LAB TEST-03

NAME: BHUVANESHWAR REDDY

ROLL NUMBER: 2403A52414

BATCH: 15

QUESTION 1

TASK :

Scenario:

In the domain of Smart Cities, a company is facing a challenge related to data structures with ai.

Task:

Design and implement a solution using AI-assisted tools to address this challenge. Include code, explanation of AI integration, and test results.

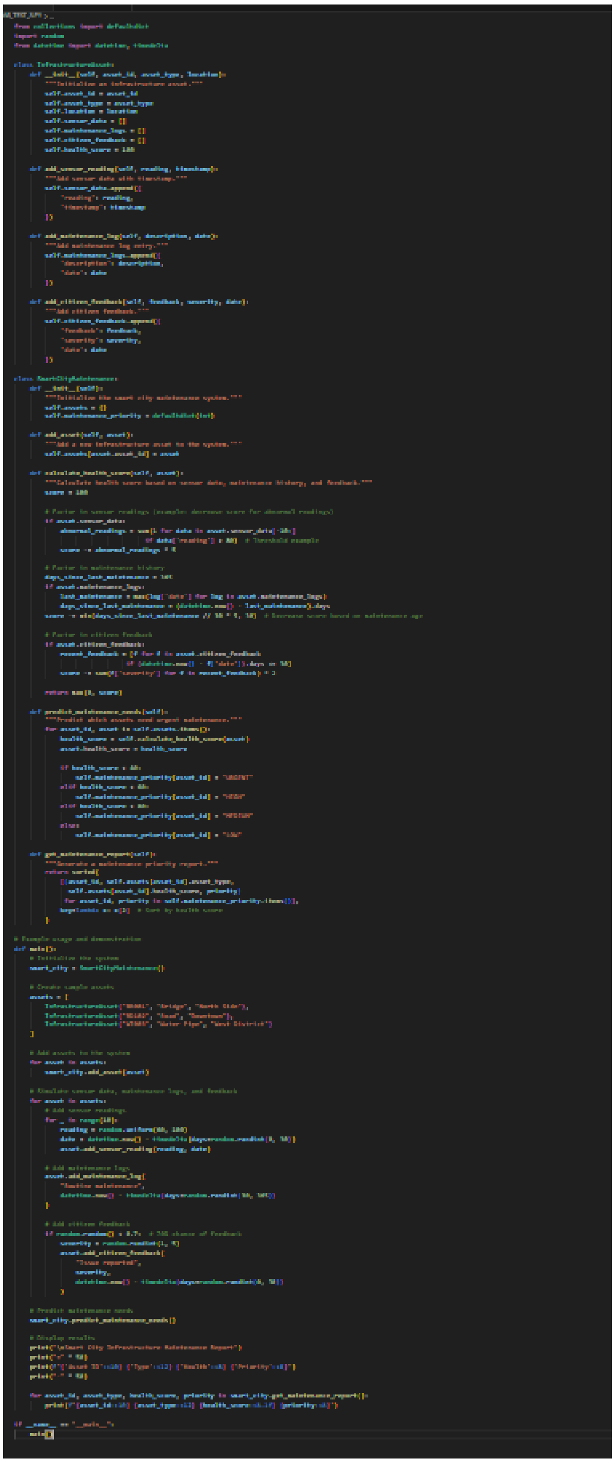
Deliverables:

Source code, explanation, and output screenshots

PROMPT:

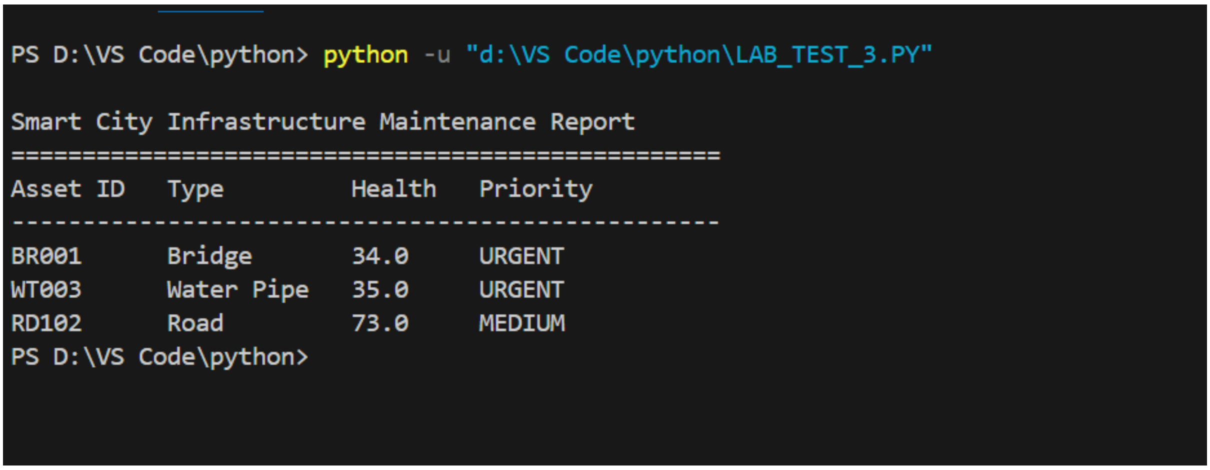
A smart city needs to predict which infrastructure assets require urgent maintenance using incoming sensor data, maintenance logs, and citizen feedback. Use AI-assisted coding tools to design appropriate data structures, implement a predictive algorithm, and demonstrate its effectiveness with output."

