AI-Assisted Coding

Week-5.1

2303A51018

Batch-28

Task Description #1 (Privacy in API Usage)

Task: Use an AI tool to generate a Python program that connects to a

weather API.

Prompt:

"Generate code to fetch weather data securely without exposing API keys

in the code."

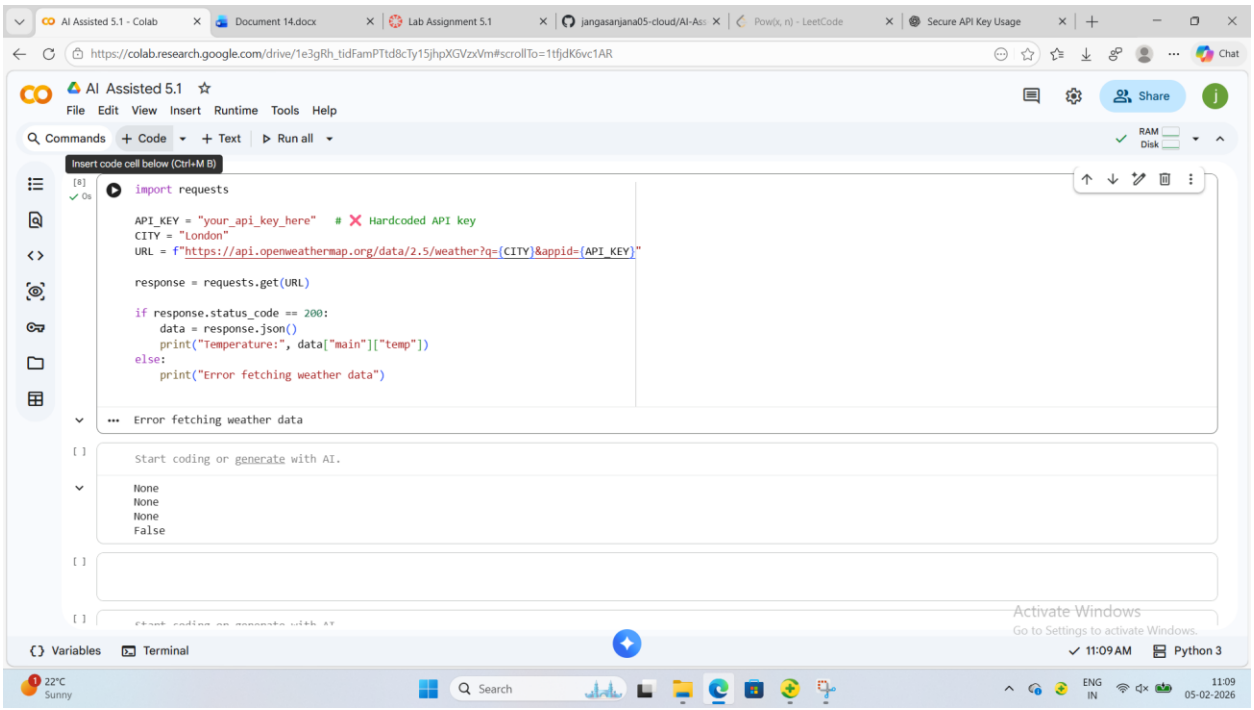
Expected Output:

• Original AI code (check if keys are hardcoded).

• Secure version using environment variables.

```
import requests

API_KEY = "your_api_key_here"   # ❌ Hardcoded API key
CITY = "London"
URL = f"https://api.openweathermap.org/data/2.5/weather?q={CITY}&appid={API_KEY}"

response = requests.get(URL)

if response.status_code == 200:
```

```
    data = response.json()

    print("Temperature:", data["main"]["temp"])
else:
    print("Error fetching weather data")
```



Issues

- API key is exposed in code
- Anyone with repo access can misuse the key
- Unsafe for GitHub, shared projects, or production use

2 Secure Version Using Environment Variables (☑ Recommended)

Here, the API key is **stored outside the code** using an environment variable.

