**KAFKA**

It is a distributed event-streaming platform that is used for building real-time data pipeline and streaming applications.

One of the tools that opened way for microservice architecture

**Event:**

Consists of key, value, timestamp, metadata

Basically events will be fired and forgot by the producers and they no need to wait for response.

**Topic:**

Grouping same type of events.

**Partitions:**

we decide based on schema design

Eg. Orders based on area – US, UK, Asia

**Consumer-groups:**

Instance of consumers -> each consumer in the same group will be assigned each partition enabling parallel consumption.

Group-id is same for all the consumers in the same group

**Kafka-broker:**

Server that stores data(messages) in topics, manages message distribution to consumer.

**Fault-tolerance:**

Topic’s partitions are distributed across multiple brokers.

Each partition has a reader broker and multiple replicas

**KAFKA’s uniqueness when compared to the other message brokers:**

Other message brokers will remove the message once consumed by consumer. In kafka we can set retention period. (how long we need to store the messages)

In real time, Kafka can be compared to Netflix and Rabbitmq/Active MQ to normal television.

**Cluster management:**

Maintaining a registry of all active brokers in cluster.

**Leader election:**

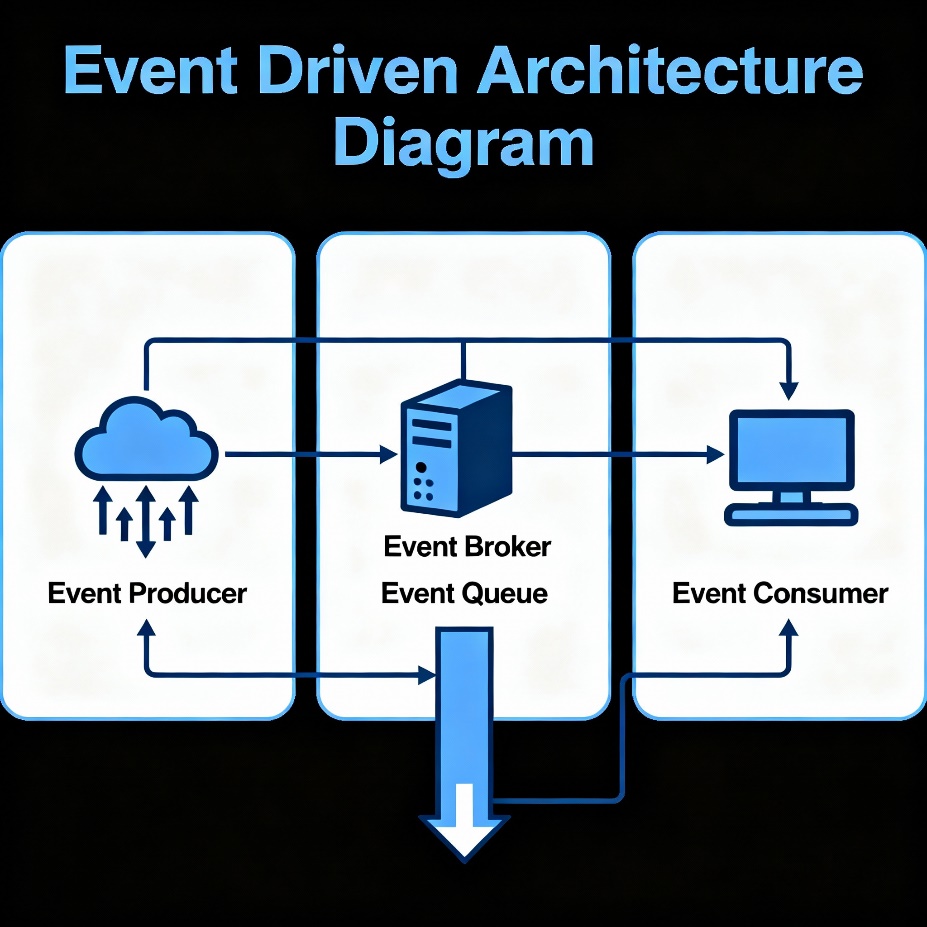
Each kafka partition has a leader broker. Zookeeper facilitates election of a leader.

Zookeeper-> centralized service for managing metdata and co-ordination tasks for distributed systems. It is actually an external dependency of kafka.

**But from kafka version 3.0 -> removed dependency on zookeeper.**

Metadata is now managed natively with kafka-broker.

**Event-driver Architecture:**



Event driven architecture pitfalls before kafka or before any broker services

1. Tight coupling
2. Reduced scalability – struggling to handle growing data and uses demands due to interdependencies.
3. Single point of failure – No backup or redundant instances.
4. No message persistence – Loss of data
5. Limited functionality

**Kafka – Advantages:**

* High throughput – kafka handles massive event streams with ease, ensuring no messages gets left behind.
* Fault tolerance – Guarantees resilience, high availability, even with failures.
* Scalability
* Real-time processing of data

In Kafka, the **foundation for message management, cluster node management, scalability, fault tolerance, and dynamic membership** is the core Kafka cluster architecture, primarily built around the following components:

**1. Broker**

* A broker is a single Kafka server responsible for storing and managing messages.
* Multiple brokers form a Kafka cluster, distributing partitions of topics among themselves.
* The broker is responsible for message management, persistence, and serving client (producer/consumer) requests.

