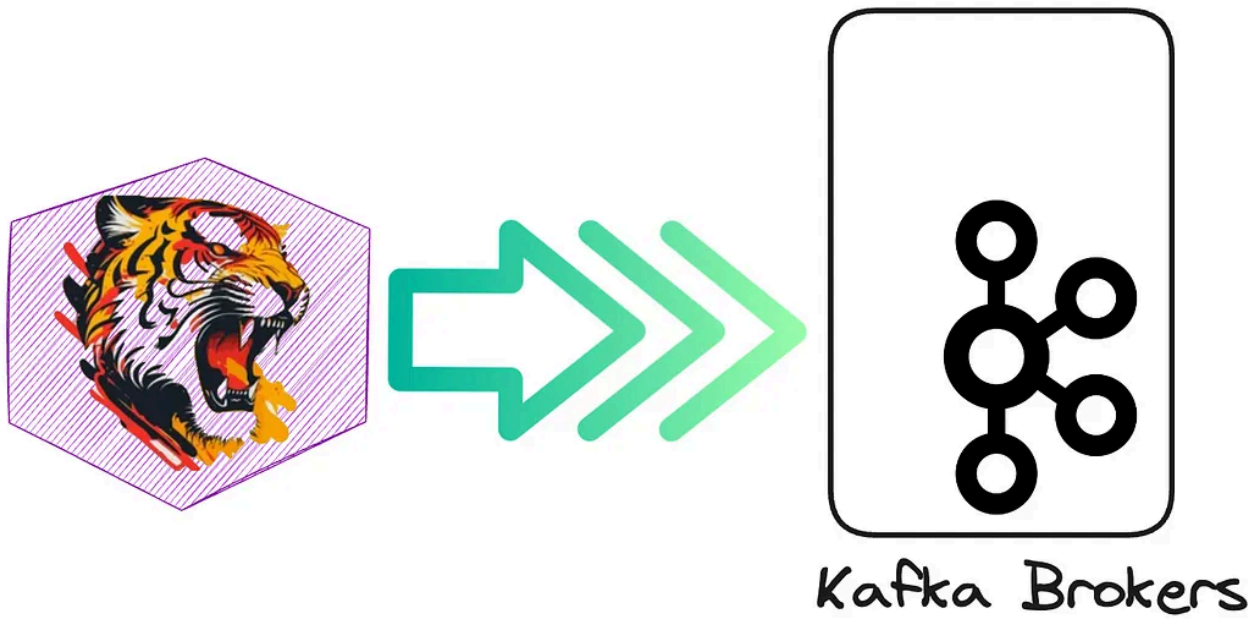
Member-only story

Kafka Producer Deep Dive

Jorge Saenz · Follow
8 min read · Apr 26, 2024

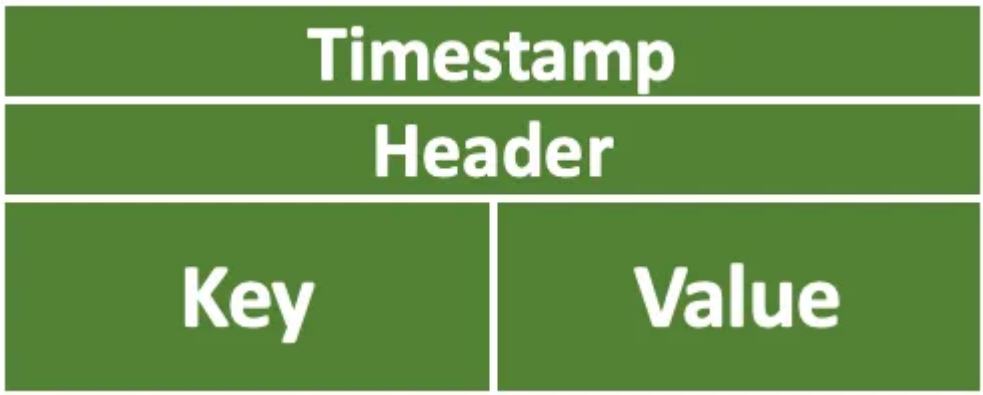


Kafka Producer Dep Dive

> *TL;DR*
>
> *We will delve into the operations of the Kafka producer, covering concepts such as Record, metadata, serializers, partitioners, and ordering guarantees in production. We will also discuss the importance of idempotence, as well as the usage and recommendations of headers — Cloud events.*

