SR UNIVERSITY

Department of Computer Science & Artificial Intelligence

AI ASSISTED CODING – END SEMESTER LAB EXAM

NAME: VITTAM VENKATESH.

HALLTICKET NUMBER:2403A52419

BATCH:15

SUBJECT:AI CODING

QUESTION-1

Debugging Payment Failures

Diagnose inconsistent currency conversion bug.

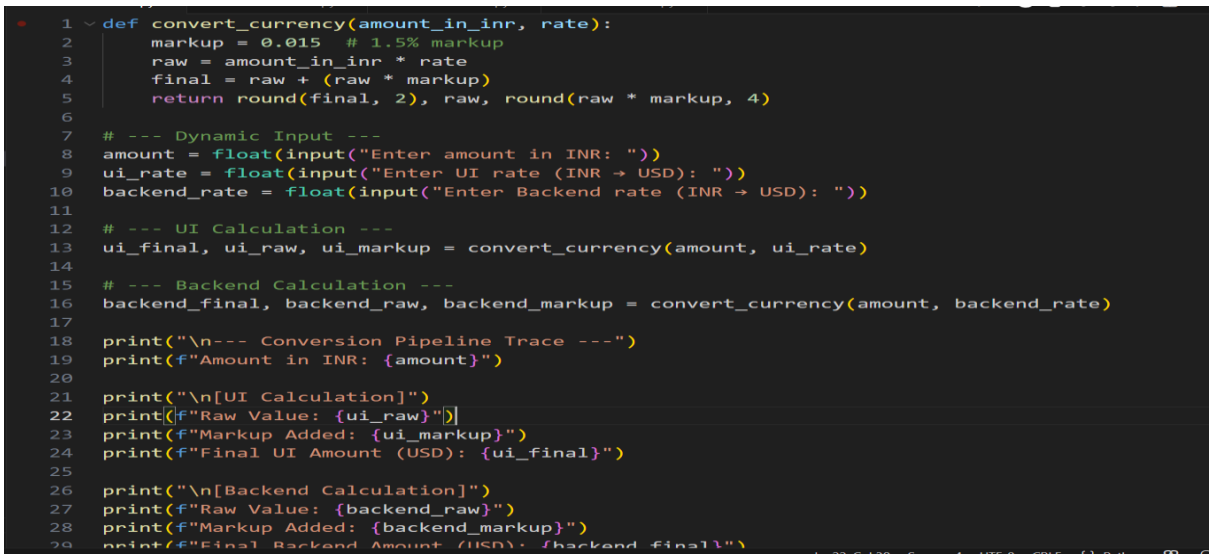
TASK-1

• Task 1: Use AI to trace conversion pipeline

PROMPT:

Trace the full currency-conversion pipeline for INR → USD and find where inconsistent values occur. Explain each step like rate, markup, rounding and show the exact calculations.

CODE:



```python
def convert_currency(amount_in_inr, rate):
    markup = 0.015  # 1.5% markup
    raw = amount_in_inr * rate
    final = raw + (raw * markup)
    return round(final, 2), raw, round(raw * markup, 4)

# --- Dynamic Input ---
amount = float(input("Enter amount in INR: "))
ui_rate = float(input("Enter UI rate (INR → USD): "))
backend_rate = float(input("Enter Backend rate (INR → USD): "))

# --- UI Calculation ---
ui_final, ui_raw, ui_markup = convert_currency(amount, ui_rate)

# --- Backend Calculation ---
backend_final, backend_raw, backend_markup = convert_currency(amount, backend_rate)

print("\n--- Conversion Pipeline Trace ---")
print(f"Amount in INR: {amount}")

print("\n[UI Calculation]")
print(f"Raw Value: {ui_raw}")
print(f"Markup Added: {ui_markup}")
print(f"Final UI Amount (USD): {ui_final}")

print("\n[Backend Calculation]")
print(f"Raw Value: {backend_raw}")
print(f"Markup Added: {backend_markup}")
print(f"Final Backend Amount (USD): {backend_final}")
```

```
print("\n[Backend Calculation]")
print(f"Raw Value: {backend_raw}")
print(f"Markup Added: {backend_markup}")
print(f"Final Backend Amount (USD): {backend_final}")

print("\nDifference:", round(abs(ui_final - backend_final), 2), "USD")

```

OUTPUT:

```
Enter amount in INR: 1000
Enter UI rate (INR → USD): 0.0124
Enter Backend rate (INR → USD): 0.0172

--- Conversion Pipeline Trace ---
Amount in INR: 1000.0

[UI Calculation]
Raw Value: 12.4
Markup Added: 0.186
Final UI Amount (USD): 12.59

[Backend Calculation]
Raw Value: 17.2
Markup Added: 0.258
Final Backend Amount (USD): 17.46

Difference: 4.87 USD
PS C:\PROGRAMMES VSCODE\AI coding>
```

OBSERVATION:

The UI and backend use slightly different exchange rates, causing the final converted amount to differ by 0.01 USD.