**2. Cluster**

* A Kafka cluster consists of multiple brokers working together.
* This setup allows horizontal scaling (adding more brokers increases capacity) and built-in fault tolerance because data (topic partitions) can be replicated across brokers.
* There is always one broker acting as the controller (or leader) that manages cluster metadata and orchestrates dynamic membership (adding/removing brokers/partitions).

**3. Topics and Partitions**

* Topics provide logical separation of message streams.
* Each topic is divided into partitions, which are distributed across brokers for scalability and parallelism.
* Partitions are the foundation for both scalability (allowing parallel reads/writes) and fault tolerance (through replication).

**4. Replication**

* Each partition can be replicated across multiple brokers.
* One broker serves as the leader for that partition; others act as followers.
* This ensures messages are available even if a broker fails, providing fault tolerance.

**5. Controller (metadata management)**

* The controller (which now may be handled by KRaft in new versions, or by Zookeeper in legacy Kafka) elect leaders for partitions, tracks cluster changes, and manages dynamic membership.
* The controller coordinates operations like adding/deleting brokers and partitions for automatic rebalancing and resilience.

**Advantages of topic:**

* Message categorization – based on key, if same key -> same partitions
* Immutable logs – message in a topic are stored sequentially
* Multi-consumer access – Multiple consumers can read from same topic
* Decouple communication – Producer and consumers don’t interact directly but through topics
* Replication for higher availability

**Single Broker limitation:**

Storing an entire topic across one broker restricts its size and availability to the capacity of the single machine.

**Partitioning for scalability:**

Kafka partitions distribute a topic across multiple brokers, enabling horizontal scalability, fault tolerance and higher throughput.

**Advantages of partitioning of the topic:**

* Distribution
* Parallelism – parallel processing of messages
* Scalability
* Fault tolerance
* Ordering – messages within the partition are strictly ordered (FCFS)

28/09/2025

**INCREASING NO. OF PARTITIONS OF PREVIOUS TOPIC:**

**Command prompt:**

1. To alter the existing topic description:

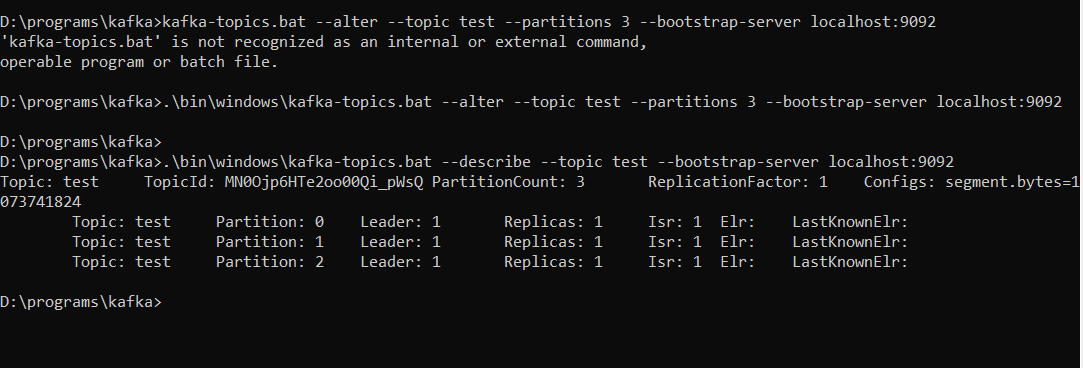
.\bin\windows\kafka-topics.bat --alter --topic your\_topic --partitions 3 --bootstrap-server localhost:9092

Formula for hashing:

partition\_id = hash(key) % total\_partitions

1. <--- To view the updated topic configuration -->

.\bin\windows\kafka-topics.bat --describe --topic test --bootstrap-server localhost:9092



From the command output, you see that all your partitions (0, 1, 2) list "Leader: 1", meaning they are currently all led by the same broker with ID 1. This means all read and write operations for these partitions are happening on broker 1.

**How Partition-to-Broker Mapping Works**

* The **Leader** field in the output from --describe indicates which broker currently hosts the active, authoritative copy ("leader") for each partition.
* If your cluster had multiple brokers, you would see different broker IDs (e.g., Leader: 2 or Leader: 3) for different partitions, meaning the load and data are spread.
* In your current setup, all partitions are being served by broker 1, so none of your partitions are in different brokers at the moment.

**Increasing no. of consumer(consumer-group) to enable parallel consumption of messages form above 3 partitions:**

With your topic upgraded to 3 partitions, you can leverage parallelism by creating a **consumer group** with 3 consumers. Kafka will automatically assign each consumer to one partition, enabling parallel processing of your data stream.

**How Kafka Parallelism Works with Consumer Groups**

* If you start 3 consumers with the **same group ID** (i.e., they belong to the same consumer group), Kafka ensures each partition is read by only one consumer—a “partition-consumer” mapping.
* Messages from partition 0 will be streamed to consumer 1, partition 1 to consumer 2, and partition 2 to consumer 3. This happens automatically.
* If a consumer dies, Kafka will rebalance partitions among the remaining consumers.

**Doubts and clarifications:  
Qn 1**- okay which is better earliest or latest how can I decide which to have and also I didn't understand the concept of auto\_offset\_reset, enable\_auto\_commit, Please help me with a real time example.

