LAB - 10.3

AI-Assisted Coding

Name – Rounak Raj

Hall-Ticket – 2303A54043

Lab 9 – Code Review and Quality: Using AI to improve code

quality and readability

Lab Objectives:

• To apply AI-based prompt engineering for code review and quality

improvement.

• To analyze code for readability, logic, performance, and

maintainability issues.

• To use Zero-shot, One-shot, and Few-shot prompting for improving

code quality.

• To evaluate AI-generated improvements using standard coding

practices.

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

to:

• Review and improve code quality using AI tools.

• Identify syntax, logic, and performance issues in code.

• Refactor code to improve readability and maintainability.

• Compare AI outputs generated using different prompting techniques.

Problem Statement 1: AI-Assisted Bug Detection

Scenario: A junior developer wrote the following Python function to

calculate factorials:

```
def factorial(n):
result = 1
for i in range(1, n):
result = result * i
return result
```

Instructions:

1. Run the code and test it with factorial(5).

2. Use an AI assistant to:

o Identify the logical bug in the code.

o Explain why the bug occurs (e.g., off-by-one error).

o Provide a corrected version.

3. Compare the AI's corrected code with your own manual fix.

4. Write a brief comparison: Did AI miss any edge cases (e.g.,

negative numbers, zero)?

Expected Output:

Corrected function should return 120 for factorial(5).

Solution:

1. Run the code and test it with factorial(5).

```
def factorial(n):
    result = 1
    for i in range(1, n):
        result = result * i
    return result
```

this is inncorrect code its giving output 24 that's wrong

2. Use an AI assistant to:

o Identify the logical bug in the code.

o Explain why the bug occurs (e.g., off-by-one error).

o Provide a corrected version.

```
for i in range(1, n):
```

- `range(1, n)` iterates from `1` **up to but not including** `n`
- For `n = 5`, the loop multiplies:
`1 × 1 × 2 × 3 × 4 = 24`
- The number `5` is never included in the multiplicatiom

3. Compare the AI's corrected code with your own manual fix.

```
def factorial(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

this is correct code ai generated code its giving the right output that is 120

4. Write a brief comparison: Did AI miss any edge cases (e.g.,

negative numbers, zero)?

My Manual Fix

My manual correction would be **identical**:

- Extend the loop range to `n + 1`
- Keep the logic simple and readable

Did the AI Miss Any Edge Cases?

Yes — **the basic AI fix does not handle edge cases**:

Case	Behavior
`factorial(0)`	Works by accident (returns 1)
`factorial(-3)`	Incorrect (returns 1, should raise error)
Non-integer input	Not handled

5. Improved Version with Edge Case Handling (Best Practice)

```
def factorial(n):
    if not isinstance(n, int):
        raise TypeError("Factorial is only defined for integers")
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers")

    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

Problem Statement 2: Task 2 — Improving Readability &

Documentation

Scenario:The following code works but is poorly written:

.

```
def calc(a, b, c):
if c == "add":
return a + b
elif c == "sub":
return a – b
elif c == "mul":
return a * b
elif c == "div":
```

Instructions:

5. Use AI to:

o Critique the function’s readability, parameter naming, and lack of documentation.

o Rewrite the function with:

1. Descriptive function and parameter names.

2. A complete docstring (description, parameters, return value, examples).

3. Exception handling for division by zero.

4. Consideration of input validation.

6. Compare the original and AI-improved versions.

7. Test both with valid and invalid inputs (e.g., division by zero, non-string operation).

Expected Output:

A well-documented, robust, and readable function that handles

errors gracefully.

Solution:

1. Use AI to:

o Critique the function’s readability, parameter naming, and

lack of documentation.

o Rewrite the function with:

1. Descriptive function and parameter names.

2. A complete docstring (description, parameters,

return value, examples).