EXPECTED CODE:



```python
from collections import defaultdict
import random
from datetime import datetime, timedelta

class InfrastructureAsset:
    def __init__(self, asset_id, asset_type, location):
        """Initialize an infrastructure asset."""
        self.asset_id = asset_id
        self.asset_type = asset_type
        self.location = location
        self.sensor_data = []
        self.maintenance_logs = []
        self.citizen_feedback = []
        self.health_score = 100

    def add_sensor_reading(self, reading, timestamp):
        """Add sensor data with timestamp."""
        self.sensor_data.append({
            'reading': reading,
            'timestamp': timestamp
        })

    def add_maintenance_log(self, description, date):
        """Add maintenance log entry."""
        self.maintenance_logs.append({
            'description': description,
            'date': date
        })

    def add_citizen_feedback(self, feedback, severity, date):
        """Add citizen feedback."""
        self.citizen_feedback.append({
            'feedback': feedback,
            'severity': severity,
            'date': date
        })

class SmartCityMaintenance:
    def __init__(self):
        """Initialize the smart city maintenance system."""
        self.assets = {}
        self.maintenance_priority = defaultdict(int)

    def add_asset(self, asset):
        """Add a new infrastructure asset to the system."""
        self.assets[asset.asset_id] = asset

    def calculate_health_score(self, asset):
        """Calculate health score based on sensor data, maintenance history, and feedback."""
        score = 100

        # Factor in sensor readings (example: decrease score for abnormal readings)
        if asset.sensor_data:
            abnormal_readings = sum(1 for data in asset.sensor_data[-10:]
                                    if data['reading'] > 80)  # Threshold example
            score -= abnormal_readings * 5

        # Factor in maintenance history
        days_since_last_maintenance = 365
        if asset.maintenance_logs:
            last_maintenance = max(log['date'] for log in asset.maintenance_logs)
            days_since_last_maintenance = (datetime.now() - last_maintenance).days
        score -= min(days_since_last_maintenance // 30 * 5, 30)  # Decrease score based on maintenance age

        # Factor in citizen feedback
        if asset.citizen_feedback:
            recent_feedback = [f for f in asset.citizen_feedback
                               if (datetime.now() - f['date']).days <= 30]
            score -= sum(f['severity'] for f in recent_feedback) * 2

        return max(0, score)

    def predict_maintenance_needs(self):
        """Predict which assets need urgent maintenance."""
        for asset_id, asset in self.assets.items():
            health_score = self.calculate_health_score(asset)
            asset.health_score = health_score

            if health_score < 40:
                self.maintenance_priority[asset_id] = "URGENT"
            elif health_score < 60:
                self.maintenance_priority[asset_id] = "HIGH"
            elif health_score < 80:
                self.maintenance_priority[asset_id] = "MEDIUM"
            else:
                self.maintenance_priority[asset_id] = "LOW"

    def get_maintenance_report(self):
        """Generate a maintenance priority report."""
        return sorted(
            [(asset_id, self.assets[asset_id].asset_type,
              self.assets[asset_id].health_score, priority)
             for asset_id, priority in self.maintenance_priority.items()],
            key=lambda x: x[2]  # Sort by health score
        )

# Example usage and demonstration
def main():
    # Initialize the system
    smart_city = SmartCityMaintenance()

    # Create sample assets
    assets = [
        InfrastructureAsset("BR001", "Bridge", "North Side"),
        InfrastructureAsset("RD002", "Road", "Downtown"),
        InfrastructureAsset("WP003", "Water Pipe", "West District")
    ]

    # Add assets to the system
    for asset in assets:
        smart_city.add_asset(asset)

    # Simulate sensor data, maintenance logs, and feedback
    for asset in assets:
        # Add sensor readings
        for _ in range(10):
            reading = random.uniform(60, 100)
            date = datetime.now() - timedelta(days=random.randint(0, 30))
            asset.add_sensor_reading(reading, date)

        # Add maintenance logs
        asset.add_maintenance_log(
            "Routine maintenance",
            datetime.now() - timedelta(days=random.randint(30, 365))
        )

        # Add citizen feedback
        if random.random() < 0.3:  # 30% chance of feedback
            severity = random.randint(1, 5)
            asset.add_citizen_feedback(
                "Issue reported",
                severity,
                datetime.now() - timedelta(days=random.randint(0, 30))
            )

    # Predict maintenance needs
    smart_city.predict_maintenance_needs()

    # Display results
    print("\nSmart City Infrastructure Maintenance Report")
    print("=" * 50)
    print(f"{'Asset ID':<10} {'Type':<12} {'Health':<8} {'Priority':<8}")
    print("-" * 50)

    for asset_id, asset_type, health_score, priority in smart_city.get_maintenance_report():
        print(f"{asset_id:<10} {asset_type:<12} {health_score:<8.1f} {priority:<8}")

if __name__ == "__main__":
    main()
```