Let’s break it down with real-world examples and explain how to choose between 'earliest' and 'latest', and what auto\_offset\_reset and enable\_auto\_commit mean.

**What is auto\_offset\_reset?**

* Definition: What should the Kafka consumer do if it starts from scratch (NO committed offset exists for your consumer group)?
* You can choose:
  + 'earliest': Read ALL existing messages from the start.
  + 'latest': Skip ALL existing messages, read ONLY new incoming ones.

**Real-World Example**

Imagine you deploy a new service to process order events:

* If you want to process all historical orders:  
  Set auto\_offset\_reset='earliest'. You’ll read every event that’s ever been written to that topic.
* If you only care about orders that happen after your service starts (future events):  
  Set auto\_offset\_reset='latest'. You’ll ignore any past orders/events and only get new ones as they occur.

Choosing which one:

* Use 'earliest' if you want to replay or bootstrap from historical data.
* Use 'latest' if you only want to process events happening now/future.

**What is enable\_auto\_commit?**

* Definition: Should Kafka automatically track and record your consumer’s "position" as it reads messages?
* Set to True:
  + Every few seconds, Kafka saves your position automatically. If your program crashes/restarts, it resumes from the last saved point.
* Set to False:
  + You must manually call consumer.commit() after processing records.

**Real-World Example**

You have a consumer that processes payments:

* With enable\_auto\_commit=True, even if your service restarts/crashes, you won’t reprocess the payment events you already read—the offset was saved!
* With enable\_auto\_commit=False, you might explicitly save your position after you’ve sent a confirmation email (to avoid marking events as consumed before actual business logic is complete).

Choosing which one:

* Use True for simplicity where "at least once" processing is fine and order isn’t critical.
* Use False for high reliability/transactional cases, when you must ensure something (DB write, notification) happened before committing progress.

**Qn 2 What is a Lock in Python?**

* A **lock** (from threading.Lock()) is a synchronization primitive that ensures that only one thread can access a section of code (or data) at a time.
* It prevents multiple threads from modifying the same **shared resource** simultaneously, which could cause data corruption or inconsistencies (called a "race condition").

**In Your Program**

* **Shared resource:** Your latest\_records dictionary, where you store the latest record per mtn.
* **Problem:** All 3 threads read and may try to write to the latest\_records at the same time.
* **Solution:** Use a lock to protect the critical section—the part where you update latest\_records.

python

lock = threading.Lock()

*# Inside your thread:*

**with** lock:

*# Only one thread can do this at a time!*

**if** (mtn **not** **in** latest\_records **or** timestamp > latest\_records[mtn]['timestamp']):

latest\_records[mtn] = record

**How It Works (Step by Step)**

1. **Thread-1** (say, Consumer for Partition 0) gets a new message to process.
   * It reaches the with lock: line and, if no one else holds the lock, acquires the lock and enters the block.
2. **Thread-2** arrives at the same time, but **must wait** because Thread-1 is inside the with lock: block.
3. **Thread-1** safely checks and potentially updates latest\_records, then exits the with lock: block, **releasing the lock**.
4. **Thread-2** now acquires the lock, and safely performs its update/check.

This ensures **no two threads ever step on each other's changes**, so your dictionary always holds the true "latest" message for each mtn, no matter how rapidly updates come in.

**Why is this important?**

* **Without a lock:**  
  Multiple threads could try to update the dictionary at once; some writes could be lost, or data could be corrupted.
* **With a lock:**  
  Updates are serialized—only one thread writes at a time, ensuring correctness and thread safety.

**Note:**

.\bin\windows\kafka-consumer-groups.bat --bootstrap-server localhost:9092 ^

--group parallel\_group ^

--topic test ^

--reset-offsets --to-earliest –execute

**To reset the offset.(start reading messages from the very first)**

**Important:**

To remove all messages from all partitions in the topic test (effectively "empty" it), without deleting the topic itself, you should use topic truncation or change the retention policy so all records are deleted.

Option 1: Set Retention to Zero (Most Common)

1. Run this command to set retention.ms for the topic to zero:

text

kafka-configs.bat --bootstrap-server localhost:9092 --entity-type topics --entity-name test --alter --add-config retention.ms=0

* + This will cause Kafka to immediately delete all messages in the topic.

1. After a minute (once deletion is complete), set retention back to your normal value (e.g., 7 days):

text

kafka-configs.bat --bootstrap-server localhost:9092 --entity-type topics --entity-name test --alter --add-config retention.ms=604800000

(Where 604800000 is 7 days in milliseconds)

This works for all partitions and does NOT remove the topic itself. Only the data is deleted.

Option 2: Use Delete Records Command (Advanced, Selective)

You can use the kafka-delete-records.sh tool to delete all records up to the latest offset for each partition.

* First, use kafka-run-class.bat kafka.tools.GetOffsetShell --broker-list localhost:9092 --topic test to get the offset for each partition.
* Then, prepare a delete.json file like this:

json

{

"partitions": [

{"topic": "test", "partition": 0, "offset": <latest\_offset0>},

{"topic": "test", "partition": 1, "offset": <latest\_offset1>},

{"topic": "test", "partition": 2, "offset": <latest\_offset2>}

]

}

* Run:

text

kafka-delete-records.bat --bootstrap-server localhost:9092 --offset-json-file delete.json

This will delete data in all partitions up to the specified offsets.