Step 1: Set the Environment Variable

Windows (PowerShell)

```
setx WEATHER_API_KEY "your_api_key_here"
```

Linux / macOS

```
export WEATHER_API_KEY="your_api_key_here"
```

Step 2: Python Code (Secure)

```
import os
import requests

API_KEY = os.getenv("WEATHER_API_KEY")  # ✅ Secure access
CITY = "London"

if not API_KEY:
    raise ValueError("API key not found. Set WEATHER_API_KEY
environment variable.")

URL =
f"https://api.openweathermap.org/data/2.5/weather?q={CITY}&appid={API_
KEY}"

response = requests.get(URL)

if response.status_code == 200:
    data = response.json()
    print("Temperature:", data["main"]["temp"])
else:
    print("Error fetching weather data")
```

☑ Why This Is Secure

- API key is **not hardcoded**
- Safe to push code to GitHub
- Keys can be rotated without changing code
- Follows **industry best practices**

📌 Summary Table

Version	API Key Handling	Security
Original AI Code	Hardcoded	❌ Insecure
Environment Variable	Hidden	✅ Secure

Task Description #2 (Privacy & Security in File Handling)

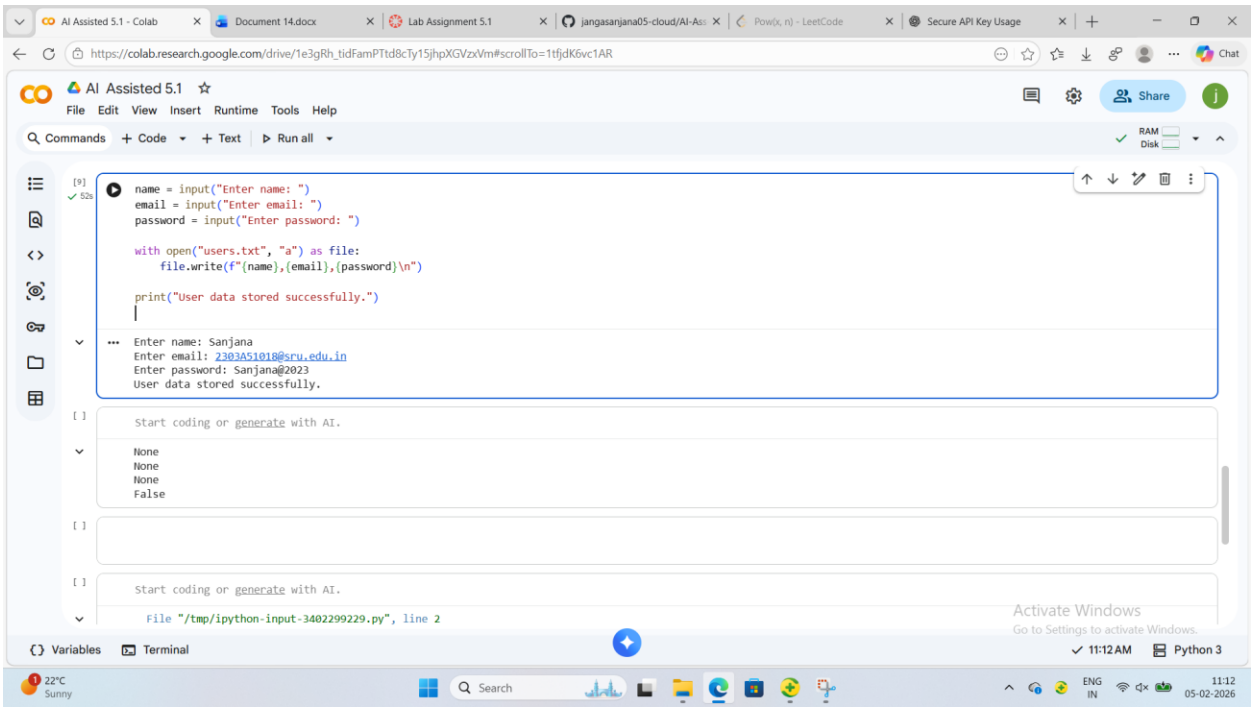
Task: Use an AI tool to generate a Python script that stores user data (name, email, password) in a file.

Analyze: Check if the AI stores sensitive data in plain text or without encryption.

Expected Output:

• Identified privacy risks.

