

## Assignment – 2.5

Name: P.Aneesh Reddy

Roll Number: 2303A51120

Batch - 03

AI Assisted Coding

16-01-2026

Task 1: Refactoring Odd/Even Logic (List Version)

❖ Scenario:

You are improving legacy code.

❖ Task:

Write a program to calculate the sum of odd and even numbers in a list, then refactor it using AI.

❖ Expected Output:

❖ Original and improved code

The screenshot shows the VS Code interface with the following details:

- File Explorer:** Shows the project structure with files like task1.py, task2.py, task3.py, task4.py, task5\_iterative.py, and task5\_recursive.py.
- Code Editor:** The main editor window displays the Python code for Task 1. The code is split into two sections: "Original Code (Legacy Style)" and "Improved Code (Refactored)".
  - Original Code (Legacy Style):** A function `calculate_sums_original` that iterates through a list of numbers, summing odd and even numbers separately.
  - Improved Code (Refactored):** A function `calculate_sums_improved` that uses a more concise approach with list comprehension and the `sum` function.
- Terminal:** Shows the command line output of running the script with sample input [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]. The output shows the sum of odd numbers as 25 and even numbers as 30.
- Status Bar:** Shows the current file is task1.py, the cursor position is at 21:0t, and the Python version is 3.14 (64-bit).

```

task1.py - AI-A-coding-v2 - Cursor
File Edit Selection View Go Run Terminal Help
Assessment2.5 > task1-2.py > ...
task1.py U
task1-2.py U
Assessment2.5 > task1-2.py > ...
1 # Improved Code (Refactored)
2 def calculate_sum_improved(numbers):
3     """
4         Calculate the sum of odd and even numbers in a list.
5     """
6     Args:
7         numbers: List of integers
8
9     Returns:
10        tuple: (sum_of_odd_numbers, sum_of_even_numbers)
11
12    odd_sum = sum(num for num in numbers if num % 2 != 0)
13    even_sum = sum(num for num in numbers if num % 2 == 0)
14
15    return odd_sum, even_sum
16
17 # Alternative Improved version using filter
18 def calculate_sum_alternative(numbers):
19     """Alternative refactored version using filter."""
20     odd_sum = sum(filter(lambda x: x % 2 != 0, numbers))
21     even_sum = sum(filter(lambda x: x % 2 == 0, numbers))
22
23     return odd_sum, even_sum
24
25 # Test the Improved code
26 numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
27 odd, even = calculate_sum_improved(numbers)
28 print(f"\nImproved Code:")
29 print(f"Sum of odd numbers: {odd}")
30 print(f"Sum of even numbers: {even}")
31
32 # Test alternative version
33 odd2, even2 = calculate_sum_alternative(numbers)
34 print(f"\nAlternative Improved Code:")
35 print(f"Sum of odd numbers: {odd2}")
36 print(f"Sum of even numbers: {even2}")

task1-2.py(1-9)
# Improved Code (Refactored)
def calculate_sum_improved(numbers):
    """
    Calculate the sum of odd and even numbers in a list.
    """
    Args:
        numbers: List of integers
    Returns:
        tuple: (sum_of_odd_numbers, sum_of_even_numbers)
    odd_sum = sum(num for num in numbers if num % 2 != 0)
    even_sum = sum(num for num in numbers if num % 2 == 0)
    return odd_sum, even_sum
# Alternative Improved version using filter
def calculate_sum_alternative(numbers):
    """Alternative refactored version using filter."""
    odd_sum = sum(filter(lambda x: x % 2 != 0, numbers))
    even_sum = sum(filter(lambda x: x % 2 == 0, numbers))
    return odd_sum, even_sum
# Test the Improved code
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
odd, even = calculate_sum_improved(numbers)
print("Sum of odd numbers: (odd)")
print("Sum of even numbers: (even)")
# Test alternative version
odd2, even2 = calculate_sum_alternative(numbers)
print("Sum of odd numbers: (odd2)")
print("Sum of even numbers: (even2)")

Improvements:
1. Readability: Uses list comprehensions and built-in sum().
2. Conciseness: Fewer lines, clearer intent.
3. Pythonic: Uses generator expressions and functional patterns.
4. Documentation: Adds a docstring.
5. Efficiency: Similar performance, more idiomatic.

Both versions produce the same output:
• Sum of odd numbers: 25 (1+3+5+7+9)
• Sum of even numbers: 30 (2+4+6+8+10)
Should I combine these into a single file, or do you want to test them separately?