Si lo prefieres en castellano: Profundizando en el Productor de Kafka

The Apache Kafka ecosystem is revolutionizing how applications manage and process large-scale data streams. Within this robust ecosystem, the producer emerges as a key component, as it is responsible for initiating the process. A good producer can accelerate the entire ingestion process by providing quality and guarantees from the outset, thereby avoiding costly debugging phases. In this article, we will delve into the depths of the Kafka producer, exploring everything from the fundamentals of events to recommendations and advanced optimization techniques.



Inside a Kafka Record

Record

Although commonly referred to as events or messages in the Kafka context, in technical terms they are called Records. These Records contain several important elements:

As we can see in the image, they include:

Timestamp

Indicates the date or moment when the event was generated. There are three configuration options: when it is sent, when it is received, and custom.

By default, the Kafka library will set the value at the moment the event is sent to the Broker, known as ***TimestampType = CreateTime***. It is also possible to configure it so that the Broker applies the timestamp upon receiving and storing it, known as ***TimestampType = LogAppendTime***. Depending on your scenario, one configuration or the other may be interesting to use. Remember that by default, these metadata are used by other libraries (streams) in their decision-making processes (windows).

Lastly, custom. This means you can set the value that suits your functional needs.

Header

Headers provide additional metadata that can be useful for consumer applications. Accessing these headers in producer code allows for greater customization and control over the data flow.

In Kafka, libraries typically allow you to access headers as Strings, but they actually work in Base64 (this avoids the need to escape special characters).

It's a place where we can include any key-value header we want. It's the ideal place to introduce Metadata for your message. By including them in the Headers, we avoid including specific Kafka technical information in your Payload, which is ideal for contract and data governance since we could have "identical" contracts in any type of integration (API, file, etc.). We'll talk more about Metadata later.

In Headers, we can also include functional or traceability fields. For high-volume scenarios and/or data with a high weight, we can provide information that allows for filtering or classification. For example, being able to filter only the data that is for my process. This way, you can consume the message in binary format and perform filtering or validation with the Headers, and only if it's of interest, proceed to deserialize and consume. Remember that the deserialization and mapping process is very fast, so you should only apply this practice in processes with many millions of messages.

Finally, remember to follow a standard or naming convention to avoid confusion for consumers. For example, for technical fields of the framework, we can create headers like: *fwk_version, fwk_class*. For application fields, we can use: *app_date, app_country, etc.*

Key

Information about the event we send. If we use the default partitioner, it's used to distribute the load among the partitions of the topic and to guarantee order.

Remember that if you're using a compacted topic, it's this value that triggers compaction. Finally, in Kafka Streams join processes, this value is what triggers the actions.

Hence, the key serves a more technical rather than functional purpose. Therefore, it's crucial to understand that altering the key contract or its configuration could

compromise the assurance of order and compaction. In other words, if you intend to modify the key, I suggest implementing it in a different version of the topic, rather than the same one.

Value

This is the simplest field, as it contains the data we want to send. I recommend that, both in Kafka and in any other integration system, you always use contracts to exchange data.

Metadata

Metadata refers to data that describes and provides information about other data. In the context of Kafka producers, we can use them to describe the content of the message we send. We can add information such as a unique identifier, generation time, used class, event reason, or any technical information that adds value to identifying and tracing our messages.