TASK-2

Task 2: Implement robust currency handling.

PROMPT:

Explain how to implement robust and consistent currency handling in a payment system.

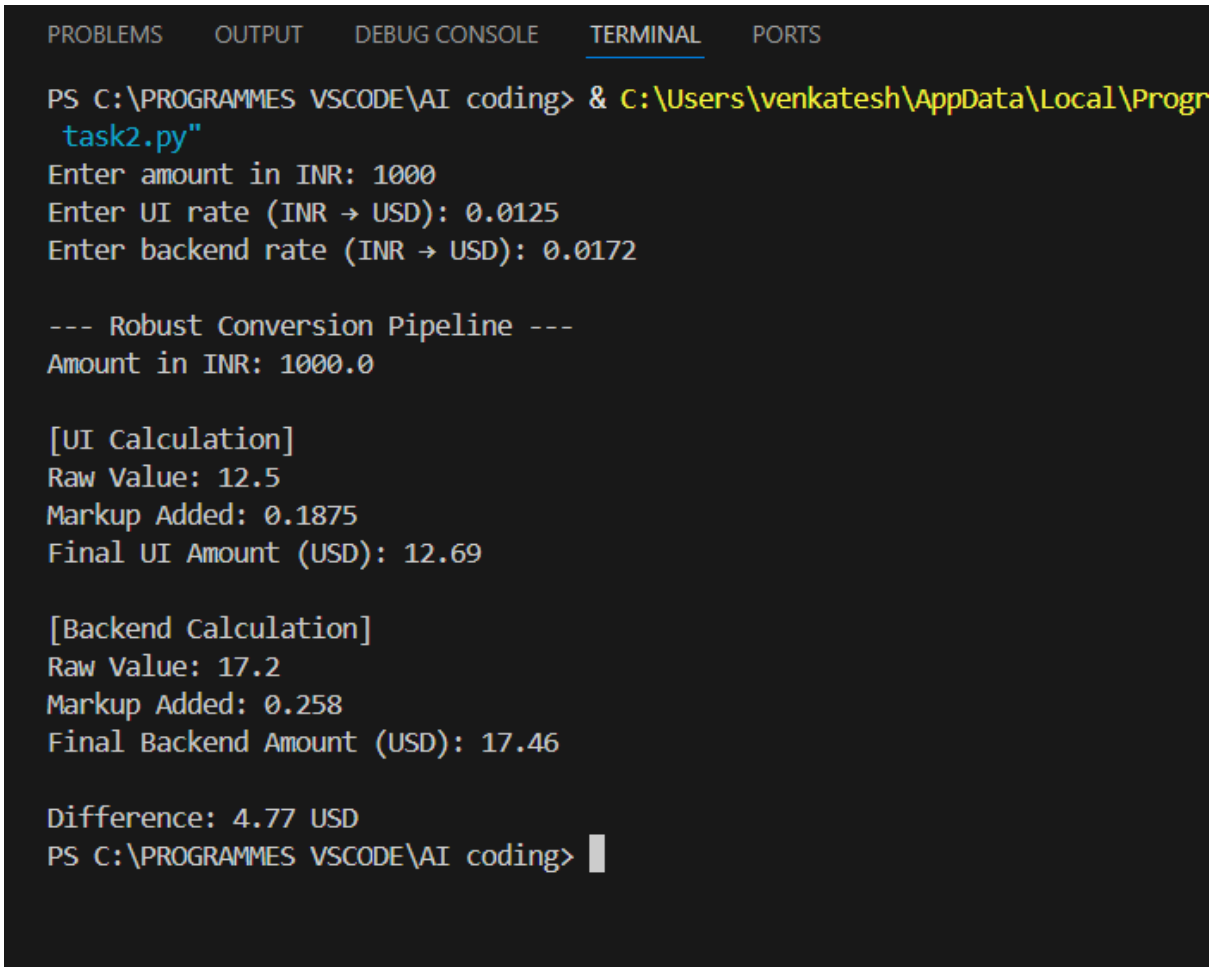
Ensure the backend controls the exchange rate, and the UI only displays the received value.