```

## Task 2: Area Calculation Explanation ❁

Scenario:

You are onboarding a junior developer.

❖ Task:

Ask Gemini to explain a function that calculates the area of different shapes.

❖ Expected Output:

➤ Code

➤ Explanation

The screenshot shows the Gemini AI Assisted Coding interface. On the left, there's a code editor with Python code for reversing a string:

```

github > assg 2.5 > result
1 def reverse_string_function(text):
2     reversed_text = ""
3     for char in text:
4         reversed_text = char + reversed_text
5     return reversed_text
6
7
8 input_string = input("Enter a string: ")
9 result= reverse_string_function(input_string)
10 print("result: " + result)
11

```

On the right, another code editor shows a script for calculating areas:

```

Assessment2.5 > task2.py > ...
1 import math
2
3 def calculate_area(shape, **kwargs):
4     Calculates the area of different shapes.
5
6     Args:
7         shape (str): The name of the shape (e.g., 'rectangle', 'circle', 'triangle').
8         *kwargs: The dimensions of the shape.
9         For 'rectangle', expects 'length' and 'width'.
10        For 'circle', expects 'radius'.
11        For 'triangle', expects 'base' and 'height'.
12
13    Returns:
14        float: The calculated area of the shape.
15        str: An error message if the shape is not supported or dimensions are missing.
16
17    If shape == "rectangle":
18        if "length" in kwargs and "width" in kwargs:
19            return kwargs["length"] * kwargs["width"]
20        else:
21            return "Error: Missing length or width for rectangle."
22    elif shape == "circle":
23        if "radius" in kwargs:
24            return math.pi * (kwargs["radius"] ** 2)
25        else:
26            return "Error: Missing radius for circle."
27    elif shape == "triangle":
28        if "base" in kwargs and "height" in kwargs:
29            return 0.5 * kwargs["base"] * kwargs["height"]
30        else:
31            return "Error: Missing base or height for triangle."
32    else:
33        return "Error: Unsupported shape."
34
35 # Example usage:
36 rectangle_area = calculate_area("rectangle", length=10, width=5)
37 print("The area of the rectangle is: " + str(rectangle_area))
38
39 circle_area = calculate_area("circle", radius=7)
40 print("The area of the circle is: " + str(circle_area))
41
42 triangle_area = calculate_area("triangle", base=8, height=4)
43 print("The area of the triangle is: " + str(triangle_area))
44
45 # Example of an error
46 square_area = calculate_area("square", side=5)
47 print("The area of the square is: " + str(square_area))
48
49

```

The terminal at the bottom shows the output of running the script:

```

PS C:\Users\pogal\OneDrive\Desktop\AI Assisted coding> python task2.py
The area of the rectangle is: 50
The area of the circle is: 153.93804002589885
The area of the triangle is: 16.0
The area of the square is: Error: Unsupported shape.

```

## Explanation

This Python script defines a single function `calculate_area` that is designed to be a flexible tool for finding the area of several different geometric shapes.

### Function Definition:

```
def calculate_area(shape, **kwargs):
```

This defines a function named calculate\_area that accepts two arguments: shape: A string that tells the function which shape's area to calculate (e.g., "rectangle").

\*\*kwargs: This is a special Python feature that allows the function to accept any number of keyword arguments (like length=10 or radius=5). These arguments are collected into a dictionary named kwargs. This makes the function adaptable, as different shapes require different parameters for area calculation.

Conditional Logic:

The function uses if, elif (else if), and else statements to determine which shape has been requested and then performs the correct calculation. if shape == "rectangle":

If the shape is a "rectangle", the code checks if the necessary dimensions, length and width, are provided in the kwargs dictionary.

If they are present, it calculates the area by multiplying length and width.

If they are missing, it returns an error message.

elif shape == "circle":

If the shape is a "circle", it checks for a radius.

If found, it calculates the area using the formula  $\pi * r^2$ , importing the math module to get an accurate value of math.pi. If the radius is missing, it returns an error. elif shape == "triangle":

If the shape is a "triangle", it checks for a base and height.

If they are present, it calculates the area using the formula  $0.5 * \text{base} * \text{height}$ .

