Assignment 3.1

Name: P. Umesh Reddy Roll.no: 2403A510F9

Batch.no: 06 course: AI Assisted coding

Lab Experiment: Prompt Engineering – Improving Prompts and Context Management (0.5 marks)

Objective

To explore how prompt design and context influence AI-generated outputs and to learn techniques to improve AI responses.

Tools Required

● GitHub Copilot / Google Gemini / ChatGPT
● VS Code / Google Colab
● Internet access

Procedure

1. Select a simple task: "Write a Python function to check if a number is prime."
2. Use different prompting strategies to generate the solution:
a) Zero-Shot – no examples.
b) One-Shot – one example provided.
c) Few-Shot – multiple examples provided.
d) Context-Managed – detailed prompt with constraints and instructions.
3. Record AI responses and refine prompts to improve code quality.
4. Request AI to optimize the logic for efficiency.
5. Compare results and document improvements.

1. Sample Prompts

● Zero-Shot:

Write a Python function to check if a number is prime.

● One-Shot:

Example: Input: 5 → Output: Prime. Now, write a function to check if a number is prime.

● Few-Shot:

Example 1: Input: 7 → Output: Prime

Example 2: Input: 10 → Output: Not Prime

Example 3: Input: 2 → Output: Prime

Generate the function accordingly.

● Context-Managed (With Optimization)

Lab Report

Objective

To explore how prompt design (zero-shot, one-shot, few-shot, and context-managed) influences AI-generated solutions, and refine prompts to improve response quality and efficiency.

Tools Required

- ChatGPT / Gemini / Copilot
- Google Colab / VS Code
- Internet access

Procedure & Results

1. Zero-Shot Prompt

Prompt:

> Write a Python function to check if a number is prime.

AI Response:

```python
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
```

Observation:

- Works, but inefficient (loops up to `n-1`).

---

2. One-Shot Prompt

Prompt:

> Example: Input: 5 → Output: Prime.
> Now, write a function to check if a number is prime.

```python
def is_prime(n):
    if n <= 1:
        return "Not Prime"
    for i in range(2, n):
        if n % i == 0:
            return "Not Prime"
    return "Prime"
```

Observation:

- AI adapts to return `"Prime"` / `"Not Prime"` (closer to requirement).
- Still inefficient looping till `n-1`.

---

3. Few-Shot Prompt

Prompt:

> Example 1: Input: 7 → Output: Prime
> Example 2: Input: 10 → Output: Not Prime
> Example 3: Input: 2 → Output: Prime
> Generate the function accordingly.

AI Response:

python

```python
def is_prime(n):
    if n <= 1:
        return "Not Prime"
    if n == 2:
        return "Prime"
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return "Not Prime"
    return "Prime"
```

Observation:

- Optimized (loops only till √n).
- Handles `2` correctly.
- Improvement over previous attempts.

4. Context-Managed Prompt (with Optimization)

Prompt:

> Write a Python function `is_prime(n)` that:
>
> - **"Returns `"Prime"` or `"Not Prime"`"**
> - **"Efficiently checks only up to √n"**
> - **"Handles edge cases (n ≤ 1, even numbers)"**
> - **"Optimized for performance"**

AI Response (Optimized):

python

```python
def is_prime(n):
    if n <= 1:
        return "Not Prime"
    if n == 2:
        return "Prime"
    if n % 2 == 0:
        return "Not Prime"
    for i in range(3, int(n**0.5) + 1, 2):
        if n % i == 0:
            return "Not Prime"
    return "Prime"
```

Observation:

- Best version:
  - √n optimization
  - Skips even numbers
  - Handles edge cases

---

5. Comparative Reflection

Prompt Type	Output Quality	Efficiency	Adaptability
Zero-Shot	Correct but basic	Low (loops to n-1)	Minimal
One-Shot	Better formatting	Low	Medium
Few-Shot	√n optimization	Good	Strong
Context-Managed	Fully optimized	High	Very Strong

Task: Mobile Data Usage Billing Application

Objective:

Use Python programming and AI-assisted coding tools to create an application that simulates mobile data billing for a telecom service provider.

Instructions

1. Use GitHub Copilot or Google Gemini to assist in writing the program.
2. Read the following inputs from the user:

o Data Consumed (in GB)

o Plan Type (Prepaid / Postpaid)

o Additional Services Used (e.g., caller tune, OTT subscription, etc.)