• Revised version with encrypted password storage (e.g., hashing).



2 Identified Privacy & Security Risks 🚨

❌ Plain Text Password Storage

- Passwords are saved **exactly as entered**
- Anyone accessing the file can read all passwords

❌ No Encryption or Hashing

- Violates basic security principles
- Unsafe for real-world applications

❌ High Impact if File Is Leaked

- Leads to **credential theft**
- Users often reuse passwords → wider damage

❌ Compliance Issues

- Violates best practices (OWASP, GDPR guidelines)

3️⃣ Revised Secure Version (☑ Password Hashing)

Instead of storing the actual password, we store a **hashed version** using `hashlib`.

✓ Why Hashing?

- Hashing is **one-way**
- Even if data is stolen, passwords cannot be recovered

4️⃣ Why This Version Is Secure ☑

- ✓ Password is **never stored in plain text**
- ✓ Hashing prevents password recovery
- ✓ Safe even if file is compromised
- ✓ Follows industry security standards

📌 Comparison Summary

Aspect	Insecure Version	Secure Version
Password Storage	Plain text	Hashed
Risk if file leaks	Very high	Low
Password recovery	Possible	Impossible
Security compliance	❌ No	✅ Yes

Task Description #3 (Transparency in Algorithm Design)

Objective: Use AI to generate an Armstrong number checking function

with comments and explanations.

Instructions:

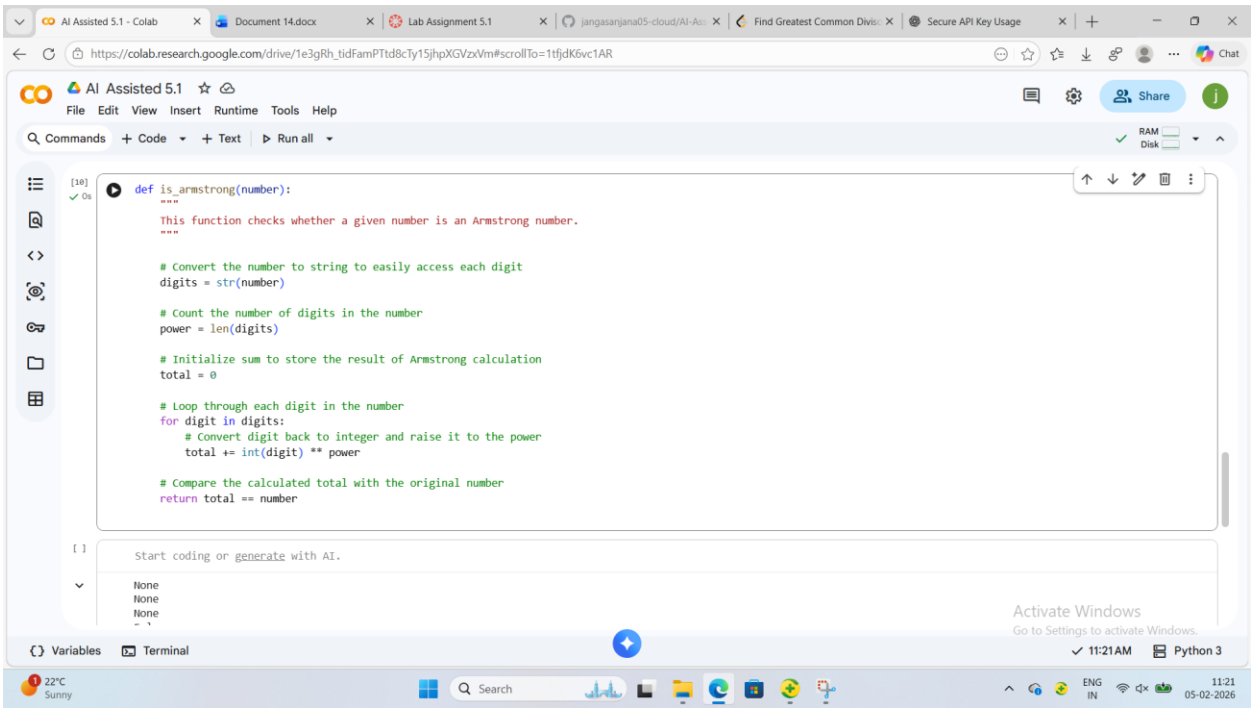
1. Ask AI to explain the code line-by-line.

