SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE		DEPARTMENT OF COMPUTER SCIENCE ENGINEERING	
Program Name: B. Tech		**Assignment Type: Lab**	**Academic Year:**2025-2026
Course Coordinator Name	Dr. Rishabh Mittal		
Instructor(s) Name	Mr. S Naresh Kumar Ms. B. Swathi Dr. Sasanko Shekhar Gantayat Mr. Md Sallauddin Dr. Mathivanan Mr. Y Srikanth Ms. N Shilpa Dr. Rishabh Mittal (Coordinator) Dr. R. Prashant Kumar Mr. Ankushavali MD Mr. B Viswanath Ms. Sujitha Reddy Ms. A. Anitha Ms. M.Madhuri Ms. Katherashala Swetha Ms. Velpula sumalatha Mr. Bingi Raju		
CourseCode	23CS002PC304	**Course Title**	AI Assisted Coding
Year/Sem	III/II	**Regulation**	R23
Date and Day of Assignment	**Week1 – Wednesday**	**Time(s)**	23CSBTB01 To 23CSBTB52
Duration	2 Hours	**Applicable to Batches**	All batches
Assignment Number:1.3(Present assignment number)/**24**(Total number of assignments)			

Q.No.	**Question**	***Expected Time to complete***
1	Lab 2: Exploring Additional AI Coding Tools beyond Copilot – Gemini (Colab) and Cursor AI **Lab Objectives:**	Week1 - Monday

- To explore and evaluate the functionality of Google Gemini for AI-assisted coding within Google Colab.
- To understand and use Cursor AI for code generation, explanation, and refactoring.
- To compare outputs and usability between Gemini, GitHub Copilot, and Cursor AI.
- To perform code optimization and documentation using AI tools.

Lab Outcomes (LOs):
After completing this lab, students will be able to:

- Generate Python code using Google Gemini in Google Colab.
- Analyze the effectiveness of code explanations and suggestions by Gemini.
- Set up and use Cursor AI for AI-powered coding assistance.
- Evaluate and refactor code using Cursor AI features.
- Compare AI tool behavior and code quality across different platforms.

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

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

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**

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

Note: Report should be submitted as a word document for all tasks in a single document with prompts, comments & code explanation, and output and if required, screenshots.

LAB ASSIGNMENT - 2

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

PROMPT:

Write a program to calculate the sum of odd and even numbers in a list, then refactor (improve or optimize) the same code version.

Original version:

```python
# Original Code
def sum_odd_even_original(numbers):
    odds = []
    evens = []
    for num in numbers:
        if num % 2 == 0:
            evens.append(num)
        else:
            odds.append(num)
    return sum(odds), sum(evens)

# Input
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
odd_sum, even_sum = sum_odd_even_original(numbers)
print(f"Odd sum: {odd_sum}, Even sum: {even_sum}")
```

✓ 0.0s Python

Sample Input: numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Output:

```
Odd sum: 25, Even sum: 30
```

Refactored Code (Improved Version):

```python
# Refactored Code (Improved Version)
def sum_odd_even_refactored(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

# Input
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
odd_sum, even_sum = sum_odd_even_refactored(numbers)
print(f"Odd sum: {odd_sum}, Even sum: {even_sum}")
```

✓ 0.0s Python

Sample Input: numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Output:

```
Odd sum: 25, Even sum: 30
```

Task 2: Area Calculation Explanation

PROMPT: Instead of having five different functions for five different shapes, use a single entry point that handles the logic based on the shape type or inputs given.