I recommend adopting "cloud_events," an open standard for cloud event interoperability, to enhance portability and integration between systems. Although not fully deployed, it offers a quick view of the data that will provide value. For example:

```
ce_specversion: "1.0"
ce_type: "com.example.someevent"
ce_source: "/mycontext/subcontext"
ce_id: "1234-1234-1234"
ce_time: "2018-04-05T03:56:24Z"
content-type: application/avro
```

Serializers and Partitioners

When sending data to the Kafka cluster, it's essential that the data is in the correct format. This is where serializers come into play, converting *Java objects* into bytes that can be transmitted across the network. Additionally, partitioners determine how records are distributed among the topic partitions, directly impacting the system's performance and scalability.

It's highly recommended to use serializers associated with schemas. Let's analyze the ones we have native support for:

Avro:

- **Schemas**: Avro utilizes a schema defined in JSON for data serialization. These schemas are explicit, allowing for schema evolution control with compatibility if you have a Schema Registry.
- **Data Size**: Avro "compresses" the data it sends.
- **Performance**: Excellent, with compact message size and fast serialization.

JSON:

- **Schemas**: JSON doesn't have an explicit associated schema. Data is serialized in a human-readable plain text format. Therefore, we cannot utilize schema evolution control functionality; however, we can include structured data validations, such as constraints, date formats, maximum lengths, minimum/maximum values, etc.
- **Data Size**: JSON tends to be less efficient as it describes all the data.
- **Performance**: Less efficient compared to binary formats like Avro or Protobuf.

Protobuf:

- **Schemas**: Protobuf uses a binary schema defined via message definition files (.proto). These schemas are less human-readable but more compact and efficient.
- **Data Size**: Protobuf produces highly compact data due to its binary format and lack of redundancy in serialized data.
- **Performance**: Very fast with small message size.

In summary, **Avro** is known for its schema evolution capability, JSON is widely compatible and easy to read, while Protobuf offers optimal efficiency and performance. The choice depends on the specific needs of the application, performance requirements, and interoperability.

Personally, I recommend using JSON for low-volume scenarios, Avro for medium to high volume and critical integrations where schema evolution control is important. Protobuf would only be used if there is already high adoption within the company.

Partitioners

As for partitioners, there are 3 types or strategies:

1. **DefaultPartitioner**: This is the default partitioner in Kafka and is used if no partitioner is explicitly specified. This partitioner uses the event key, applies a hash function, and determines the partition where the message will be stored.
2. **RoundRobinPartitioner**: This partitioner uses a Round Robin approach to sequentially assign events to available partitions in a topic. This means that each event is assigned to a different partition in sequential order, starting from the first available partition in the partition list.
3. **CustomPartitioner**: It is possible to create a custom partitioner that fits the specific needs of an application or use case. To do this, it is necessary to implement a custom partitioner interface and provide the appropriate partition logic for that application.