Option 3: (Quickest) Just Set retention.ms to 1 for a minute

This will purge all data (set as 1ms instead of 0 if needed):

text

kafka-configs.bat --bootstrap-server localhost:9092 --entity-type topics --entity-name test --alter --add-config retention.ms=1

Wait a minute, then reset retention to normal value.

Summary:

* No data loss of topic structure or configs—just the data is removed.
* After this, only new messages will be present in the topic.

When deletion topic and trying to restart the kafka -server and face issue with some exceptions

Do NOT delete \_consumer\_offset and consumer\_metadata folders.

* These folders are critical system topics for Kafka:
  + \_consumer\_offsets stores all consumer group offsets (tracking where consumers are in every topic/partition).
  + consumer\_metadata is used for internal group management and metadata.

Deleting these folders will cause Kafka to lose consumer group history and may introduce serious errors in group coordination and offset tracking!

What SHOULD you delete?

* ONLY remove folders that correspond to your deleted user topic (for example, folders like test-0, test-1, test-2, or with 'stray' in the name related to your 'test' topic).

**For parallel working concept.**

1. **Using Multithreading(separate process)**

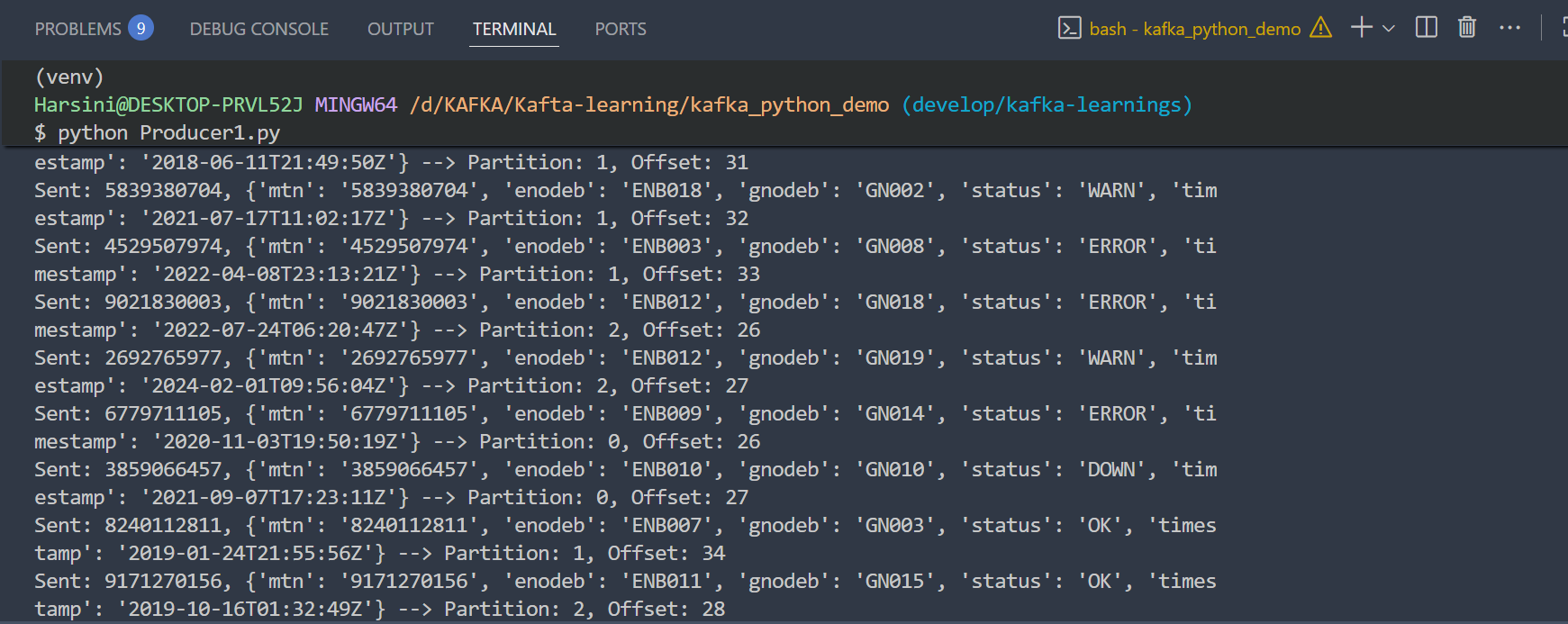
Created a new topic – test1.