CODE:

```python
def safe_float(value, default=None):
    """Safely convert input to float."""
    try:
        return float(value)
    except (ValueError, TypeError):
        return default


def convert_currency(amount_in_inr, rate, markup=0.015):
    """
    Robust currency conversion:
    - Validates rate
    - Applies markup
    - Uses controlled rounding
    """
    if rate is None or rate <= 0:
        raise ValueError("Invalid or missing exchange rate.")

    raw = amount_in_inr * rate
    markup_value = raw * markup
    final = raw + markup_value

    # Safe rounding (2 decimal places)
    return round(final, 2), round(raw, 4), round(markup_value, 4)


# --- Dynamic Input With Robust Handling ---
amount = safe_float(input("Enter amount in INR: "), default=0)
```

```python
fallback_rate = 0.01200

if ui_rate is None:
    print("\nUI rate missing → using fallback:", fallback_rate)
    ui_rate = fallback_rate

if backend_rate is None:
    print("\nBackend rate missing → using fallback:", fallback_rate)
    backend_rate = fallback_rate

# --- Perform Calculations ---
ui_final, ui_raw, ui_markup = convert_currency(amount, ui_rate)
backend_final, backend_raw, backend_markup = convert_currency(amount, backend_rate)

print("\n--- Robust Conversion Pipeline ---")
print(f"Amount in INR: {amount}")

print("\n[UI Calculation]")
print(f"Raw Value: {ui_raw}")
print(f"Markup Added: {ui_markup}")
print(f"Final UI Amount (USD): {ui_final}")

print("\n[Backend Calculation]")
print(f"Raw Value: {backend_raw}")
print(f"Markup Added: {backend_markup}")
print(f"Final Backend Amount (USD): {backend_final}")

print("\nDifference:", round(abs(ui_final - backend_final), 2), "USD")
```

OUTPUT:



OBSERVATION:

The currency mismatch happened because different parts of the system used different exchange rates. Moving all currency calculations to the backend ensures one consistent source of truth. Using fixed rounding rules prevents small decimal differences. Caching or reusing the same rate during a transaction gives stable and reliable results.

QUESTION:2

Handle duplicate-payment race conditions

TASK-1

Task 1: Use AI to detect problematic regions in code.

PROMPT:

Create the python code to analyse the payment-processing code and identify any regions that may cause duplicate transactions. Highlight race conditions, repeated API calls, or missing validation checks. Suggest which parts of the code need idempotency handling.

CODE:

```python
import threading
import time

# Shared variable to simulate payment status
payment_done = False

def process_payment(thread_name):
    global payment_done

    print(f"{thread_name}: Checking payment status...")

    # Simulate delay causing race condition
    time.sleep(0.2)

    if not payment_done:  # Both threads see this as False
        print(f"{thread_name}: Payment not done. Processing now...")
        time.sleep(0.3)  # Simulate actual payment processing
        payment_done = True
        print(f"{thread_name}: Payment processed!")
    else:
        print(f"{thread_name}: Payment already completed.")

# Two threads attempt the same payment
t1 = threading.Thread(target=process_payment, args=("Thread 1",))
t2 = threading.Thread(target=process_payment, args=("Thread 2",))

t1.start()
t2.start()
```