3. Exception handling for division by zero.

4. Consideration of input validation.

```
def calc(a, b, c):
    if c == "add":
        return a + b
    elif c == "sub":
        return a – b
    elif c == "mul":
        return a * b
    elif c == "div":
```

2. Compare the original and AI-improved versions.

```
def calculate_numbers(number1, number2, operation):
```

```
"""
Perform a basic arithmetic operation on two numbers.

Parameters:
number1 (int or float): The first numeric value.
number2 (int or float): The second numeric value.
operation (str): The operation to perform.
        Supported values: "add", "sub", "mul", "div".

Returns:
int or float: The result of the arithmetic operation.

Raises:
TypeError: If inputs are of incorrect types.
ValueError: If the operation is unsupported.
ZeroDivisionError: If division by zero is attempted.

Examples:
>>> calculate_numbers(10, 5, "add")
15
>>> calculate_numbers(10, 5, "div")
2.0
>>> calculate_numbers(10, 0, "div")
ZeroDivisionError
```

```
"""

# Input validation
if not isinstance(number1, (int, float)) or not isinstance(number2, (int, float)):
    raise TypeError("number1 and number2 must be numeric values.")

if not isinstance(operation, str):
    raise TypeError("operation must be a string.")

operation = operation.lower()

if operation == "add":
    return number1 + number2
elif operation == "sub":
    return number1 - number2
elif operation == "mul":
    return number1 * number2
elif operation == "div":
    if number2 == 0:
        raise ZeroDivisionError("Cannot divide by zero.")
    return number1 / number2
else:
    raise ValueError(f"Unsupported operation: {operation}")
```

3. Comparison: Original vs Improved

Aspect	Original Function	AI-Improved Function
Readability	Poor	Clear and structured
Parameter names	`a, b, c`	`number1, number2, operation`
Documentation	None	Complete docstring with examples
Error handling	None	Handles invalid input & division by zero
Input validation	None	Type and value checks
Maintainability	Low	High

4. Testing Both Versions

Testing the Original Function (Conceptual)

```
calc(10, 5, "add")      # Works
calc(10, 0, "div")      # Crashes / undefined behavior
calc(10, 5, 123)        # No error handling
calc("a", "b", "add")   # Type error
```

The original function fails silently or crashes without clear messages.

Testing the Improved Function

```
# Valid inputs
print(calculate_numbers(10, 5, "add"))   # 15
print(calculate_numbers(10, 5, "sub"))   # 5
print(calculate_numbers(10, 5, "mul"))   # 50
print(calculate_numbers(10, 5, "div"))   # 2.0

# Invalid operation
calculate_numbers(10, 5, "power")
# ValueError: Unsupported operation: power

# Division by zero
calculate_numbers(10, 0, "div")
# ZeroDivisionError: Cannot divide by zero.

# Invalid types
calculate_numbers("10", 5, "add")
# TypeError: number1 and number2 must be numeric values.
```

5. Final Outcome

✅The improved function is:

- Readable
- Self-documented
- Safer
- Easier to test and maintain
- More professional and production-ready

If you want, I can also:

- Refactor this using dictionaries instead of `if/elif`
- Convert it into a class-based calculator
- Add unit tests using `unittest` or `pytest`

Problem Statement 3: Enforcing Coding Standards

Scenario: A team project requires PEP8 compliance. A developer

submits:

def Checkprime(n):

for i in range(2, n):

if n % i == 0:

return False

return True

Instructions:

8. Verify the function works correctly for sample inputs.

9. Use an AI tool (e.g., ChatGPT, GitHub Copilot, or a PEP8 linter

with AI explanation) to:

o List all PEP8 violations.

o Refactor the code (function name, spacing, indentation,

naming).

10. Apply the AI-suggested changes and verify functionality is

preserved.

11. Write a short note on how automated AI reviews could streamline

code reviews in large teams.

Expected Output:

A PEP8-compliant version of the function, e.g.:

def check_prime(n):

for i in range(2, n):

if n % i == 0:

return False

return True

Solution:

1. Verify the Original Function Works Correctly

Original Code

```
def Checkprime(n):
for i in range(2, n):
if n % i == 0:
return False
return True
```

Functional Issues (before PEP8)

- As written, this code will **raise an `IndentationError`** in Python.
- Assuming correct indentation, the logic is *mostly* correct but:
    - It incorrectly reports `0` and `1` as prime numbers.
    - It does not validate input type.