Guaranteeing Order in Production and Idempotence

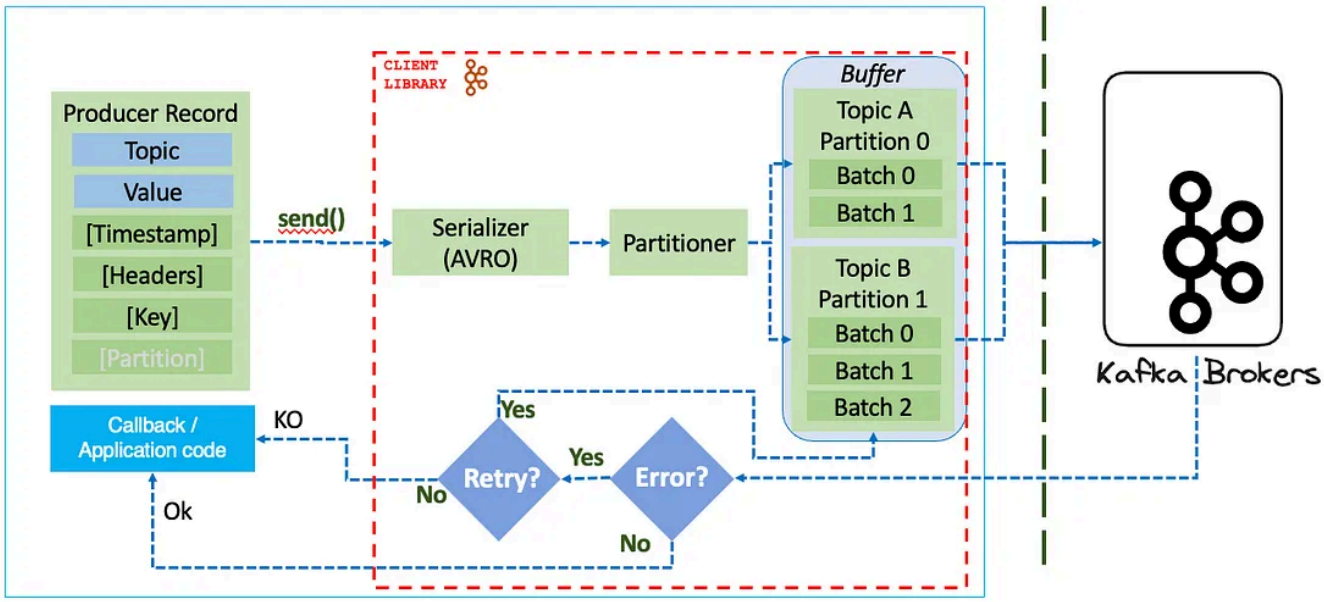
In many cases, maintaining the order of messages when they are sent to the Kafka cluster is crucial. To achieve this, the producer offers options to ensure the temporal coherence of records, which is essential in applications sensitive to order, such as transaction processing and tracking systems.

On the other hand, idempotence is a vital concept in distributed systems, and the Kafka producer offers support to ensure idempotent operations. By enabling the *enable.idempotence* parameter, the producer can safely retry sending records in case of failures, thus avoiding unwanted duplicates and ensuring order.

It is important to understand that the guarantee only applies within the Kafka system, but for further details on error handling, I recommend this Blog:Building Resilient Kafka Producers: Strategies and Best Practices.

Performance

Message sending performance can be improved using techniques such as batching and compression in the producer library. Kafka is designed to batching, but to better leverage these capabilities, we need to tune it.



Deep Dive into Kafka Producer

In the diagram, we can see that all messages sent to Kafka are first stored in small buffers we have in memory, which are always partition-specific. Depending on how we configure these buffers, we can achieve lower latency or better performance. This is achieved with the parameters:

- **batch.size**: Maximum number of bytes accumulated in a batch to send to the broker.
- **linger.ms**: Maximum time the producer waits before sending the batch to the broker.

By default, Kafka is tuned for better latency, meaning *linger.ms* is set to 0; therefore, it will send as soon as possible without waiting to utilize the batch advantages. If we increase this value to *50ms*, we can increase the amount of data sent each time and thus improve performance. **Why does this happen?**

Sending larger batches makes better use of network capabilities. More importantly, if we're using Avro/Protobuf or message compression, it's crucial because compression is done per batch to be sent, not per message. Therefore, the larger the batch, the better the compression and overall performance.

On the other hand, *batch.size* is the maximum size to accumulate. If you increase this value, it's important to also review the total value of your buffer (*buffer.memory*) to avoid overflow.

Message Compression

The Kafka library provides native compression capabilities, offering the following formats:

- **GZIP** and **ZStd** for a good compression ratio, although it naturally incurs a small CPU cost.
- **Snappy** and **LZ4** to reduce CPU cost, but with simpler or basic compression.

However, depending on the content of the records and the batch size, it's recommended to test all options and choose the best one.

For **consumers**, **compression is transparent** as the library retrieves specific metadata and decompresses during the deserialization process. Therefore, there's no need to worry about programming this part.