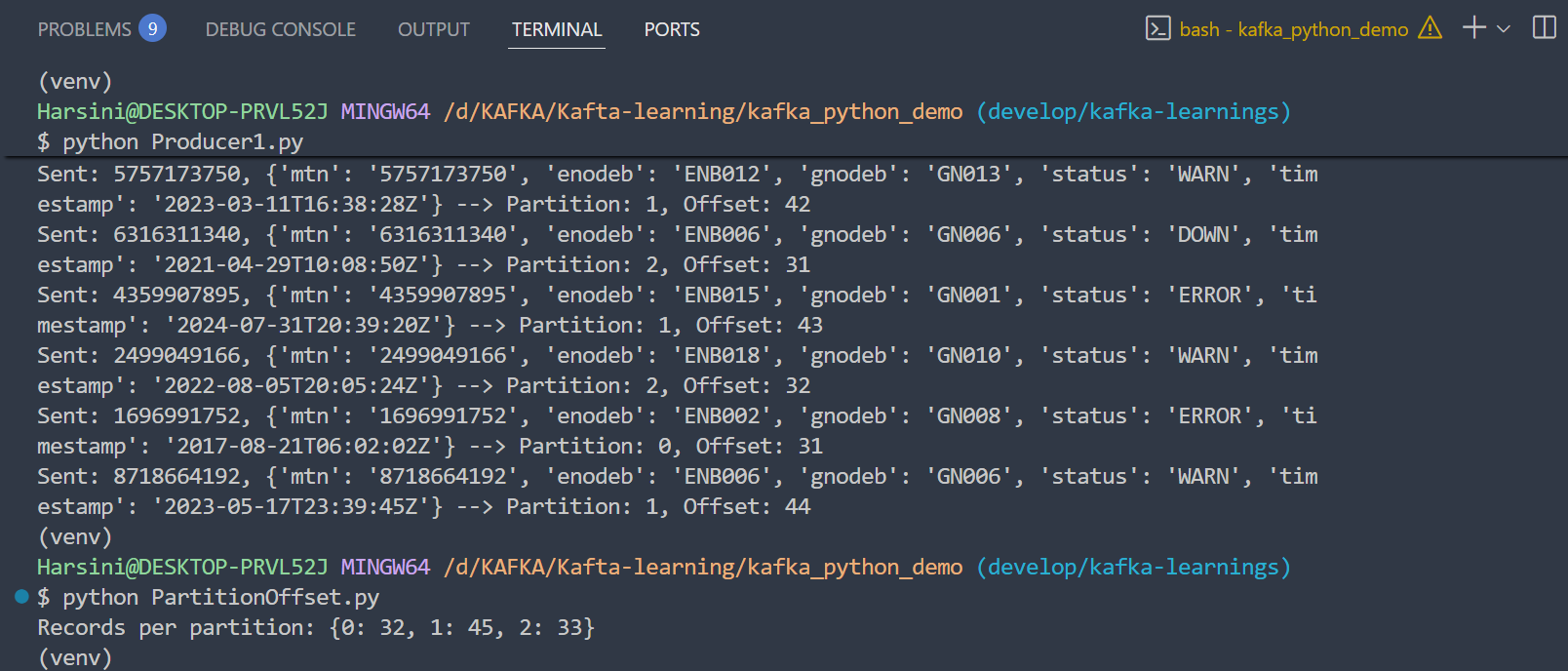
Inserted data from Producer1.py (contains 110 records with 10 duplicates)

ParallelProcessConsumer.py -> used multithreading and consumer parallel (3 consumers from the same group\_id but with different client\_id)

**Prodcuer1.py code flow:**

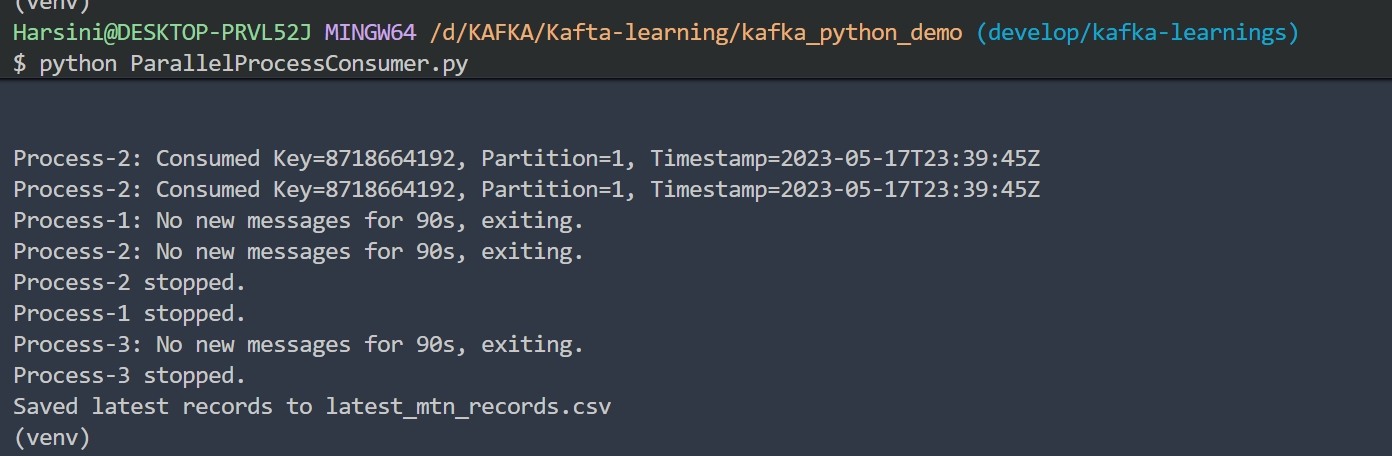
1. Establishes connection with kafka broker on that topic,
2. Reads the mock\_kafka\_data.csv file and inserts into the topic test1 with 3 partitions
3. Key is choses as the MTN. so a new message with same MTN will be send to the same partition.
4. Finally finishes the execution





**ParallelProcessConsumer.py flow:**

1. **Launches 3 parallel consumer processes** to consume records from the Kafka topic (test1) as part of the same consumer group.
2. **Each process polls for messages on its assigned partitions**, reading JSON records and extracting mtn and timestamp.
3. **For each consumed record**, the process updates a shared dictionary (latest\_records) to keep only the record with the latest timestamp for each mtn.
4. **Consumers exit after 90 seconds of inactivity**, and all processes join (finish).
5. **After all consumers finish**, the script writes the latest record for each unique mtn into latest\_mtn\_records.csv.