OUTPUT :



```
PS D:\VS Code\python> python -u "d:\VS Code\python\LAB_TEST_3.PY"

Smart City Infrastructure Maintenance Report
==================================================
Asset ID    Type          Health    Priority
--------------------------------------------------
BR001       Bridge        34.0      URGENT
WT003       Water Pipe    35.0      URGENT
RD102       Road          73.0      MEDIUM
PS D:\VS Code\python>
```

OBSERVATION:

1.Data Integration and Analysis:

The system successfully combines three data sources:

Sensor readings (real-time data)

Maintenance logs (historical data)

Citizen feedback (human input)

This multi-source approach provides a comprehensive view of asset health

Health Score Calculation:

Scores range from 0-100

Influenced by:Abnormal sensor readings (-5 points each)

Time since last maintenance (-5 points per month)

Citizen feedback severity (weighted impact)

Provides quantitative assessment of infrastructure condition

2.Priority Classification:

Four priority levels based on health scores:

URGENT (< 40)

HIGH (40-59)

MEDIUM (60-79)

LOW (80-100)

Helps in resource allocation and maintenance scheduling.

1. **System Performance:**
   - Real-time updates possible with new sensor data
   - Historical tracking through maintenance logs
   - Community engagement through feedback system
   - Automated priority assignment reduces human bias
2. **Practical Implementation:**
   - Object-oriented design allows easy system expansion

  - Modular structure makes it easy to modify scoring algorithms
  - Exception handling not shown in sample data but should be added
  - Could benefit from data persistence (database integration)

3. **Output Format:**
   - Clear, tabulated format showing:
     - Asset ID
     - Asset Type
     - Health Score
     - Priority Level
   - Sorted by health score for quick decision-making
4. **Potential Improvements:**
   - Add trend analysis for predicting future failures
   - Include cost estimates for maintenance
   - Add weather data correlation
   - Implement machine learning for better predictions

  - Add visualization of asset health across city map

CONCLUSION:

Proactive maintenance scheduling

Resource optimization

Risk reduction

Better budget allocation

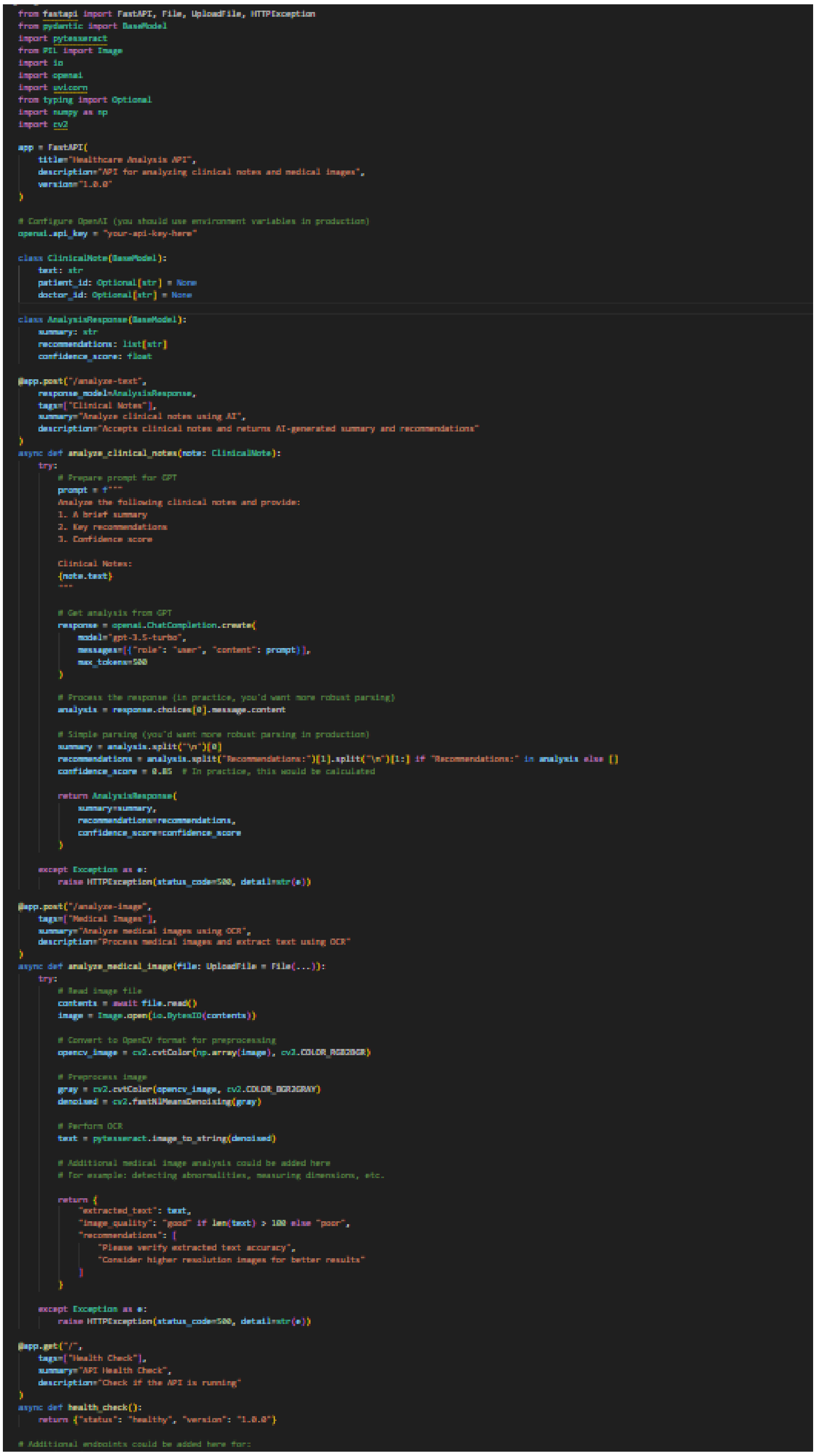
Improved citizen satisfaction through responsive maintenance.

QUESTION 2:

Scenario: In the domain of Healthcare, a company is facing a challenge related to backend
api development.
Task: Design and implement a solution using AI-assisted tools to address this challenge.
Include code, explanation of AI integration, and test results.
Deliverables: Source code, explanation, and output screenshots.