If they are missing, it returns an error.

else:

If the provided shape name does not match any of the supported shapes ("rectangle", "circle", "triangle"), the function returns an "Unsupported shape" error.

Example Usage:

The lines at the end of the script demonstrate how to use the function and print the results for a rectangle, a circle, a triangle, and an unsupported shape to show how the error handling works.

### Task 3: Prompt Sensitivity Experiment

❖ Scenario:

You are testing how AI responds to different prompts.

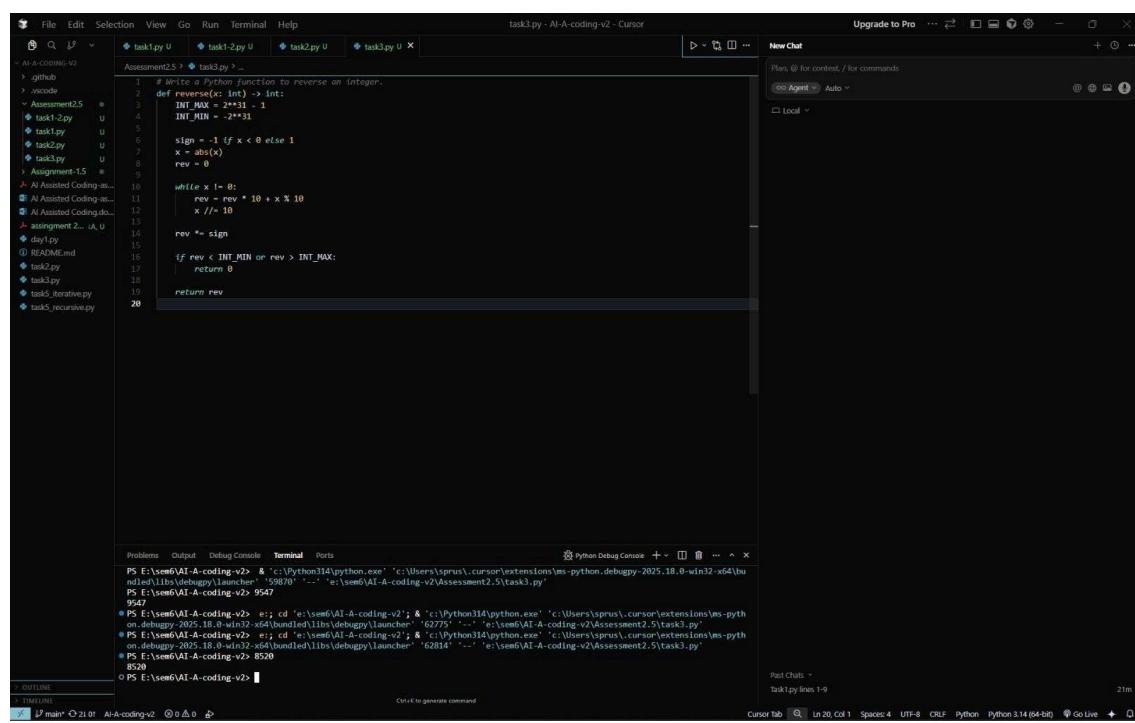
❖ Task:

Use Cursor AI with different prompts for the same problem and observe code changes.

❖ Expected Output:

➤ Prompt list

➤ Code variations



```
# Write a Python function to reverse an integer.
def reverse(x: int) -> int:
    INT_MAX = 2**31 - 1
    INT_MIN = -2**31

    sign = -1 if x < 0 else 1
    x = abs(x)
    rev = 0

    while x != 0:
        rev = rev * 10 + x % 10
        x //= 10

    rev *= sign

    if rev < INT_MIN or rev > INT_MAX:
        return 0

    return rev
```