Intended Behavior (with fixed indentation)

```
def Checkprime(n):
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
```

Sample Tests

```
Checkprime(2)    # True
Checkprime(3)    # True
Checkprime(4)    # False
Checkprime(5)    # True
Checkprime(1)    # Incorrectly returns True
```

So: **works for basic cases, but has logical edge-case issues.**

---

2. PEP8 Violations Identified

Using PEP8 standards (as an AI-assisted review / linter would):

Violations List

1. **Function naming**
   - `Checkprime` should be lowercase with underscores → `check_prime`
2. **Indentation**
   - Missing required indentation (4 spaces)
3. **Spacing**
   - No blank line between function definition and body (depending on context)
4. **Readability**
   - No docstring
   - Single-letter variable naming acceptable here (`n`, `i`) but undocumented
5. **Logic clarity**
   - No handling for numbers less than 2 (not a PEP8 violation, but a quality issue AI would flag)

---

3. Refactored, PEP8-Compliant Version

AI-Improved Code (PEP8 + Logic Fix)

```
def check_prime(n):
    """
    Determine whether a number is prime.
    Parameters:
    n (int): The number to check.

    Returns:
    bool: True if n is a prime number, False otherwise.
    """
    if not isinstance(n, int):
        return False

    if n < 2:
        return False

    for i in range(2, n):
        if n % i == 0:
            return False

    return True
```

---

4. Verify Functionality Is Preserved

Test Cases

```
print(check_prime(2))     # True
print(check_prime(3))     # True
print(check_prime(4))     # False
print(check_prime(5))     # True
print(check_prime(1))     # False
print(check_prime(0))     # False
print(check_prime(-7))    # False
print(check_prime(2.5))   # False
```

---

5. Short Note: AI in Large-Scale Code Reviews

Automated AI reviews can significantly streamline code reviews in large teams by consistently enforcing coding standards like PEP8, catching formatting and naming issues early, and reducing reviewer fatigue. AI tools provide immediate, objective feedback, allowing human reviewers to focus on higher-level concerns such as architecture, performance, and business logic. This leads to faster review cycles, fewer style-related comments, and more maintainable codebases overall.

Final Deliverable (PEP8-Compliant Function)

```
def check_prime(n):
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
```

Problem State ment 4: AI as a Code Reviewer in Real Projects

Scenario:

In a GitHub project, a teammate submits:

```
def processData(d):
return [x * 2 for x in d if x % 2 == 0]
```

Instructions:

1. Manually review the function for:

o Readability and naming.

o Reusability and modularity.

o Edge cases (non-list input, empty list, non-integer elements).

2. Use AI to generate a code review covering:

a. Better naming and function purpose clarity.

b. Input validation and type hints.

c. Suggestions for generalization (e.g., configurable multiplier).

3. Refactor the function based on AI feedback.

4. Write a short reflection on whether AI should be a standalone reviewer or an assistant.

Expected Output:

An improved function with type hints, validation, and clearer intent, e.g.:

```
from typing import List, Union
```

```
def double_even_numbers(numbers: List[Union[int,
float]]) -> List[Union[int, float]]:
if not isinstance(numbers, list):
raise TypeError("Input must be a list")
return [num * 2 for num in numbers if isinstance(num,
(int, float)) and num % 2 == 0]
```

Problem Statement 5: — AI-Assisted Performance Optimization

Scenario: You are given a function that processes a list of integers, but

it runs slowly on large datasets:

```
def sum_of_squares(numbers):
total = 0
for num in numbers:
total += num ** 2
return total
```

Instructions:

1. Test the function with a large list (e.g., range(1000000)).

2. Use AI to:

o Analyze time complexity.

o Suggest performance improvements (e.g., using built-in

functions, vectorization with NumPy if applicable).

o Provide an optimized version.