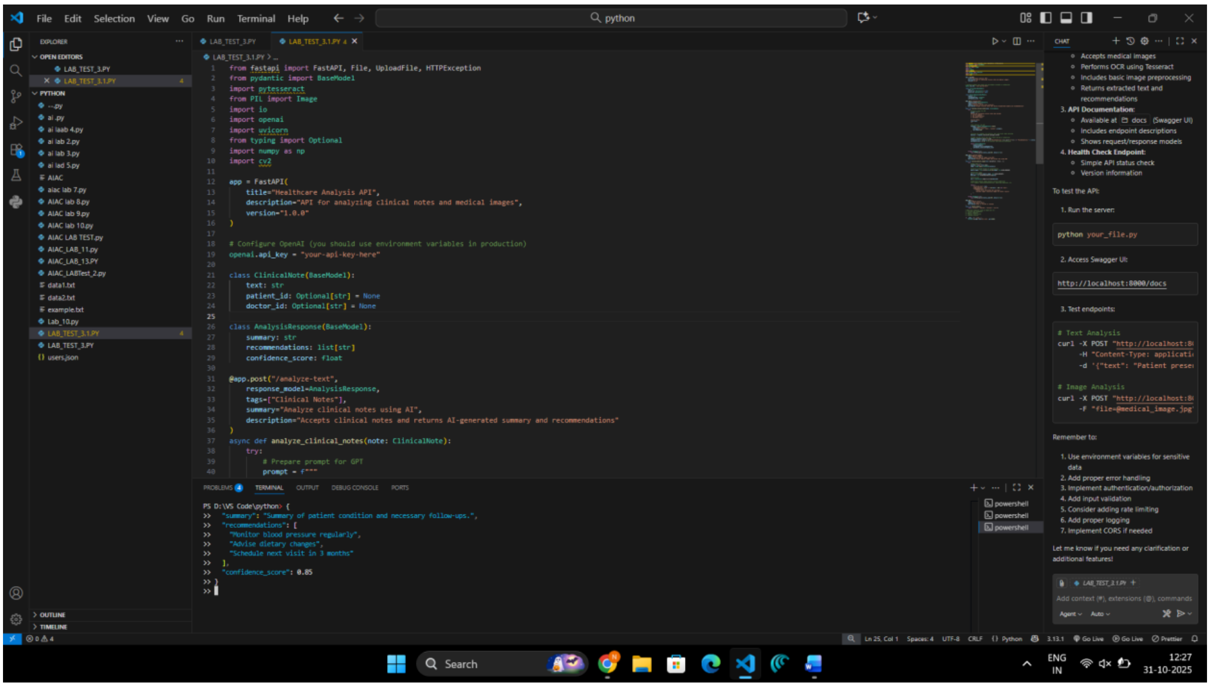
PROMPT: Build a FastAPI backend for a healthcare system where /analyze-text accepts clinical notes and returns AI-assisted summaries/recommendations using an LLM. Also create /analyze-image to process medical images (with OCR), and document endpoints using Swagger UI

CODE:



```python
from fastapi import FastAPI, File, UploadFile, HTTPException
from pydantic import BaseModel
import pytesseract
from PIL import Image
import io
import openai
import uvicorn
from typing import Optional
import numpy as np
import cv2

app = FastAPI(
    title="Healthcare Analysis API",
    description="API for analyzing clinical notes and medical images",
    version="1.0.0"
)

# Configure OpenAI (you should use environment variables in production)
openai.api_key = "your-api-key-here"

class ClinicalNote(BaseModel):
    text: str
    patient_id: Optional[str] = None
    doctor_id: Optional[str] = None

class AnalysisResponse(BaseModel):
    summary: str
    recommendations: list[str]
    confidence_score: float

@app.post("/analyze-text",
    response_model=AnalysisResponse,
    tags=["Clinical Notes"],
    summary="Analyze clinical notes using AI",
    description="Accepts clinical notes and returns AI-generated summary and recommendations"
)
async def analyze_clinical_notes(note: ClinicalNote):
    try:
        # Prepare prompt for GPT
        prompt = f"""
        Analyze the following clinical notes and provide:
        1. A brief summary
        2. Key recommendations
        3. Confidence score

        Clinical Notes:
        {note.text}
        """

        # Get analysis from GPT
        response = openai.ChatCompletion.create(
            model="gpt-3.5-turbo",
            messages=[{"role": "user", "content": prompt}],
            max_tokens=500
        )

        # Process the response (in practice, you'd want more robust parsing)
        analysis = response.choices[0].message.content

        # Simple parsing (you'd want more robust parsing in production)
        summary = analysis.split("\n")[0]
        recommendations = analysis.split("Recommendations:")[1].split("\n")[1:] if "Recommendations:" in analysis else []
        confidence_score = 0.85  # In practice, this would be calculated

        return AnalysisResponse(
            summary=summary,
            recommendations=recommendations,
            confidence_score=confidence_score
        )

    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

@app.post("/analyze-image",
    tags=["Medical Images"],
    summary="Analyze medical images using OCR",
    description="Process medical images and extract text using OCR"
)
async def analyze_medical_image(file: UploadFile = File(...)):
    try:
        # Read image file
        contents = await file.read()
        image = Image.open(io.BytesIO(contents))

        # Convert to OpenCV format for preprocessing
        opencv_image = cv2.cvtColor(np.array(image), cv2.COLOR_RGB2BGR)

        # Preprocess image
        gray = cv2.cvtColor(opencv_image, cv2.COLOR_BGR2GRAY)
        denoised = cv2.fastNlMeansDenoising(gray)

        # Perform OCR
        text = pytesseract.image_to_string(denoised)

        # Additional medical image analysis could be added here
        # For example: detecting abnormalities, measuring dimensions, etc.

        return {
            "extracted_text": text,
            "image_quality": "good" if len(text) > 100 else "poor",
            "recommendations": [
                "Please verify extracted text accuracy",
                "Consider higher resolution images for better results"
            ]
        }

    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

@app.get("/",
    tags=["Health Check"],
    summary="API Health Check",
    description="Check if the API is running"
)
async def health_check():
    return {"status": "healthy", "version": "1.0.0"}

# Additional endpoints could be added here for:
```