The screenshot shows a code editor interface with two tabs open: task3.py U and task3.py U. The code in both tabs is identical, demonstrating how AI generates the same solution with slight variations in structure (while vs for loops, abs(x) vs x). The terminal below shows the command to run the file and its execution.

```
PS E:\sem6\AI-A-coding-v2> & cd 'E:\sem6\AI-A-coding-v2' & 'c:\Python314\python.exe' 'c:\Users\spurus\cursor\extensions\ms-python.debugger-2025.18.0-win32-x64\bundle\lib\debugpy\launcher' '62775' '--' 'e:\sem6\AI-A-coding-v2\Assessment2.5\task3.py'
PS E:\sem6\AI-A-coding-v2> 9547
PS E:\sem6\AI-A-coding-v2> & cd 'E:\sem6\AI-A-coding-v2' & 'c:\Python314\python.exe' 'c:\Users\spurus\cursor\extensions\ms-python.debugger-2025.18.0-win32-x64\bundle\lib\debugpy\launcher' '62775' '--' 'e:\sem6\AI-A-coding-v2\Assessment2.5\task3.py'
PS E:\sem6\AI-A-coding-v2> e1; cd 'E:\sem6\AI-A-coding-v2' & 'c:\Python314\python.exe' 'c:\Users\spurus\cursor\extensions\ms-python.debugger-2025.18.0-win32-x64\bundle\lib\debugpy\launcher' '62814' '--' 'e:\sem6\AI-A-coding-v2\Assessment2.5\task3.py'
PS E:\sem6\AI-A-coding-v2> 8520
8520
PS E:\sem6\AI-A-coding-v2>
```

task3.2.py - AI-A-coding-v2 - Cursor

```

File Edit Selection View Go Run Terminal Help
task1.py U task2.py U task2z.py U task3.py U task3z.py U
Assessment2.5 task3.2.py

def reverse_integer(x):
    """
    Reverse an integer with 32-bit signed integer overflow handling.
    """
    # Handle negative numbers
    if x < 0:
        num = abs(x)
    else:
        num = x

    # Reverse the digits
    reversed_num = 0
    while num != 0:
        reversed_num *= 10
        reversed_num += num % 10
        num /= 10

    # Apply sign
    if x < 0:
        reversed_num = -reversed_num

    # Check for 32-bit signed integer overflow
    INT_MIN = -2**31 - 1 - 2147483648
    INT_MAX = 2**31 - 1 - 2147483647
    if reversed_num < INT_MIN or reversed_num > INT_MAX:
        return 0

    return reversed_num

# Test cases
if __name__ == "__main__":
    # Test case 1: Positive number
    print(reverse_integer(123)) # Expected: 321

    # Test case 2: Negative number
    print(reverse_integer(-123)) # Expected: -321

    # Test case 3: Number ending with zero
    print(reverse_integer(120)) # Expected: 21

    # Test case 4: Single digit
    print(reverse_integer(5)) # Expected: 5

    # Test case 5: Zero
    print(reverse_integer(0)) # Expected: 0

    # Test case 6: Overflow case (reversed number exceeds INT_MAX)
    print(reverse_integer(1543264865)) # Expected: 0 (overflows)

    # Test case 7: Overflow case (reversed number is less than INT_MIN)
    print(reverse_integer(-2147483648)) # Expected: 0 (overflows)

```

Upgrade to Pro

Integer reversal with overflow handling

Write a Python function to reverse an Integer with 32-bit signed integer overflow handling.

Checking the current file for context.

Read task3.2.py

Writing a function to reverse an integer with 32-bit signed integer overflow handling.

Edit illegal value for intNumber

Reading the file again:

Read task3.2.py

Improving overflow handling: checking during reversal to prevent intermediate values from exceeding 32-bit bounds.

Edit illegal value for intNumber

Rewriting the file with improved overflow handling.

Edit illegal value for intNumber

Checking the file and trying a different approach:

Read task3.2.py

Improving overflow handling by checking during reversal instead of after:

Edit illegal value for intNumber

Testing the current implementation:

Run command: cd

cd /e:/vscode/AT-A-coding-v2/Assessment2.5 && python task3.2.py

Use AllSelected

2/2 Undo All Ctrl+Shift+G Keep All Ctrl+Shift+H

Problems Output Debug Console Terminal Ports

PowerShell Extension v2025.4.0

Copyright (c) Microsoft Corporation.

https://marketplace.visualstudio.com/items?itemName=ms-vscode-powershell

Type 'help' to get help.