1. **Using threads**

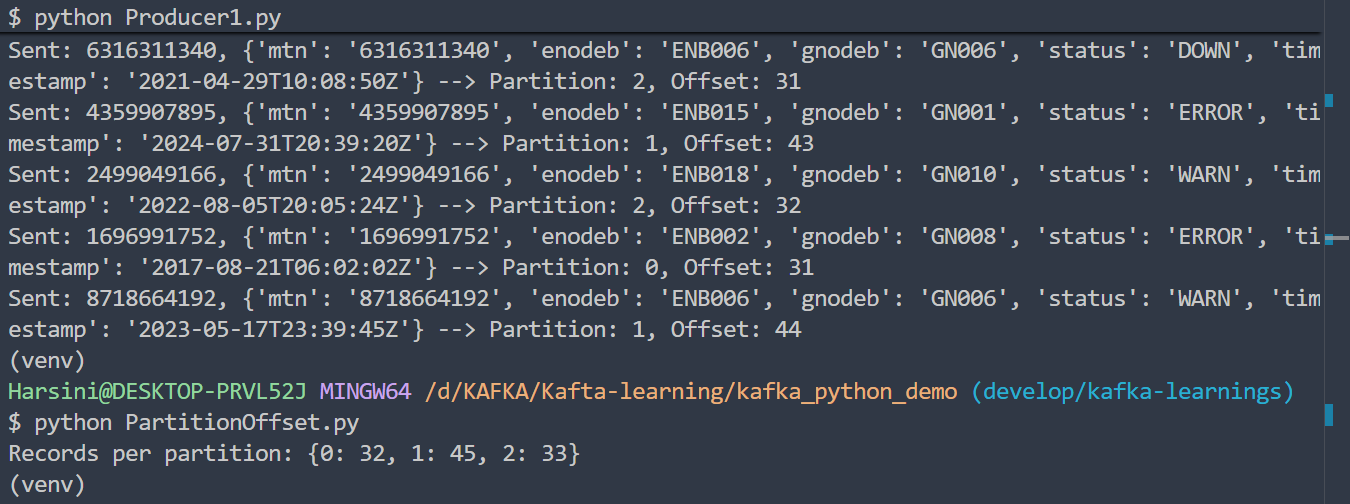
Created a new topic – test2.

Inserted data from Producer1.py (contains 110 records with 10 duplicates)

ParallelConsumer.py -> used multithreading and consumer parallel (3 consumers from the same group\_id but with different client\_id)