3. Implement billing logic to calculate:

o DC (Data Charges) – charges based on data consumption

o VC (Value-added Charges) – charges for additional services

o Tax – applicable tax on the total bill

4. Display an itemized bill showing:
o Plan Type
o Data Usage and Charges
o Value-added Services and Charges
o Tax
o Total Bill Amount

Requirements

● Students must refer to their actual mobile bill for charge structure (data cost, service fees, taxes) to make the program realistic.

● AI assistance (Copilot/Gemini) must be used to generate and refine the initial code.

Deliverables

● AI prompts used for code generation.

● AI-generated Python code and any optimized version.

● Screenshots of:
o AI interactions
o Program execution and output
o Comparison with the student's actual mobile bill.

Python

```python
# Function to calculate the mobile bill
def calculate_mobile_bill():
    """Calculates and displays an itemized mobile data bill."""

    # Define realistic charges (these can be updated based on a real bill)
    DATA_CHARGE_PER_GB_PREPAID = 15  # per GB after plan data limit
    DATA_CHARGE_PER_GB_POSTPAID = 10  # per GB after plan data limit
    FIXED_CHARGE_PREPAID = 199  # Example base plan cost
    FIXED_CHARGE_POSTPAID = 299 # Example base plan cost
    TAX_RATE = 0.18  # 18% GST (Goods and Services Tax) in India
    DATA_INCLUDED_IN_PLAN = 10 # Data included in the base plan (in GB)

    # Dictionary for value-added services and their costs
    value_added_services = {
        'caller tune': 30.00,
        'ott subscription': 100.00,
```

Python

```python
        'ott subscription': 100.00,
        'cloud storage': 50.00,
        'gaming pass': 75.00
    }

    # --- User Input and Validation ---
    print("Welcome to the Mobile Data Billing Application.")

    try:
        data_consumed = float(input("Enter data consumed (in GB): "))
        if data_consumed < 0:
            print("Error: Data consumed cannot be negative. Please try again.")
            return

        plan_type_input = input("Enter Plan Type (Prepaid / Postpaid): ").strip().l
        if plan_type_input not in ['prepaid', 'postpaid']:
            print("Error: Invalid plan type. Please enter 'Prepaid' or 'Postpaid'.'
```

```python
        return

    additional_services_input = input("Enter Additional Services Used (e.g., ca
except ValueError:
    print("Error: Invalid input for data consumed. Please enter a number.")
    return

# --- Billing Logic ---
data_charges = 0
if data_consumed > DATA_INCLUDED_IN_PLAN:
    excess_data = data_consumed - DATA_INCLUDED_IN_PLAN
    if plan_type_input == 'prepaid':
        data_charges = excess_data * DATA_CHARGE_PER_GB_PREPAID
    elif plan_type_input == 'postpaid':
        data_charges = excess_data * DATA_CHARGE_PER_GB_POSTPAID

# Calculate value-added charges
```

```python
# Calculate value-added charges
value_added_charges = 0
services_used = [service.strip() for service in additional_services_input.split

# Store found services and their costs
found_services = {}

for service in services_used:
    if service in value_added_services:
        charge = value_added_services[service]
        value_added_charges += charge
        found_services[service] = charge
    else:
        if service: # Only print for non-empty strings
            print(f"Warning: '{service}' is not a recognized service. No charge
```

```python
# Determine fixed plan charge
fixed_plan_charge = 0
if plan_type_input == 'prepaid':
    fixed_plan_charge = FIXED_CHARGE_PREPAID
elif plan_type_input == 'postpaid':
    fixed_plan_charge = FIXED_CHARGE_POSTPAID

# Calculate total sub-bill before tax
sub_total = fixed_plan_charge + data_charges + value_added_charges

# Calculate tax
tax = sub_total * TAX_RATE

# Calculate total bill
total_bill_amount = sub_total + tax
```

```python
# --- Display Itemized Bill ---
print("\n" + "="*40)
print("            ITEMIZED MOBILE BILL")
print("="*40)
print(f"Plan Type:                  {plan_type_input.capitalize()}")
print("-" * 40)

# Data Charges
print(f"Data Usage:                 {data_consumed:.2f} GB")
print(f"Base Plan Charge:           ₹{fixed_plan_charge:.2f}")
if data_consumed > DATA_INCLUDED_IN_PLAN:
    print(f"Excess Data Charge:         ₹{data_charges:.2f} ({data_consumed - DATA_
print("-" * 40)

# Value-added Services
print("Value-added Services:")
```

```python
if found_services:
    for service, charge in found_services.items():
        print(f"  - {service.capitalize():<25} ₹{charge:.2f}")
else:
    print("  - No additional services used.")
print("-" * 40)

# Final Summary
print(f"Sub-Total (before Tax):      ₹{sub_total:.2f}")
print(f"Applicable Tax ({TAX_RATE*100:.0f}%):        ₹{tax:.2f}")
print("="*40)
print(f"TOTAL BILL AMOUNT:           ₹{total_bill_amount:.2f}")
print("="*40)

un the program
__name__ == "__main__":
 calculate_mobile_bill()
```

Screenshots and Comparison

This screenshot would show a conversation with Google Gemini where the user provides prompts and the AI generates and refines the Python code, as described above.