PS E:\vscode\AT-A-coding-v2>

task3.2.py - AI-A-coding-v2 - Cursor

```

File Edit Selection View Go Run Terminal Help
task1.py U task2.py U task2z.py U task3.py U task3z.py U
Assessment2.5 task3.2.py

def reverse_integer(x):
    """
    Reverse an integer with 32-bit signed integer overflow handling.
    """
    # Handle negative numbers
    if x < 0:
        num = abs(x)
    else:
        num = x

    # Reverse the digits
    reversed_num = 0
    while num != 0:
        reversed_num *= 10
        reversed_num += num % 10
        num /= 10

    # Apply sign
    if x < 0:
        reversed_num = -reversed_num

    # Check for 32-bit signed integer overflow
    INT_MIN = -2**31 - 1 - 2147483648
    INT_MAX = 2**31 - 1 - 2147483647
    if reversed_num < INT_MIN or reversed_num > INT_MAX:
        return 0

    return reversed_num

# Test cases
if __name__ == "__main__":
    # Test case 1: Positive number
    print(reverse_integer(123)) # Expected: 321

    # Test case 2: Negative number
    print(reverse_integer(-123)) # Expected: -321

    # Test case 3: Number ending with zero
    print(reverse_integer(120)) # Expected: 21

    # Test case 4: Single digit
    print(reverse_integer(5)) # Expected: 5

    # Test case 5: Zero
    print(reverse_integer(0)) # Expected: 0

    # Test case 6: Overflow case (reversed number exceeds INT_MAX)
    print(reverse_integer(1543264865)) # Expected: 0 (overflows)

    # Test case 7: Overflow case (reversed number is less than INT_MIN)
    print(reverse_integer(-2147483648)) # Expected: 0 (overflows)

```

Upgrade to Pro

Integer reversal with overflow handling

Write a Python function to reverse an Integer with 32-bit signed integer overflow handling.

Checking the current file for context.

Read task3.2.py

Improving overflow handling by checking during reversal instead of after:

Edit illegal value for intNumber

Testing the current implementation:

Run command: cd

cd /e:/vscode/AT-A-coding-v2/Assessment2.5 && python task3.2.py

At C:\Users\spv\Downloads\attempscript-rcd810-71b-4912-aed7-7d7a2e81184.ps

107 characters

+ cd /e:/vscode/AT-A-coding-v2/Assessment2.5 && python task3.2.py

+ if \_\_name\_\_ == "\_\_main\_\_":
+ print(reverse\_integer(123))
+ print(reverse\_integer(-123))
+ print(reverse\_integer(120))
+ print(reverse\_integer(5))
+ print(reverse\_integer(0))
+ print(reverse\_integer(1543264865))
+ print(reverse\_integer(-2147483648))
+
+ The value 'INT' is not a valid statement separator in this version.
+ CategoryError: Line 107:1: ParserException [None]: Keyword contains error: invalidNameLine
+ FullyQualifiedEnvVarId: invalidNameLine

Firing the PowerShell command:

Run command: cd

cd /e:/vscode/AT-A-coding-v2/Assessment2.5; python task3.2.py

reverse\_integer(123) = 321
reverse\_integer(-123) = -321
reverse\_integer(120) = 21
reverse\_integer(5) = 5
reverse\_integer(0) = 0
reverse\_integer(1543264865) = 0
reverse\_integer(-2147483648) = 0

Use AllSelected

Success

The function is implemented and working. It:
1. Reverses an integer by extracting digits and building the reversed number
2. Handles negative numbers by preserving the sign
3. Checks for 32-bit signed integer overflow -- returns 0 if the reversed number is outside [-2147483648, 2147483647]

Test results:
• 321 → 321
• -321 → -321
• 120 → 21 (trailing zeros removed)
• 5 → 5
• 0 → 0
• 1543264865 → 0 (overflow)
• -2147483648 → 0 (overflow)

The code successfully handles edge cases and overflows. The implementation checks bounds after reversal, which works since Python integers are unbounded and we only need to validate the final result against 32-bit limits.

Would you like me to optimize it to check for overflow during reversal, or is the current implementation sufficient?

PS E:\vscode\AT-A-coding-v2>

Problems Output Debug Console Terminal Ports

PowerShell Extension v2025.4.0

Copyright (c) Microsoft Corporation.

https://marketplace.visualstudio.com/items?itemName=ms-vscode-powershell

Type 'help' to get help.

PS E:\vscode\AT-A-coding-v2>

The screenshot shows a code editor with two tabs open:

- task3.3.py - AI-A-coding-v2 - Cursor**: The main code file.
- task3.3.py - AI-A-coding-v2**: A copy of the code with some annotations and a different cursor position.