CODE:

```python
import math

def calculate_area(shape: str, **kwargs) -> float:
    """
    Calculates the area of a given shape using keyword arguments.
    """
    shape = shape.lower()

    if shape == "circle":
        # Formula: πr²
        return math.pi * (kwargs.get("radius") ** 2)

    elif shape == "rectangle":
        # Formula: length * width
        return kwargs.get("length") * kwargs.get("width")

    elif shape == "triangle":
        # Formula: 0.5 * base * height
        return 0.5 * kwargs.get("base") * kwargs.get("height")

    else:
        raise ValueError(f"Shape '{shape}' is not supported.")
```

✓ 0.0s Python

Sample Input :

```
# Calculating a circle's area
print(calculate_area("circle", radius=5))

# Calculating a rectangle's area
print(calculate_area("rectangle", length=10, width=20))
```

✓ 0.0s

Output:

```
78.53981633974483
200
```

EXPLANATION:

This function serves as a unified entry point that identifies a geometric shape via a string and selects the corresponding mathematical formula to execute.

It leverages `**kwargs` (keyword arguments) to flexibly accept varying inputs—like a radius for circles or length/width for rectangles—without needing a rigid parameter list.

By centralizing the logic and including error handling for unsupported shapes, it ensures the code is scalable, readable, and easy for other developers to integrate.

Task 3: Prompt Sensitivity Experiment

Prompt Variations:

1. **Write a Python function to calculate the area of a circle, rectangle, and triangle.**

   Code:

```python
import math

def calculate_area(shape, val1, val2=None):
    shape = shape.lower()
    if shape == "circle":
        return math.pi * (val1 ** 2)
    elif shape == "rectangle":
        return val1 * val2
    elif shape == "triangle":
        return 0.5 * val1 * val2
    else:
        return "Shape not recognized"

# Usage
print(calculate_area("circle", 5))
print(calculate_area("rectangle", 10, 5))
```

✓ 0.0s Python

Sample Input: “Circle”, 5

Output:

```
78.53981633974483
```

2. **Create a single Python function `calculate_area` that takes a `shape` string and `**kwargs`. Support circle, rectangle, and triangle. Use the `math` library for pi.**

   Code:

```python
import math

def calculate_area(shape: str, **kwargs) -> float:
    shape = shape.lower()

    if shape == "circle":
        return math.pi * (kwargs.get("radius") ** 2)

    elif shape == "rectangle":
        return kwargs.get("length") * kwargs.get("width")

    elif shape == "triangle":
        return 0.5 * kwargs.get("base") * kwargs.get("height")

    raise ValueError(f"Unknown shape: {shape}")

# Usage
print(calculate_area("triangle", base=10, height=8))
```

✓ 0.0s Python

   Sample Input: “triangle”, base=10, height = 8

   Output :

```
40.0
```

3. **You are a Senior Python Engineer. Write a modular area calculation system. Use a dictionary-based dispatch pattern instead of `if/else`. Include type hints, docstrings, and error handling for missing arguments.**

Code:

```python
import math
from typing import Dict, Callable

# A registry of calculation 'strategies'
AREA_STRATEGIES: Dict[str, Callable] = {
    "circle": lambda k: math.pi * (k.get("radius") ** 2),
    "rectangle": lambda k: k.get("length") * k.get("width"),
    "triangle": lambda k: 0.5 * k.get("base") * k.get("height"),
    "square": lambda k: k.get("side") ** 2
}

def calculate_area(shape: str, **kwargs) -> float:
    """Entry point using dictionary dispatch for O(1) lookup."""
    try:
        strategy = AREA_STRATEGIES[shape.lower()]
        return strategy(kwargs)
    except KeyError:
        supported = ", ".join(AREA_STRATEGIES.keys())
        raise ValueError(f"Unsupported shape '{shape}'. Supported: {supported}")
    except TypeError as e:
        raise TypeError(f"Missing required dimensions for {shape}: {e}")

# Usage
print(calculate_area("square", side=4))
```

Sample Input : “square”, side = 4

Output :

16

Task 4 : Tool Comparison Reflection

Recommendations :

- **If you are a student or researcher:** Use Gemini. The free tier is generous, and its ability to explain "why" is unmatched for learning.
- **If you work in a large enterprise:** Stick with Copilot. The security, GitHub integration, and "set it and forget it" nature make it the safest bet.
- **If you are building a product from scratch:** Use Cursor. The ability to refactor across multiple files at once will save you hundreds of hours of manual "plumbing."