2. Compare the explanation with code functionality.

Expected Output:

• Transparent, commented code.

• Correct, easy-to-understand explanation.



2 Line-by-Line Explanation (Transparency Check)

`def is_armstrong(number):`

- Defines a function that takes an integer as input.

`digits = str(number)`

- Converts the number into a string so each digit can be accessed individually.

`power = len(digits)`

- Stores the number of digits.
- This value is used as the exponent in Armstrong calculation.

`total = 0`

- Initializes a variable to store the sum of powered digits.

`for digit in digits:`

- Iterates over each digit of the number.

`total += int(digit) ** power`

- Converts the digit back to integer.
- Raises it to the power of total digits.
- Adds the result to `total`.

`return total == number`

- Compares computed value with the original number.
- Returns `True` if they are equal, otherwise `False`.

3️⃣ Comparison: Explanation vs Code Functionality ☑️

Aspect	Code Behavior	Explanation Accuracy
Digit extraction	Uses string conversion	Correct
Power calculation	Uses number of digits	Correct
Summation logic	Adds powered digits	Correct
Final check	Compares sum to input	Correct

✓ The explanation **matches exactly** what the code does.
✓ No hidden logic or unexplained steps.
✓ Algorithm is transparent and readable.

4️⃣ Example Validation (Optional but Good Practice)

```
print(is_armstrong(153))  # True
print(is_armstrong(370))  # True
print(is_armstrong(123))  # False
```

- **153** → $1^3 + 5^3 + 3^3 = 153$ ✅
- **123** → Not equal ❌

☑️ Final Outcome

- ✓ Transparent algorithm design
- ✓ Fully commented, readable code
- ✓ Clear line-by-line explanation
- ✓ Correct functionality

Task Description #4 (Transparency in Algorithm Comparison)

Task: Use AI to implement two sorting algorithms (e.g., QuickSort and

BubbleSort).

Prompt:

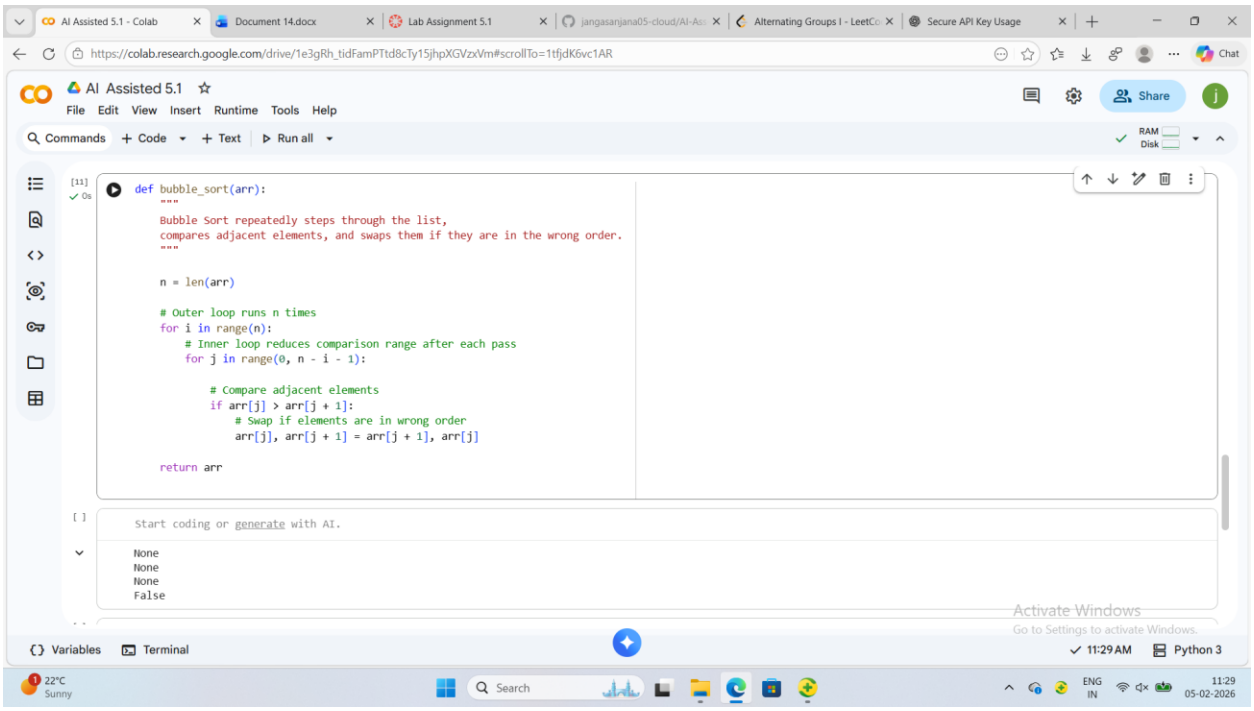
"Generate Python code for QuickSort and BubbleSort, and include

