**1. My Approach Explanation**

**1.a To recover and fix the missing or incorrect data in the event-driven system, I’d follow these steps:**

* **Processing Events and Calculating Results**:  
  When an event (in this case, a number) comes in, I process it by performing a calculation (like squaring the number). The result of this calculation is then stored in an array called results and sent back to the RESULT\_TOPIC Kafka topic.
* **Error Handling**:  
  I’ve built in error handling to make sure that if there’s any issue with the event data (like a bad format), it’s caught, and the event isn’t processed incorrectly.
* **Back-calculating Missed or Incorrect Data**:  
  If some data was missed or processed incorrectly, I have a mechanism that runs when the system shuts down or after the event stream is done. It goes through the stored results, checks for any discrepancies (like an incorrect calculation), and redoes the calculation if necessary.

**1.b. Tools and Techniques**:

* + **Kafka** is used to transport events between the producer (which sends events) and the consumer (which processes them).
  + The **Confluent Kafka Python client** helps with consuming and producing events.
  + **JSON** is used to serialize and deserialize the event data.
  + Results are temporarily stored in an in-memory array (results).
  + If the results are found to be wrong, I re-run the correct calculations.

**1.c. Ensuring Accuracy**:  
To make sure everything is accurate; I check the processed results and recalculate them if there’s a mismatch. This ensures the correct data is produced, even if something goes wrong.

1. **Solution:**

**Code:**

from confluent\_kafka import Consumer, Producer

import json

# Kafka configurations

KAFKA\_BROKER = 'localhost:9092'

EVENT\_TOPIC = 'events'

RESULT\_TOPIC = 'results'

# Array to store processed results

results = []

# Function to process an event

def process\_event(event):

try:

# Deserialize event

event\_data = json.loads(event)

number = event\_data['number']

# Perform a calculation (e.g., square the number)

result = number \*\* 2

return result

except Exception as e:

print(f"Error processing event: {e}")

return None

# Initialize Kafka consumer

consumer\_config = {

'bootstrap.servers': KAFKA\_BROKER,

'group.id': 'recalc\_group',

'auto.offset.reset': 'earliest'

}

consumer = Consumer(consumer\_config)

# Subscribe to the event topic

consumer.subscribe([EVENT\_TOPIC])

# Initialize Kafka producer

producer\_config = {'bootstrap.servers': KAFKA\_BROKER}

producer = Producer(producer\_config)

# Function to back-calculate missing or incorrect data

def back\_calculate\_missed\_events():

print("Back-calculating missed or incorrect data...")

for event in results:

# In real use cases, check if the result is incorrect and recalculate

# Example: Add custom logic to identify incorrect results based on certain conditions

number = event['number']

correct\_result = number \*\* 2

if event['result'] != correct\_result:

print(f"Recalculating for number {number}")

event['result'] = correct\_result # Recalculate the result

# Function to store results in array

def store\_result(event, result):

results.append({'number': event['number'], 'result': result})

try:

print("Starting event processor...")

while True:

# Poll for messages

msg = consumer.poll(1.0) # Timeout of 1 second

if msg is None:

continue # No message received

if msg.error():

print(f"Consumer error: {msg.error()}")

continue

# Process the event

event = msg.value().decode('utf-8')

print(f"Received event: {event}")

result = process\_event(event)

if result is not None:

event\_data = json.loads(event)

# Store the processed result

store\_result(event\_data, result)

# Send the result to the results topic

result\_event = {'number': event\_data['number'], 'result': result}

producer.produce(RESULT\_TOPIC, json.dumps(result\_event))

print(f"Sent result: {result\_event}")

# Flush producer to ensure delivery

producer.flush()

except KeyboardInterrupt:

print("Shutting down event processor...")

# Back-calculate missing or incorrect data on shutdown

back\_calculate\_missed\_events()

finally:

consumer.close()

1. **Code Breakdown:**

**3.a Summarize my approach**

1. Event Processing: The script listens to events from Kafka, processes them (like squaring a number), and sends the results back to Kafka.
2. Back-calculation: If any result is incorrect, the back\_calculate\_missed\_events function will recalculate them.
3. Storing Results: All the processed results are temporarily stored in the results array, so they can be recalculated or used later.

**Why I Used it**

This approach fits naturally into an **event-driven system** like Kafka. It ensures:

1. **Real-time processing** – Events are processed as they arrive.
2. **Error correction without a database** – The back-calculation method fixes mistakes on shutdown.
3. **Scalability** – Kafka allows adding more consumers if event volume increases.
4. **Simplicity & efficiency** – It avoids unnecessary complexity while keeping data accurate.

**3.b Trade-offs and Limitations:**

* In-memory Storage: Storing results in an in-memory array (results) is fine for small-scale scenarios but won’t scale well for larger systems. Ideally, I’d use a more robust storage option like a database or distributed cache (e.g., Redis) for long-term storage.
* Event Processing Time: For a system processing millions of events per hour, storing everything in memory could be inefficient. To handle that, I’d move the results to persistent storage and use parallel processing to manage the load.
* **Error Handling**: Currently, the system only recalculates data when it shuts down or finishes processing. In a real-world scenario, I’d automate this process to check for errors continuously.

**3.c Alternative Approaches:**

* **Database**: If there was a database available, I’d store the event data there and use SQL queries to backtrack and recalculate missed data.
* **Logs**: If detailed logs were available, I could analyze them to find missed or faulty events.
* **Event Replay**: Kafka could also be used to replay the missed events, so I could reprocess them without worrying about missing data.
* **Replay Missed Events with Kafka Itself:** Instead of storing processed results in memory, I’d use Kafka’s built-in features to replay events whenever needed. This way, if something goes wrong, I can just reprocess past events instead of manually recalculating them.
* **Use a Fast Cache (Like Redis):** Instead of keeping results in an array, I’d store them in Redis (a fast temporary storage). If the result is wrong, I can update it quickly instead of waiting until shutdown.
* **Save Results in a File or Cloud Storage:** To prevent losing data on restart, I could periodically save results to a file or cloud storage (like AWS S3). That way, I could reload them if needed instead of relying only on memory

**3.d Scaling Considerations:**

* For scaling, I’d introduce parallel processing. I’d use multiple Kafka consumers to handle larger event streams. Additionally, I’d optimize the recalculation logic to avoid overloading the system with millions of events.
* If I were working with a system that processes millions of events per hour, I’d consider using a more advanced streaming platform like Apache Flink or Kafka Streams to handle the real-time processing efficiently.