3. Compare execution time before and after optimization.

4. Discuss trade-offs between readability and performance.

Expected Output:

An optimized function, such as:

def sum_of_squares_optimized(numbers):

return sum(x * x for x in numbers)

Solution:

Submitted Code

```
def processData(d):
    return [x * 2 for x in d if x % 2 == 0]
```

---

1. Manual Review

Readability & Naming

- `processData` is vague — *what* processing?
- Parameter `d` gives no clue about expected data type or contents.
- No docstring or comments.

Reusability & Modularity

- Hardcoded behavior: always doubles values and filters evens.
- Not reusable for different multipliers or conditions.

Edge Cases

- ❌Non-list input → crashes
- ❌Empty list → works, but undocumented
- ❌Non-integer elements → `% 2` raises `TypeError`
- ❌Mixed numeric types → unclear intent

---

2. AI-Generated Code Review Feedback

a. Better Naming & Purpose Clarity

- Rename function to describe intent, e.g. `double_even_numbers`
- Rename parameter to `numbers`

b. Input Validation & Type Hints

- Validate input is a list
- Ensure elements are numeric
- Add type hints for maintainability and tooling support

c. Generalization Suggestions

- Allow configurable multiplier instead of hardcoded `2`
- Make function reusable across contexts

---

3. Refactored Function (AI-Improved)

```
from typing import List, Union


def multiply_even_numbers(
    numbers: List[Union[int, float]],
    multiplier: Union[int, float] = 2
) -> List[Union[int, float]]:
    """
    Multiply even numbers in a list by a given multiplier.

    Parameters:
    numbers (list of int or float): Input list of numbers.
    multiplier (int or float): Value to multiply even numbers by.

    Returns:
    list of int or float: Processed list containing multiplied even numbers.
    """
    if not isinstance(numbers, list):
        raise TypeError("Input must be a list.")

    result = []
    for num in numbers:
        if isinstance(num, (int, float)) and num % 2 == 0:
            result.append(num * multiplier)

    return result
```

Example Usage

```
multiply_even_numbers([1, 2, 3, 4])        # [4, 8]
multiply_even_numbers([2, 4, 6], 3)        # [6, 12, 18]
multiply_even_numbers([])                 # []
```

---

4. Reflection: AI as Reviewer vs Assistant

AI should act as an **assistant**, not a standalone reviewer. While AI excels at catching naming issues, style violations, edge cases, and suggesting refactors, it lacks full context about business rules, architectural intent, and trade-offs. Human reviewers provide judgment, domain understanding, and accountability, while AI accelerates reviews by handling repetitive and mechanical feedback. The best results come from combining both.

Problem Statement 5 — AI-Assisted Performance Optimization

Original Function

```
def sum_of_squares(numbers):
    total = 0
    for num in numbers:
        total += num ** 2
    return total
```

---

1. Testing with Large Input

```
numbers = range(1_000_000)
sum_of_squares(numbers)
```

✔Works correctly

✖Slower for very large datasets due to Python loop overhead

---

2. AI Analysis

Time Complexity

- **O(n)** — optimal, cannot be reduced asymptotically
- Bottleneck is Python's explicit loop

Performance Improvement Suggestions

- Use generator expressions with `sum()` (faster in CPython)
- Avoid repeated attribute lookups
- Optional: NumPy vectorization (only beneficial if already using NumPy)

---

3. Optimized Version

Pythonic Optimization (Best Trade-off)

```
def sum_of_squares_optimized(numbers):
    return sum(num * num for num in numbers)
```

Optional NumPy Version (for data science workloads)

```
import numpy as np

def sum_of_squares_numpy(numbers):
    arr = np.array(numbers)
    return int(np.sum(arr * arr))
```

---

4. Performance Comparison (Conceptual)

Version	Speed	Readability	Dependencies
Original loop	Slowest	High	None
Generator + sum	Faster	Very high	None
NumPy	Fastest (large arrays)	Medium	NumPy

---

5. Trade-offs: Readability vs Performance

- The generator-based version improves speed **without sacrificing readability**
- NumPy offers maximum performance but increases complexity and dependencies
- For most production code, the optimized pure-Python version is ideal

---

✓Final Deliverables

Improved, Readable, Optimized Function

```
def sum_of_squares_optimized(numbers):
    return sum(num * num for num in numbers)
```