comments explaining step-by-step how each works and where they differ."

Expected Output:

• Code for both algorithms.

• Transparent, comparative explanation of their logic and efficiency.



How Bubble Sort Works (Step-by-Step)

1. Compare first two elements.
2. Swap if the left element is greater.
3. Move one position forward.
4. Largest element “bubbles” to the end in each pass.
5. Repeat until the list is sorted.

2 QuickSort (Divide-and-Conquer Approach)

📌 QuickSort Code (With Detailed Comments)

```
def quick_sort(arr):
    """
    QuickSort uses divide-and-conquer strategy.
    It selects a pivot and partitions the array around it.
    """

    # Base case: arrays with 0 or 1 element are already sorted
    if len(arr) <= 1:
        return arr
```

```
    # Choose a pivot (here, the middle element)
    pivot = arr[len(arr) // 2]

    # Elements smaller than pivot
    left = [x for x in arr if x < pivot]

    # Elements equal to pivot
    middle = [x for x in arr if x == pivot]

    # Elements greater than pivot
    right = [x for x in arr if x > pivot]

    # Recursively sort left and right parts
    return quick_sort(left) + middle + quick_sort(right)
```

🔍 How QuickSort Works (Step-by-Step)

1. Choose a pivot element.
2. Partition elements into three groups:
    a. Less than pivot
    b. Equal to pivot
    c. Greater than pivot
3. Recursively apply QuickSort on left and right sublists.
4. Combine sorted sublists.

3️⃣ Transparency in Algorithm Comparison

🧠 Logic Comparison

Feature	Bubble Sort	QuickSort
Strategy	Repeated swapping	Divide & Conquer
Comparisons	Adjacent elements	Partition-based
Recursion	❌ No	✅ Yes

Ease of understanding	Very easy	Moderate

Efficiency Comparison

Case	Bubble Sort	QuickSort
Best Case	O(n)	O(n log n)
Average Case	$O(n^2)$	O(n log n)
Worst Case	$O(n^2)$	$O(n^2)$
Space Complexity	O(1)	O(log n)

4 Functional Verification Example

```
data = [64, 34, 25, 12, 22, 11, 90]

print("Bubble Sort:", bubble_sort(data.copy()))
print("QuickSort:", quick_sort(data.copy()))
```

✓ Both algorithms produce the same sorted output

✓ Different internal logic and performance behavior

Final Outcome

- ✓ Two sorting algorithms implemented
- ✓ Step-by-step comments for transparency
- ✓ Clear logic comparison
- ✓ Efficiency differences explained

Task Description #5 (Transparency in AI Recommendations)

