and an explanation of changes."
    )
    response = openai.ChatCompletion.create(
        model="gpt-4o-mini",
        messages=[{"role": "system", "content": prompt}],
        max_tokens=1500,
        temperature=0.3
    )
    result = response.choices[0].message.content
    update_session(session_id, result, user_command)
    return result
```

- **Output**: Structured JSON:

```json
{
    "predicted_bugs": [
        {"line": 15, "issue": "Potential NoneType error", "severity": "high", "explanation": "..."}
    ],
    "refactor_suggestions": [
        {"line": 20, "original": "for i in range(len(lst))", "suggested": "for item in lst", "explanation": "..."}
    ],
    "optimized_code": "def foo(): ...",
    "impact_ranking": [
        {"issue": "NoneType error", "severity": "high", "impact": "Could cause runtime crash"}
    ]
}
```

- **Content Validation**:

```python
def validate_code_content(code):
    return "eval(" not in code and "exec(" not in code
```

3.4 Flask Routes (routes.py)

Endpoints support the conversational workflow:

- `/`: Home page with upload form.
- `/analyze`: Processes file upload, creates session, runs analysis.
- `/regenerate`: Accepts user commands, updates optimized code.
- `/readability`: Displays readability score (bonus).
- `/visualize`: Renders complexity visualization (bonus).
- Example for regeneration:

```
@app.route('/regenerate', methods=['POST'])
def regenerate():
    session_id = request.form.get('session_id')
    user_command = request.form.get('user_command')
    if not session_id or not user_command:
        return jsonify({"error": "Missing session_id or
           command"}), 400
    result = regenerate_code(session_id, user_command)
    return jsonify({"response": result})
```

4 Bonus Features

4.1 Readability Scoring (readability.py)

Evaluates code maintainability:

- **Purpose**: Rate readability on a 1-10 scale using `gpt-4o-mini`.
- **Implementation**:

```
def get_readability_score(code):
    if not validate_code_content(code):
        return {"error": "Invalid code content"}
    response = openai.ChatCompletion.create(
        model="gpt-4o-mini",
        messages=[
            {
                "role": "system",
                "content": (
                    "Evaluate the readability and
                       maintainability of this Python code on
                        a scale of 1-10. "
                    "Consider factors like code clarity,
                       structure, naming conventions, and
                       comments. "
                    "Provide a score and a brief
                       justification. "
                    f"Code: ```{code}```"
                )
```

```
            }
        ],
        max_tokens=200
    )
    return response.choices[0].message.content
```

- **Output**: Tailwind-styled progress bar:

```
{
    "score": 7,
    "justification": "Clear structure but lacks docstrings."
}
```

4.2 Complexity Visualization (visualize.py)

Displays function-level complexity:

- **Purpose**: Compare original and AI-optimized code complexity.
- **Implementation**: Use `radon` and Plotly:

```
{
    "type": "bar",
    "data": {
        "labels": [], % Function names
        "datasets": [
            {
                "label": "Original Complexity",
                "data": [], % Complexity values
                "backgroundColor": "rgba(75, 192, 192, 0.6)",
                "borderColor": "rgba(75, 192, 192, 1)",
                "borderWidth": 1
            },
            {
                "label": "AI-Refactored Complexity",
                "data": [], % Refactored complexity
                "backgroundColor": "rgba(255, 99, 132, 0.6)",
                "borderColor": "rgba(255, 99, 132, 1)",
                "borderWidth": 1
            }
        ]
    },
    "options": {
        "scales": {
            "y": { "beginAtZero": true, "title": { "display":
                true, "text": "Cyclomatic Complexity" } },
            "x": { "title": { "display": true, "text": "
               Functions" } }
        },
        "plugins": {
            "legend": { "display": true },
            "title": { "display": true, "text": "Code
               Complexity Comparison" }
```

```
        }
    }
}
```

5 Tech Stack

- **Backend**: Python 3.8+, Flask
- **Frontend**: HTML, Tailwind CSS (via CDN), JavaScript (Plotly)
- **Static Analysis**: `pylint`, `flake8`, `radon`
- **AI Engine**: OpenAI API (`text-embedding-3-small`, `gpt-4o-mini`)
- **Visualization**: Plotly (via CDN)
- **Database**: SQLite for sessions and logs
- **Testing**: `pytest`, Postman
- **Version Control**: GitHub

- **Environment**: `.env` for $\texttt{OPENAI}_A PI_K EY$

6 Database Usage

SQLite stores session data and logs:

- **Purpose**: Persist conversation history, code, and analysis results.
- **Schema**:

```
CREATE TABLE sessions (
    session_id TEXT PRIMARY KEY,
    created_at DATETIME,
    original_code TEXT,
    analysis_results TEXT,
    optimized_code TEXT,
    conversation_history TEXT
);
```

- **Implementation**:

```
import sqlite3
import hashlib
import json
import uuid
def save_session(session_id, original_code, analysis_results,
    optimized_code):
    conn = sqlite3.connect('logs/analyzer.db')
    conn.execute(
        "INSERT INTO sessions (session_id, created_at,
          original_code, analysis_results, optimized_code,
          conversation_history) "
```

```
        "VALUES (?, datetime('now'), ?, ?, ?, ?)",
        (session_id, original_code, json.dumps(
            analysis_results), optimized_code, json.dumps([]))
    )
    conn.commit()
    conn.close()
def update_session(session_id, new_optimized_code,
    user_command):
    conn = sqlite3.connect('logs/analyzer.db')
    cursor = conn.cursor()
    cursor.execute("SELECT conversation_history FROM sessions
        WHERE session_id = ?", (session_id,))
    history = json.loads(cursor.fetchone()[0])
    history.append({"user_command": user_command, "response":
        new_optimized_code})
    cursor.execute(
        "UPDATE sessions SET optimized_code = ?,
            conversation_history = ? WHERE session_id = ?",
        (new_optimized_code, json.dumps(history), session_id)
    )
    conn.commit()
    conn.close()
```

7 Testing and Validation

- **Unit Tests** (`tests/`):

```
import pytest
from app.analyzer import analyze_code
def test_static_analysis():
    code = "def foo(): pass"
    result = analyze_code(code)
    assert "complexity" in result
    assert len(result["style_issues"]) == 1
```

- **API Testing**: Use Postman to test endpoints (`/analyze`, `/regenerate`, etc.).
- **UI Testing**: Verify chat UI responsiveness on mobile and desktop.

8 Documentation and Git Workflow

8.1 README Structure

The `README.md` includes:

- **Project Title & Description**: AI-powered code quality tool with conversational interface.
- **Installation**:
  1. Clone: `git clone <repo-link>`

2. Install: `pip install -r requirements.txt`

3. Set `.env`: $\texttt{OPENAI}_A PI_K EY = sk{-}proj{-}...Initialize SQLite$: `python -c "from app import in`

4. **Features**: Static analysis, AI optimization, conversational modifications, bonus features.

- **Sample Output**:

```
{
    "complexity": {"module_complexity": 10, "functions": [{"
       name": "foo", "complexity": 3}]},
    "predicted_bugs": [{"line": 15, "issue": "Potential
       NoneType error", "severity": "high"}],
    "optimized_code": "def foo(): ...",
    "readability": {"score": 7, "justification": "Clear
       structure but lacks docstrings"}
}
```

- **Technologies**: Python, Flask, Tailwind CSS, OpenAI API, SQLite.
- **API Notes**: Use provided key responsibly, avoid harmful content.

8.2 Git Workflow

- Initialize: `git init`
- `.gitignore`:

```
__pycache__/
uploads/
logs/
.env
*.pyc
```

- Commit messages: e.g., `Add conversational AI with session persistence`.

9 Submission Requirements

- **Deliverables**: Functional code, conversational interface, bonus features, GitHub repo or zipped folder.
- **Evaluation Alignment**:
  - **Implementation**: All features plus conversational workflow.
  - **Code Organization**: Modular structure.
  - **API Integration**: Secure OpenAI API usage.
  - **UI Design**: Responsive chat-style UI.
  - **Error Handling**: File, code, and command validation.
  - **Creativity**: Conversational interface and bonus features.

10 Additional Notes

- **API Key Usage**:
  - Use `sk-proj-wL`$_R vm7SK5qkcAnB8JW-nzPtpClV9L35rEuZmUHzNfHxieq0XkEi6G7$
    $JXTayA_Z uT3BlbkFJeySbrCpb06ytPMnxCqQTSeoveVwkcQ_k q-HSogQeWTxjl0Q-$
    $9c64jZANfCUdlNT5fFkvsp5GMAfor$`gpt-4o-mini`$and$`text-embedding-3-small`.$Store$
  - Validate code to avoid harmful content (e.g., `eval`, `exec`).
  - Cache responses in SQLite to reduce API calls.
- **Conversational Workflow**:
  - Store session data (code, analysis, history) in SQLite.
  - Use `session`$_i d(viacookiesorURL)totrackconversations.Exampleusercommand$: "$Uselistcomprehensioninsteadofloop"updatesoptimizedcode.$
- **Security**:
  - Sanitize uploads to prevent directory traversal.
  - Limit file size to 5MB.
  - Validate user commands for relevance.
- **Challenges**:
  - **API Rate Limits**: Use `tenacity` and caching.
  - **Response Parsing**: Parse `gpt-4o-mini` responses into structured JSON.
  - **Conversation Persistence**: Ensure session data is reliably stored and retrieved.
- **Development Tips**:
  - Start with static analysis and file upload.
  - Implement conversational AI incrementally, testing prompts.
  - Use Postman for API debugging.
  - Document edge cases in `README`.

11 Implementation Notes

- **Security**: Sanitize uploads, exclude `.env`, validate code and commands.
- **Performance**: Cache AI responses, limit file size.
- **Scalability**: Use Flask for development, Gunicorn for production.