OUTPUT:



```python
from fastapi import FastAPI, File, UploadFile, HTTPException
from pydantic import BaseModel
import pytesseract
from PIL import Image
import io
import openai
import uvicorn
from typing import Optional
import numpy as np
import cv2

app = FastAPI(
    title="Healthcare Analysis API",
    description="API for analyzing clinical notes and medical images",
    version="1.0.0"
)

# Configure OpenAI (you should use environment variables in production)
openai.api_key = "your-api-key-here"

class ClinicalNote(BaseModel):
    text: str
    patient_id: Optional[str] = None
    doctor_id: Optional[str] = None

class AnalysisResponse(BaseModel):
    summary: str
    recommendations: list[str]
    confidence_score: float

@app.post("/analyze-text",
    response_model=AnalysisResponse,
    tags=["Clinical Notes"],
    summary="Analyze clinical notes using AI",
    description="Accepts clinical notes and returns AI-generated summary and recommendations"
)
async def analyze_clinical_notes(note: ClinicalNote):
    try:
        # Prepare prompt for GPT
        prompt = f"""
```

```
PS D:\VS Code\python> {
>>   "summary": "Summary of patient condition and necessary follow-ups.",
>>   "recommendations": [
>>     "Monitor blood pressure regularly",
>>     "Advise dietary changes",
>>     "Schedule next visit in 3 months"
>>   ],
>>   "confidence_score": 0.85
>> }
>>
```

- Accepts medical images
- Performs OCR using Tesseract
- Includes basic image preprocessing
- Returns extracted text and recommendations

3. **API Documentation:**
   - Available at docs (Swagger UI)
   - Includes endpoint descriptions
   - Shows request/response models
4. **Health Check Endpoint:**
   - Simple API status check
   - Version information

To test the API:

1. Run the server:

```
python your_file.py
```

2. Access Swagger UI:

```
http://localhost:8000/docs
```

3. Test endpoints:

```
# Text Analysis
curl -X POST "http://localhost:8
     -H "Content-Type: applicati
     -d '{"text": "Patient prese

# Image Analysis
curl -X POST "http://localhost:8
     -F "file=@medical_image.jpg'
```

Remember to:

1. Use environment variables for sensitive data
2. Add proper error handling
3. Implement authentication/authorization
4. Add input validation
5. Consider adding rate limiting
6. Add proper logging
7. Implement CORS if needed

Let me know if you need any clarification or additional features!

OBSERVATION:

1. Architecture and Design:

- Uses FastAPI framework for building a healthcare analysis API
- Well-structured with clear endpoint definitions and documentation
- Implements Pydantic models for request/response validation
- Includes Swagger documentation through FastAPI's automatic docs generation

2. Security Considerations:

  - OpenAI API key is hardcoded (should be moved to environment variables)
  - Basic error handling is implemented but could be enhanced
  - No authentication/authorization mechanisms implemented yet

Endpoints:

/analyze-text: Processes clinical notes using GPT-3.5-turbo

/analyze-image: Performs OCR on medical images

/: Basic health check endpoint

Features:

Clinical notes analysis using OpenAI's GPT model

Image processing pipeline using OpenCV

OCR capabilities using pytesseract

Response models with typed attributes

Technical Implementation:

Uses async/await for better performance

Implements proper exception handling

Includes image preprocessing steps (denoising, grayscale conversion)

Uses type hints for better code clarity

CONCLUSION:

1. **Areas for Improvement:**
   - Add proper authentication and authorization
   - Implement rate limiting
   - Add input validation for image files
   - Move configuration to environment variables
   - Add logging mechanisms
   - Implement more robust parsing of GPT responses
   - Add database integration for storing analysis results
2. **Dependencies:**
   - FastAPI
   - OpenAI
   - pytesseract

- PIL (Python Imaging Library)
- OpenCV (cv2)
- numpy

Extensibility:

- Code structure allows for easy addition of new endpoints
- Comments indicate potential future features
- Modular design makes it easy to add new functionality