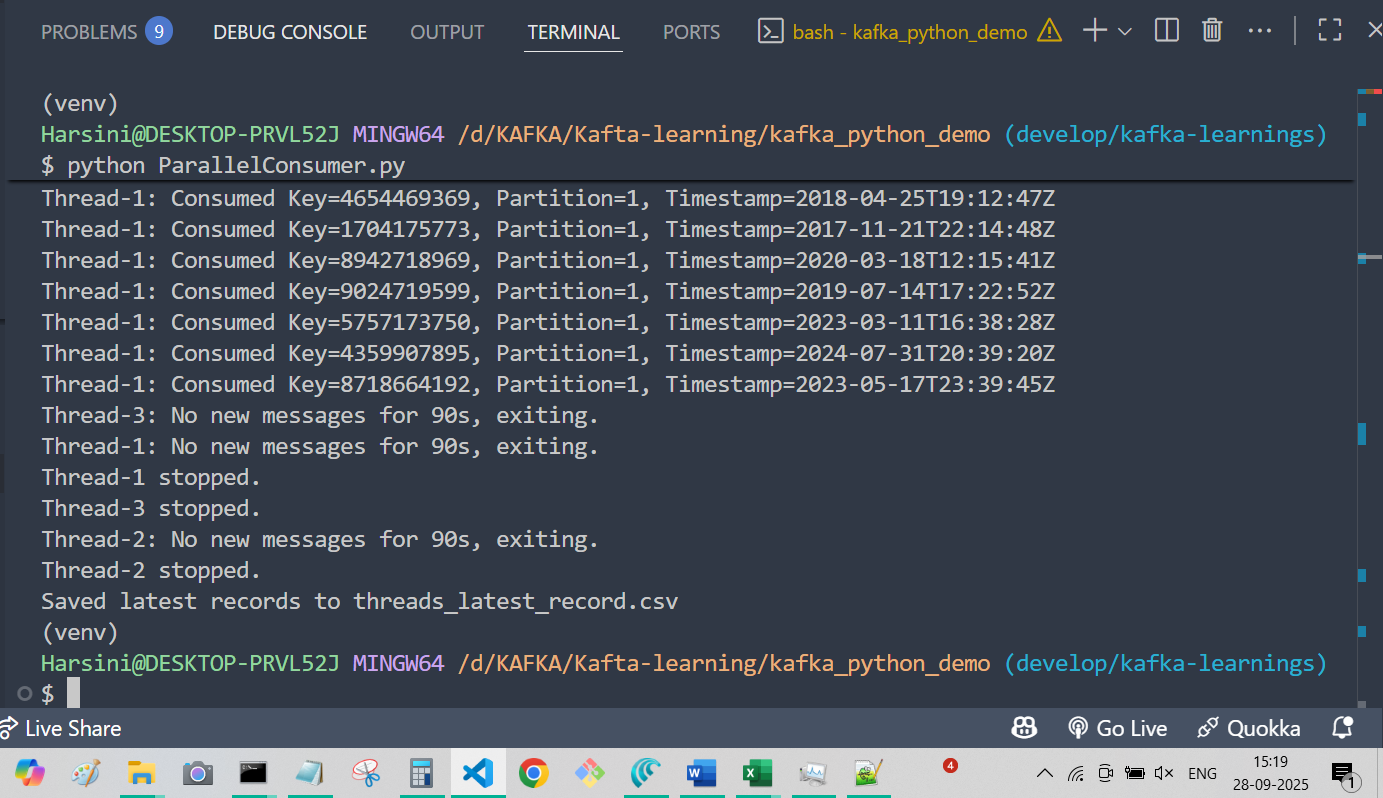
**Prodcuer1.py code flow:**

1. Establishes connection with kafka broker on that topic,
2. Reads the mock\_kafka\_data.csv file and inserts into the topic test1 with 3 partitions
3. Key is choses as the MTN. so a new message with same MTN will be send to the same partition.
4. Finally finishes the execution



**ParallelConsumer.py flow:**

1. The program launches 3 threads, each starting a Kafka consumer to read from topic test2 in consumer group parallel\_group1.
2. Each consumer polls Kafka for messages and extracts the mtn and timestamp from each record, updating a shared dictionary with only the newest timestamp per mtn.
3. A threading lock ensures only one thread can write to the shared dictionary at a time to avoid race conditions.
4. Each thread runs for up to 90 seconds while receiving messages, then exits after a period of inactivity.
5. After all threads finish, the latest records for each mtn are written to threads\_latest\_record.csv.



Here’s the **key difference between using 3 consumers as threads vs. processes** with Kafka Python:

| **Aspect** | **Threads** | **Processes** |
| --- | --- | --- |
| **Python GIL** | All threads share the Global Interpreter Lock (GIL); only one thread runs Python code at a time per process. | Each process has its own Python interpreter and memory space, so all run in true parallel. |
| **Kafka Consumer Group** | **Kafka detects threads as one client/app:** All threads from one process are seen by Kafka as a single consumer, so only one thread will actually pull from the group. | **Each process is a separate group member:** Kafka will assign partitions to all (e.g., with 3 consumers and 3 partitions, all handle work in parallel). |
| **Partition Assignment** | Only one thread in the process will receive partition assignments; others are idle. | Each process can be assigned different partitions for true parallel consumption. |
| **Resource Isolation** | Threads share memory and state; easy to communicate but not truly concurrent in CPU-bound tasks due to GIL. | Processes have separate RAM; must use inter-process communication, but scale and performance are higher. |
| **Best Practice for Kafka** | Threads are ineffective for parallel consumption; use them only for other non-Kafka logic. | **Processes are the recommended method for parallel Kafka partition consumption in Python.** |

**Doubts**

**Your understanding is spot on about the locking and the difference between threads and processes in Python for this use case!**

Details:

1. Why use a lock for threads?

* With threads, all share the same memory and dictionary (latest\_records).
* Python's threading is not truly parallel (because of the Global Interpreter Lock, GIL), but I/O (network) can overlap, and context switches can happen at any time.
* If two threads modify the dictionary simultaneously, you can get race conditions (data corruption or missing updates).
* The lock ensures only one thread writes to the dictionary at a time, protecting against such issues.

2. Why not use a lock for multiprocessing?

* With processes (your earlier example), each process has its own address space and memory.
* When you want to share a resource (e.g., dictionary) across processes, Python’s multiprocessing.Manager().dict() handles the necessary locking and synchronization behind the scenes.
* No explicit lock is generally needed when using a manager dict, as it's safe for cross-process updates (protected by the multiprocessing library itself).

3. Real outcome for Kafka with threads

* Only one thread will actually consume messages from Kafka (because the KafkaConsumer in Python isn't designed for true multithreaded parallelism). The lock will work, but you won't get real parallel partition assignment.
* You’ll see that most/all messages are processed by one thread, the others mostly idle.

Summary Table

| Mechanism | Lock Needed? | Why? | True Parallel Kafka Consumption? |
| --- | --- | --- | --- |
| Threads | Yes | Shared memory, possible race conditions | No (1 thread consumes, others idle) |
| Processes | No\* | Separate memory/process space; Manager handles locking | Yes (each process consumes) |

\*Except for rare advanced use cases—then, use multiprocessing.Manager().Lock().

In conclusion:

* Locking is required for thread safety, not process safety (when using Manager dict).
* But with threads and Python’s Kafka client, you get parallel safety, not parallel consumption!

Let me know if you want a clear mini-demo or code that visually proves thread/process partition assignments!Your analysis is correct:

* With threads:
  + All threads share the same latest\_records dictionary (because they are in the same memory space).
  + You must use a lock to prevent race conditions—otherwise, two threads might try to update the dictionary at the same time, causing data corruption or missing updates.
* With processes (multiprocessing):
  + Each process is in a separate memory space, so normal variables are not shared.
  + When you use a multiprocessing.Manager().dict(), it automatically handles process-safe access to the dictionary, so you don't manually acquire a lock.