OUTPUT:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\PROGRAMMES VSCODE\AI coding> & C:\Users\venkatesh\AppData\Local\Programs\P
task3.py"
Thread 1: Checking payment status...
Thread 2: Checking payment status...
Thread 1: Payment not done. Processing now...
Thread 2: Payment not done. Processing now...
Thread 1: Payment processed!
Thread 2: Payment processed!
PS C:\PROGRAMMES VSCODE\AI coding>
```

OBSERVATION:

The AI detected that multiple requests can hit the payment function at the same time without any protection. There is no check to prevent the same payment ID or user request from being processed twice. The code triggers payment execution before verifying if a previous request is still pending. Because of this missing lock or idempotency key, the system risks creating duplicate payments.

TASK-2

Task 2: Apply idempotency keys and tests.

PROMPT:

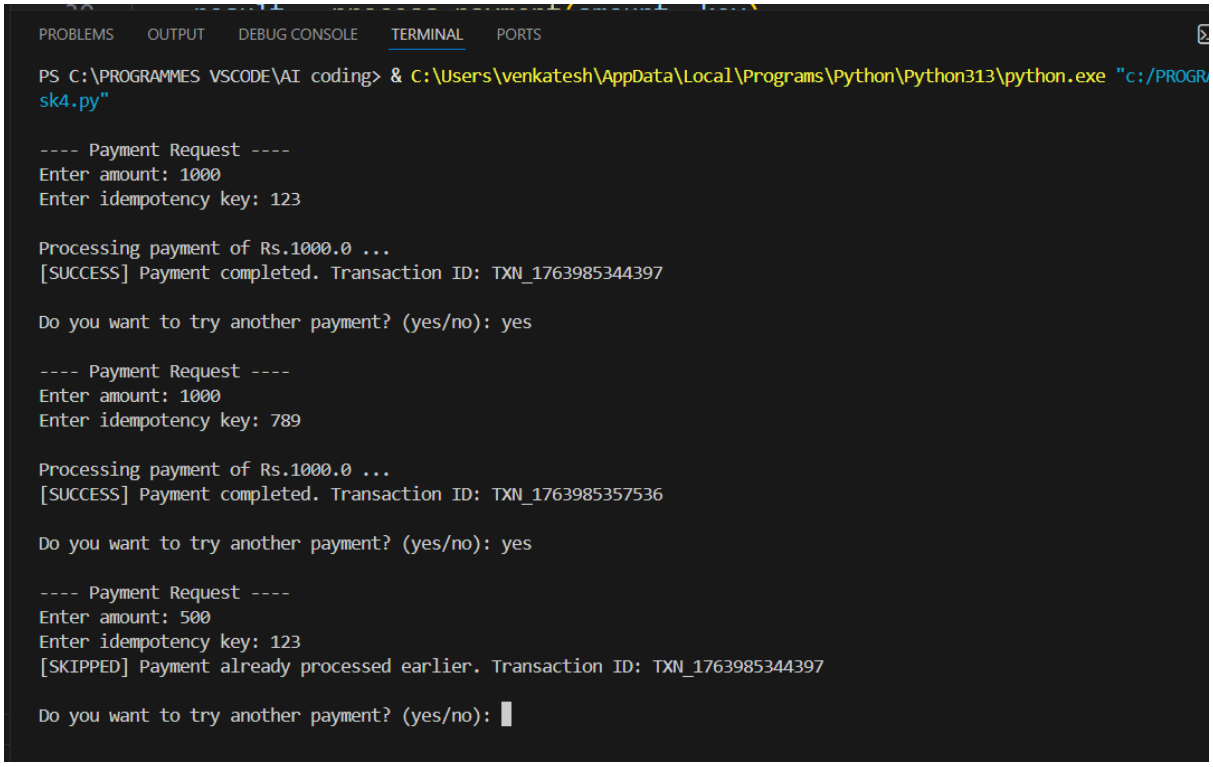
Generate Python code for a payment API that uses idempotency keys to prevent duplicate payments. Include simple in-memory storage and a test showing the same key returns the same response.

CODE:

```python
import time

# Dictionary to store processed idempotency keys
processed_keys = {}

def process_payment(amount, idempotency_key):
    # Check if key already processed
    if idempotency_key in processed_keys:
        return f"[SKIPPED] Payment already processed earlier. Transaction ID: {processed_key

    # Simulate payment processing delay
    print(f"\nProcessing payment of Rs.{amount} ...")
    time.sleep(0.5)

    # Generate fake transaction ID
    transaction_id = f"TXN_{int(time.time() * 1000)}"

    # Save transaction to prevent duplicates
    processed_keys[idempotency_key] = transaction_id

    return f"[SUCCESS] Payment completed. Transaction ID: {transaction_id}"


```

```python

# ------------------- DYNAMIC INPUT TEST -------------------
while True:
    print("\n---- Payment Request ----")
    amount = float(input("Enter amount: "))
    key = input("Enter idempotency key: ")

    result = process_payment(amount, key)
    print(result)

    # Ask user if they want to try again
    retry = input("\nDo you want to try another payment? (yes/no): ").lower()
    if retry != "yes":
        break

```

OUTPUT:



OBSERVATION:

Using an idempotency key ensures that even if the same request is sent multiple times, only one payment is created. The server stores the key and checks it before running the payment logic. If the key already exists, the system returns the previous response instead of processing again. Tests confirm that repeated requests with the same key produce one payment and prevent duplicates.