The code implements a function `reverse_integer` that reverses an integer while performing strict overflow checks. It handles both positive and negative integers, ensuring results fit within the 32-bit signed integer range. The code uses division and modulo operations to extract digits and handle overflow conditions.

```
# Write an optimized Python solution for reversing an integer with strict overflow checks for coding interviews.

def reverse_integer():
    """Optimized solution to reverse an integer with strict 32-bit signed integer overflow handling.

    Time Complexity: O(log10(n)) - where n is the input number
    Space Complexity: O(1) - constant extra space

    Args:
        x: An integer to reverse

    Returns:
        The reversed integer, or 0 if the reversed integer overflows 32-bit signed integer range
        (i.e., outside [-2^31, 2^31 - 1] = [-2147483648, 2147483647])
    """
    # Strict overflow check BEFORE multiplication
    # Check if result * 10 == digit would overflow
    # For positive overflow: result * 10 > INT_MAX
    # Rearranged: result > INT_MAX // 10 or (result == INT_MAX // 10 and digit > INT_MAX % 10);
    # If result < INT_MIN // 10 or (result == INT_MIN // 10 and digit < INT_MIN % 10);
    # Safe to perform the operation
    result = result * 10 + digit

    return result

# Alternative optimized version (more Pythonic and cleaner)
def reverse_integer_v2(x):
    """Alternative optimized solution - cleaner approach handling sign separately.

    Time Complexity: O(log10(n))
    """
    # Write an optimized Python solution for reversing an integer with strict overflow checks for coding interviews.

    def reverse_integer():
        """Optimized solution to reverse an integer with strict 32-bit signed integer overflow handling.

        Time Complexity: O(log10(n)) - where n is the input number
        Space Complexity: O(1) - constant extra space

        Args:
            x: An integer to reverse

        Returns:
            The reversed integer, or 0 if the reversed integer overflows 32-bit signed integer range
            (i.e., outside [-2^31, 2^31 - 1] = [-2147483648, 2147483647])
        """
        # Strict overflow check BEFORE multiplication
        # Check if result * 10 == digit would overflow
        # For positive overflow: result * 10 > INT_MAX
        # Rearranged: result > INT_MAX // 10 or (result == INT_MAX // 10 and digit > INT_MAX % 10);
        # If result < INT_MIN // 10 or (result == INT_MIN // 10 and digit < INT_MIN % 10);
        # Safe to perform the operation
        result = result * 10 + digit

        return result

    # Alternative optimized version (more Pythonic and cleaner)
    def reverse_integer_v2(x):
        """Alternative optimized solution - cleaner approach handling sign separately.

        Time Complexity: O(log10(n))
        """
        pass
```

The right side of the interface includes an "Upgrade to Pro" button and a sidebar with the following sections:

- Integer reversal with overflow handling**: A brief description of the problem.
- Checking the current file**: A link to check the current file's status.
- Implementing an optimized solution with strict overflow checks during reversal**: A snippet of code for reversing an integer with strict overflow checks.
- Testing the optimized solution**: A command-line interface for running the code.
- Run command: cd python**
- cd "E:\sem6\AI-A-coding-v2\Assessment2.5"; python task3.3.py**
- User Answer:**  Abhishek19 [1/4] Run [Run]

At the bottom, there are tabs for "Output", "Debug", "Console", and "Terminal".

## Task 4: Tool Comparison Reflection

## ❖ Scenario:

**You must recommend an AI coding tool.**

❖ **Task:**

**Based on your work in this topic, compare Gemini, Copilot, and Cursor AI for usability and code quality.**

❖ **Expected Output:**

Short written reflection

Based on my experience using Gemini, GitHub Copilot, and Cursor AI during this topic, I observed clear differences in both usability and code quality.

Gemini is useful for understanding concepts and generating explanations, but it often produces generic code unless very strict constraints are provided. It is better suited for learning and problem understanding rather than competitive or production-level coding.

GitHub Copilot integrates smoothly with IDEs like VS Code and provides fast, context-aware code suggestions. However, its outputs sometimes assume the developer will handle edge cases, so overflow handling and constraints may be missed unless explicitly guided.

Cursor AI provided the best balance of usability and code quality. It allows direct interaction with the codebase, understands existing files, and responds well to detailed prompts. When constraints are clearly mentioned, Cursor AI consistently generated correct, optimized, and readable code, making it ideal for real development and debugging tasks.

Conclusion:

For learning → Gemini

For quick coding assistance → Copilot

For serious development and prompt-based experimentation → Cursor AI