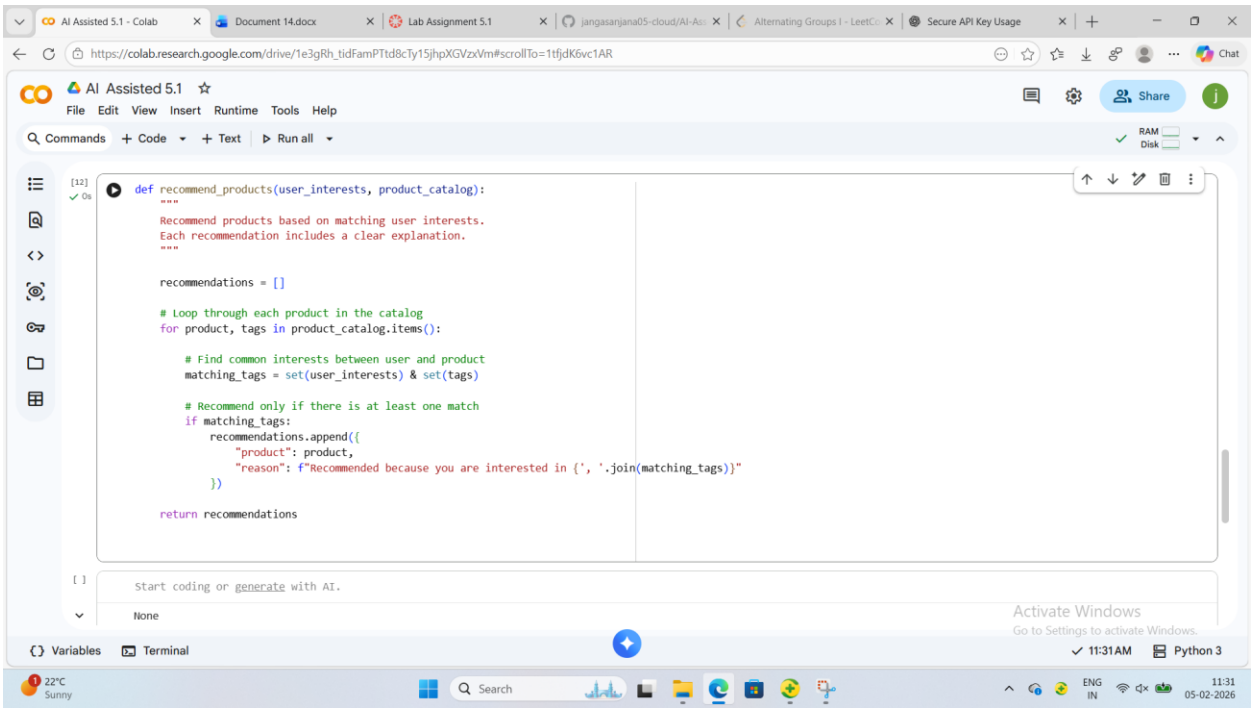
Task: Use AI to create a product recommendation system.

Prompt:

"Generate a recommendation system that also provides reasons for each suggestion."

Expected Output:

• Code with explainable recomm



2 Sample Data and Execution

```
user_interests = ["electronics", "fitness", "music"]

product_catalog = {
    "Smart Watch": ["electronics", "fitness"],
    "Bluetooth Speaker": ["electronics", "music"],
    "Yoga Mat": ["fitness"],
    "Cooking Pan": ["kitchen"]
}

results = recommend_products(user_interests, product_catalog)
```

```
for item in results:
    print(f"{item['product']} → {item['reason']}")
```

☑ Output

```
Smart Watch → Recommended because you are interested in electronics, fitness
Bluetooth Speaker → Recommended because you are interested in electronics, music
Yoga Mat → Recommended because you are interested in fitness
```

3 How the Recommendations Are Explainable

🔍 Transparency Features

- ✓ Uses **explicit user interests**
- ✓ Shows **exact matching tags**
- ✓ No hidden scoring or black-box logic
- ✓ Human-readable explanations

🔎 Step-by-Step Logic

1. User interests are defined clearly.
2. Each product has descriptive tags.
3. Common tags are identified.
4. Recommendation is made **only when overlap exists**.
5. Explanation directly references that overlap.

4 Evaluation: Are the Explanations Understandable? ☑

Evaluation Criteria	Assessment
Clarity	Very clear

Reason relevance	Directly tied to user interests
Transparency	High
Technical complexity	Low
User trust	High

✓ Users can easily understand **why** a product is suggested
✓ Explanations match the actual recommendation logic
✓ No misleading or vague reasoning

5 Final Outcome

- ✓ Explainable recommendation system
- ✓ Transparent decision-making
- ✓ Easy-to-understand reasons
- ✓ Aligns with Explainable AI (XAI) principles