This screenshot would display the terminal or command prompt output of the Python script running. It would show the user inputs and the final itemized bill, similar to the output format in the code.

This screenshot would show a real mobile bill from a telecom provider, highlighting key sections like:

- Plan Name/Type
- Data Usage details (including excess charges)
- Value-added services
- Applicable taxes (e.g., GST)
- Total amount due

Comparison and Explanation

The Python program's structure is modeled after a typical real-world mobile bill. The code successfully emulates the key components:

- **Plan Type and Fixed Charges:** The program distinguishes between prepaid and postpaid plans and applies a fixed base charge, mirroring how telecom providers structure their plans.
- **Data Usage and Charges:** It calculates charges for **excess data consumption** beyond a set limit, which is a standard practice. The program uses different per-GB rates for different plan types, adding to the realism.
- **Value-added Services:** The use of a dictionary to store service prices is an efficient way to handle multiple add-ons. The program iterates through the user's input and totals the costs,

just as a real bill would.

- **Tax Calculation:** The program correctly applies a percentage-based tax (18% GST in the example) to the sub-total (base plan, data, and value-added charges), which is a crucial part of any billing system.
- **Itemized Bill Display:** The final output is formatted to look like a professional bill, with separate sections for each charge component and a clear final total, making it easy for the user to understand the breakdown of their costs.

Task: Develop an LPG Billing System (1.0 Marks)

Objective

Apply your Python programming skills and utilize AI-assisted coding tools to build an application that calculates the LPG bill based on specified customer inputs and billing parameters.

Instructions

1. Use GitHub Copilot or Google Gemini to assist in writing and refining the program.
2. Read the following user inputs:
o Cylinder Type (Domestic 14.2 kg / Domestic 5 kg / Commercial 19 kg / Commercial 47.5 kg)
o Number of Cylinders Booked
o Subsidy Amount (applicable only for domestic cylinders)
3. Refer to the given LPG Price List to determine the price per cylinder:
o Domestic LPG (14.2 kg) → ₹905.00
o Domestic LPG (5 kg) → ₹335.50
o Commercial LPG (19 kg) → ₹1,886.50
o Commercial LPG (47.5 kg) → ₹4,712.00

Week2 -
Monday

o Delivery Charges (₹10 to ₹50)
4. Implement the billing formula:

Bill Amount = (Price per Cylinder × Quantity) - Subsidy (if applicable)
+ Delivery Charges

5. Calculate and display an itemized bill including:

● Cylinder Type
● Number of Cylinders
● Base Amount
● Subsidy
● Delivery Charges
● Total Bill Amount

Deliverables

● A report containing:

o AI prompts used to generate the program
o AI-generated Python code
o Line-by-line explanation of the code

```
import random

def calculate_lpg_bill():
    """Calculates and displays an itemized LPG bill based on user inputs
        ."""

    # Define LPG prices
    lpg_prices = {
        'domestic 14.2 kg': 905.00,
        'domestic 5 kg': 335.50,
        'commercial 19 kg': 1886.50,
        'commercial 47.5 kg': 4712.00
    }

    print("Welcome to the LPG Billing System!")
    print("Available Cylinder Types:")
    for cylinder_type in lpg_prices:
        print(f"  - {cylinder_type.title()}")

    # Get user inputs with validation
```

```
while True:
    cylinder_type_input = input("\nEnter Cylinder Type: ").strip
        ().lower()
    if cylinder_type_input in lpg_prices:
        break
    print("Error: Invalid cylinder type. Please choose from the list
        .")

while True:
    try:
        num_cylinders = int(input("Enter Number of Cylinders Booked:
            "))
        if num_cylinders > 0:
            break
        else:
            print("Error: Number of cylinders must be a positive
                integer.")
    except ValueError:
        print("Error: Invalid input. Please enter a number.")

is_domestic = cylinder_type_input.startswith('domestic')
```

```python
subsidy = 0
if is_domestic:
    while True:
        try:
            subsidy = float(input("Enter Subsidy Amount (₹): "))
            if subsidy >= 0:
                break
            else:
                print("Error: Subsidy amount cannot be negative.")
        except ValueError:
            print("Error: Invalid input. Please enter a number.")

# Calculate billing components
price_per_cylinder = lpg_prices[cylinder_type_input]
base_amount = price_per_cylinder * num_cylinders

# Randomly determine delivery charges
delivery_charges = random.uniform(10.00, 50.00)

# Calculate total bill
total_bill = base_amount - subsidy + delivery_charges
```

```
    # --- Display Itemized Bill ---
    print("\n" + "="*40)
    print("            LPG BILL STATEMENT")
    print("="*40)
    print(f"Cylinder Type:               {cylinder_type_input.title()}")
    print(f"Number of Cylinders:         {num_cylinders}")
    print("-" * 40)
    print(f"Base Amount:                 ₹{base_amount:.2f}")
    if is_domestic:
        print(f"Subsidy:                 - ₹{subsidy:.2f}")
    else:
        print(f"Subsidy:                 - ₹{0.00:.2f}")
    print(f"Delivery Charges:            + ₹{delivery_charges:.2f}")
    print("-" * 40)
    print(f"TOTAL BILL AMOUNT:           ₹{total_bill:.2f}")
    print("="*40)

# Run the program
if __name__ == "__main__":
    calculate_lpg_bill()
```

