

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")

```

The screenshot shows a browser window with multiple tabs open. The active tab is titled 'AI Assisted 5.1 - Colab' and contains Python code. The code imports requests, defines an API key as 'your\_api\_key\_here', sets the city to 'London', and constructs a URL to fetch weather data from openweathermap.org. It then attempts to get the response and print the temperature if successful, or print an error message if it fails.

```

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).

The screenshot shows a Google Colab notebook titled "AI Assisted 5.1". The code cell contains the following Python script:

```

name = input("Enter name: ")
email = input("Enter email: ")
password = input("Enter password: ")

with open("users.txt", "a") as file:
    file.write(f"{name},{email},{password}\n")

print("User data stored successfully.")

```

When run, the output is:

```

... Enter name: Sanjana
Enter email: 2303A51018@sru.edu.in
Enter password: Sanjana@2023
User data stored successfully.

```

The sidebar on the right shows AI-related options like "start coding or generate with AI". The status bar at the bottom indicates it's a Python 3 environment.

## 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)

## 3 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

### 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.

The screenshot shows a Google Colab notebook titled "AI Assisted 5.1 - Colab". The code cell contains the following Python function:

```
def is_armstrong(number):
    """
    This function checks whether a given number is an Armstrong number.
    """

    # Convert the number to string to easily access each digit
    digits = str(number)

    # Count the number of digits in the number
    power = len(digits)

    # Initialize sum to store the result of Armstrong calculation
    total = 0

    # Loop through each digit in the number
    for digit in digits:
        # Convert digit back to integer and raise it to the power
        total += int(digit) ** power

    # Compare the calculated total with the original number
    return total == number
```

The code cell has a status bar indicating "10s" and "0s". Below the code cell, there is an AI interface with the text "Start coding or generate with AI." and a dropdown menu showing "None", "None", and "None". The status bar at the bottom right shows "Activate Windows", "Go to Settings to activate Windows.", "11:21AM", "Python 3", "22°C Sunny", "ENG IN", "11:21", and "05-02-2026".

## 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.

## 3 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.

## Example Validation (Optional but Good Practice)

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

- $153 \rightarrow 1^3 + 5^3 + 3^3 = 153$  ✓
- $123 \rightarrow$  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.

```

def bubble_sort(arr):
    """
    Bubble Sort repeatedly steps through the list,
    compares adjacent elements, and swaps them if they are in the wrong order.
    """

    n = len(arr)

    # Outer loop runs n times
    for i in range(n):
        # Inner loop reduces comparison range after each pass
        for j in range(0, n - i - 1):

            # Compare adjacent elements
            if arr[j] > arr[j + 1]:
                # Swap if elements are in wrong order
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

    return arr

```

Start coding or generate with AI.

None  
None  
None  
False

Activate Windows  
Go to Settings to activate Windows.

Variables Terminal

22°C Sunny 11:29 AM ENG IN 11:29 05-02-2026 Python 3

## 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.

## 3 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	$O(1)$	$O(\log n)$

## 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)

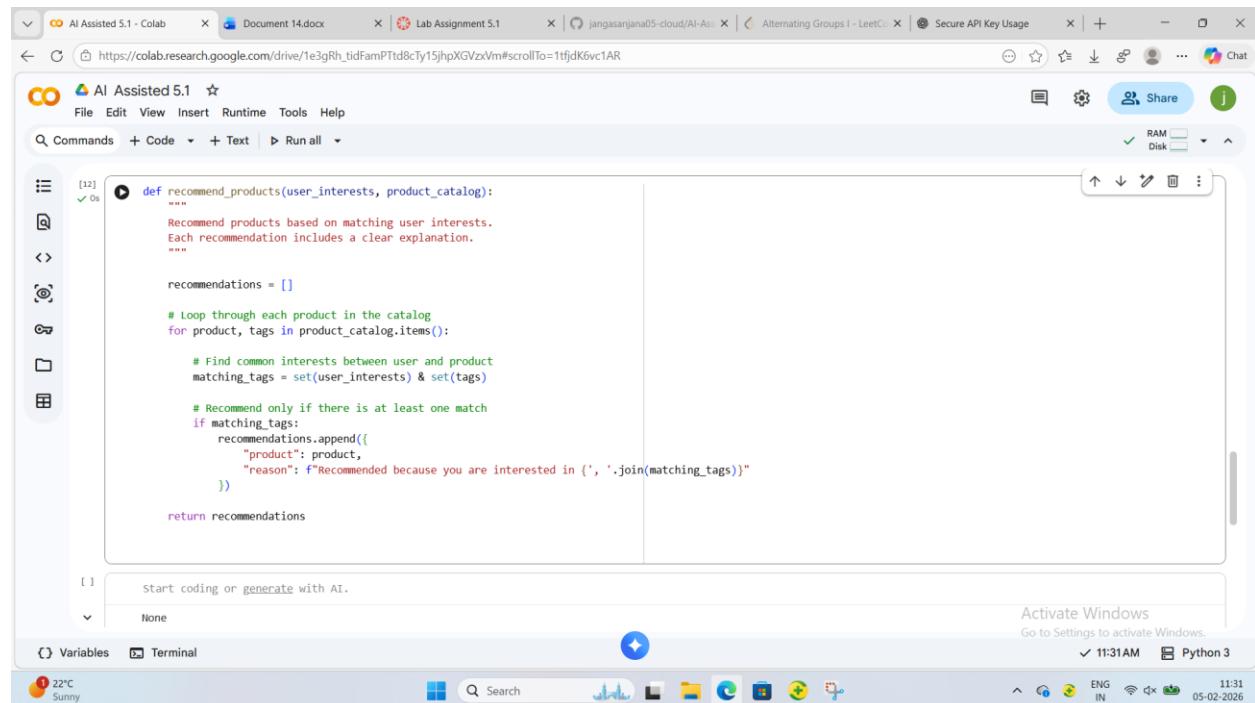
Task: Use AI to create a product recommendation system.

Prompt:

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

Expected Output:

- Code with explainable recomm



The screenshot shows a Google Colab notebook titled "AI Assisted 5.1". The code cell contains the following Python function:

```
def recommend_products(user_interests, product_catalog):
    """
    Recommend products based on matching user interests.
    Each recommendation includes a clear explanation.
    """

    recommendations = []
    # Loop through each product in the catalog
    for product, tags in product_catalog.items():
        # Find common interests between user and product
        matching_tags = set(user_interests) & set(tags)

        # Recommend only if there is at least one match
        if matching_tags:
            recommendations.append({
                "product": product,
                "reason": f"Recommended because you are interested in {', '.join(matching_tags)}"
            })
    return recommendations
```

The code cell has a status bar indicating "12" cells, "0s" execution time, and "RAM" usage. Below the code cell is a text input field with placeholder text "Start coding or generate with AI." and a "None" dropdown. At the bottom of the screen, there is a taskbar with icons for search, file explorer, and other applications, along with system status indicators like battery level, signal strength, and date/time.

## 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.

## 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