However, as explained earlier, with Python's threading and Kafka:

* Only one thread will actually consume from the Kafka topic, because Python's KafkaConsumer is not thread-safe for multiple parallel consumers in the same process. The others will usually be idle. The lock keeps updates safe, but it does not give you real parallel consumption from multiple partitions.

Summary:

* Use a lock for threads (shared memory, possible race conditions).
* With processes, use Manager dicts—no explicit lock needed.
* For real Kafka partition parallelism, prefer multiprocessing over threading in Python.

Note:  
**group.id**

* **Purpose:** Identifies a group of consumers coordinating to share the workload of one or more topics.
* **Kafka Role:** Consumers in the same group.id get assigned topic partitions (no duplicate work; parallel scaling).
* **Behavior:** Kafka tracks committed offsets per (group.id, topic, partition).

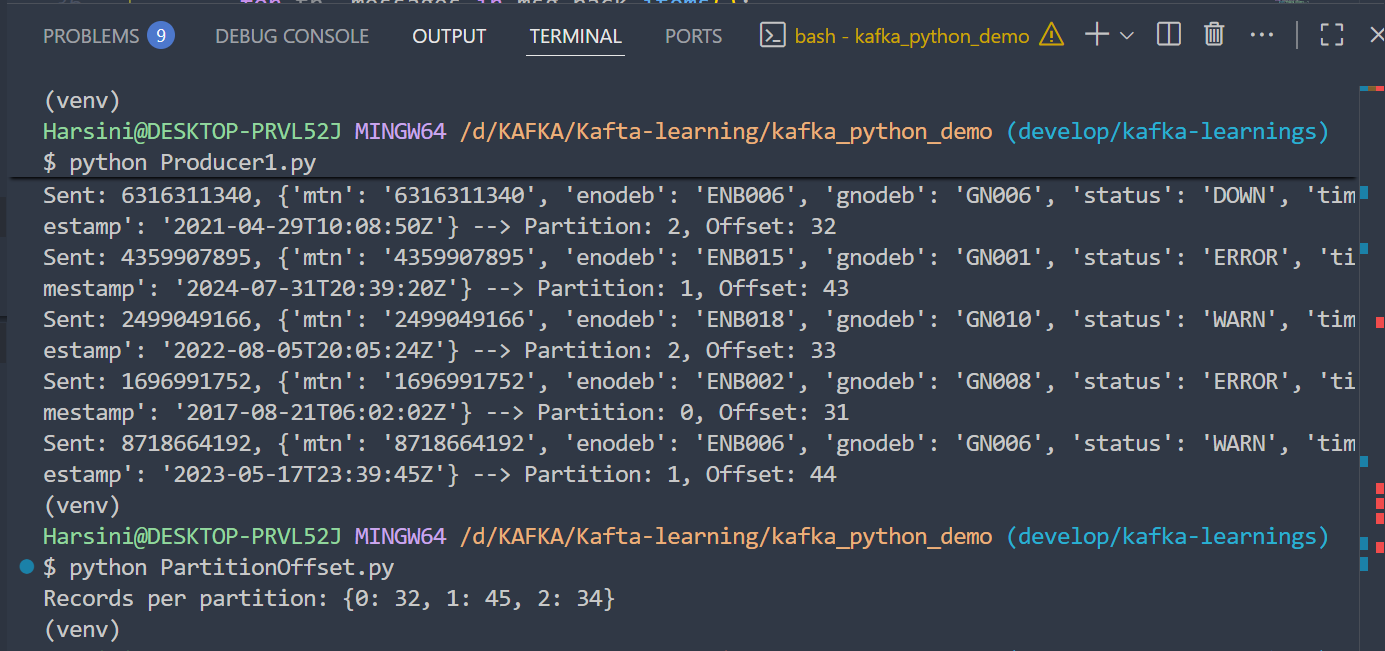
**client.id**

* **Purpose:** Identifies the individual Kafka client instance (for logging, tracking, metrics).
* **Kafka Role:** Used in broker logs and metrics, not for partition assignment or offset tracking.
* **Behavior:** Has no effect on how messages are delivered or processed, but helps you monitor/follow client actions.

Multiple consumer-groups for each consumer (unique group-id)

1. **Producer1.py**

* Created a new topic – test3.
* Inserted data from Producer1.py (contains 110 records with 10 duplicates)
* ParallelConsumer.py -> used multithreading and consumer parallel (3 consumers from the different group\_id and different client\_id)



1. **DifferentConsumerGroup.py**

* The program launches 3 threads, each running a Kafka consumer subscribed to topic test3.
* Each consumer is assigned a **unique, dynamically generated consumer group ID** using its thread number and a timestamp, so every thread acts independently.
* The consumers poll for messages and, for each unique mtn key, update a shared dictionary only if the new record has a more recent timestamp (using a lock for thread safety).
* Each thread tracks and prints the count of messages it reads and logs details about each message (including partition and timestamp).
* If a thread receives no messages for 90 seconds, it prints how many messages it read and exits.
* After all threads finish, the latest per-mtn records are written to diff\_consumer\_latest\_record.csv.
* **Result:** Since each thread has a separate consumer group, **every thread will independently process all records in the topic**, i.e., each will print and count all messages, not just a subset.