The following prompts were used to guide the AI in generating the Python code:

- **Initial Code Generation:** "Write a Python program for an LPG billing system. The program should ask the user for the cylinder type (Domestic 14.2 kg, Domestic 5 kg, Commercial 19 kg, Commercial 47.5 kg), the number of cylinders, and the subsidy amount. It should then calculate the total bill based on a given price list and display an itemized bill."
- **Adding Price List and Formula:** "Integrate the specific LPG prices into the Python script: Domestic 14.2 kg at ₹905.00, Domestic 5 kg at ₹335.50, Commercial 19 kg at ₹1,886.50, and Commercial 47.5 kg at ₹4,712.00. Also, add a calculation for random delivery charges between ₹10 and ₹50. The bill amount should be (Price per Cylinder × Quantity) - Subsidy + Delivery Charges."
- **Refining User Input and Output:** "Improve the user interface for the LPG billing system. Make the input clear and add validation to handle incorrect cylinder types or non-numeric inputs. Format the final itemized bill to be easy to read, showing the cylinder type, quantity, base amount, subsidy, delivery charges, and total."

Line-by-Line Code Explanation

- `import random`: This line imports the `random` module, which is used to generate a random delivery charge.
- `def calculate_lpg_bill():`: This defines a function named `calculate_lpg_bill` that encapsulates the entire program logic.
- `lpg_prices = { ... }`: A dictionary is created to store the fixed prices for each cylinder type. This makes it easy to look up the price based on the user's input.
- `print("Welcome...")`: The program starts by printing a welcome message and listing the available cylinder types for the user's reference.
- `while True: ...`: This is a loop for input validation. It continues to prompt the user for the cylinder type until a valid option from the `lpg_prices` dictionary is entered.
- `try: ... except ValueError: ...`: These blocks handle potential errors. The program attempts to convert the user's input for the number of cylinders and subsidy into a number. If the user enters text, it will catch the `ValueError` and prompt them to enter a number.
- `is_domestic = cylinder_type_input.startswith('domestic')`: This line checks if the cylinder type starts with the word "domestic". This is used to determine whether or not to apply a subsidy.
- `subsidy = 0; if is_domestic: ...`: A variable for the subsidy is initialized to 0. If the cylinder is domestic, the program then prompts the user for the subsidy amount.
- `price_per_cylinder = lpg_prices[cylinder_type_input]`: The price for the selected cylinder type is retrieved from the `lpg_prices` dictionary.
- `base_amount = price_per_cylinder * num_cylinders`: The total cost of the cylinders is calculated by multiplying the price per cylinder by the quantity.
- `delivery_charges = random.uniform(10.00, 50.00)`: A random delivery charge between ₹10.00 and ₹50.00 is generated using the `random.uniform()` function.

- `delivery_charges = random.uniform(10.00, 50.00)`: A random delivery charge between ₹10.00 and ₹50.00 is generated using the `random.uniform()` function.
- `total_bill = base_amount - subsidy + delivery_charges`: The final bill is calculated using the specified formula.
- `print("\n" + "="*40) ...`: The rest of the code is dedicated to formatting and displaying the itemized bill in a clear, easy-to-read layout. The `:.2f` formatting ensures that all monetary values are displayed with two decimal places.
- `if __name__ == "__main__":`: This is a standard Python construct that ensures the `calculate_lpg_bill()` function is called only when the script is executed directly.