When to compress? If your platform is on-premise and your clients are close to the data, there's very little latency, so compression loses value. Conversely, if your platform is in the cloud and/or you pay for usage (typically for network and storage), it can lead to significant cost savings. In terms of performance, compression can indeed increase when there's latency with the broker.

In conclusion, proper configuration of the Kafka producer plays a crucial role in Kafka's architecture, facilitating efficient and reliable data ingestion. Understanding its complexities, from record structure to advanced optimization techniques, is essential for building robust and scalable systems. With the right tools and practices, we can fully leverage Kafka's potential across a wide range of business applications.

If you enjoyed reading, "*follow me*"; for questions or input, "*drop a comment*".

Kafka | Integration | Data Science | Programming | Software Development

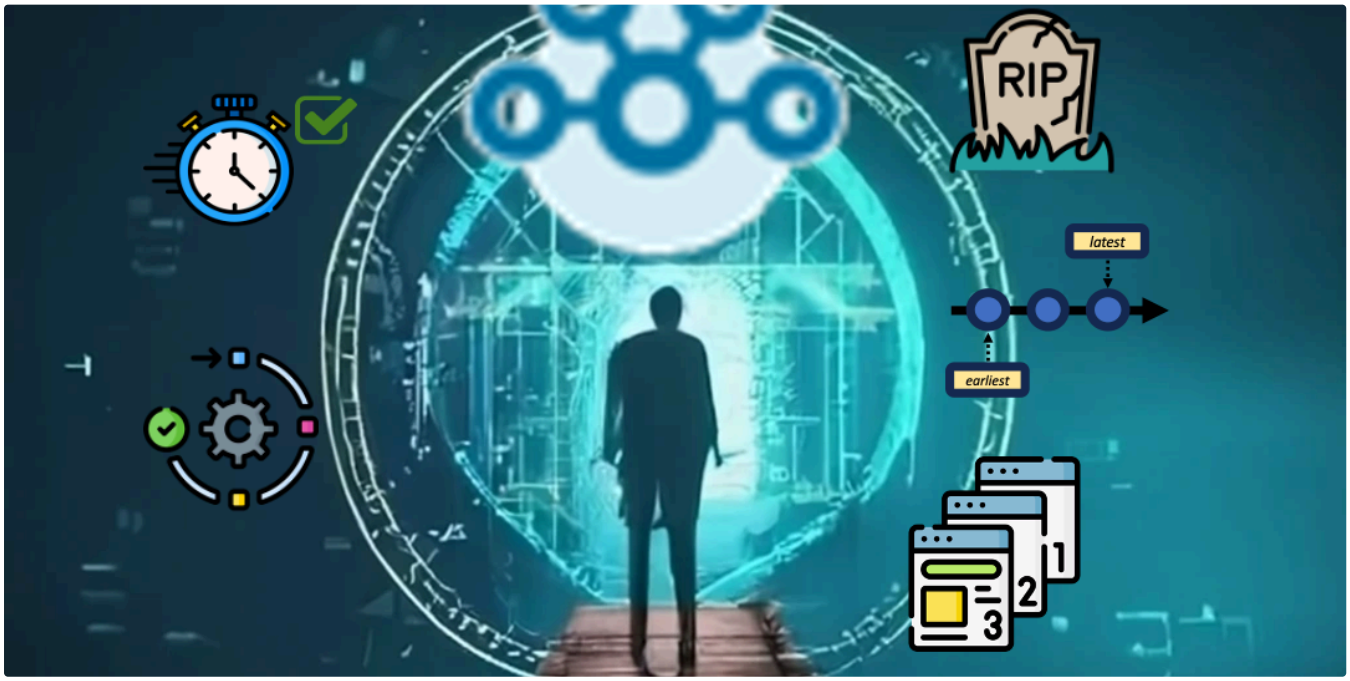
Follow

Written by Jorge Saenz

78 Followers

Event Broker PM | IT Integration & Orchestration Expert

More from Jorge Saenz

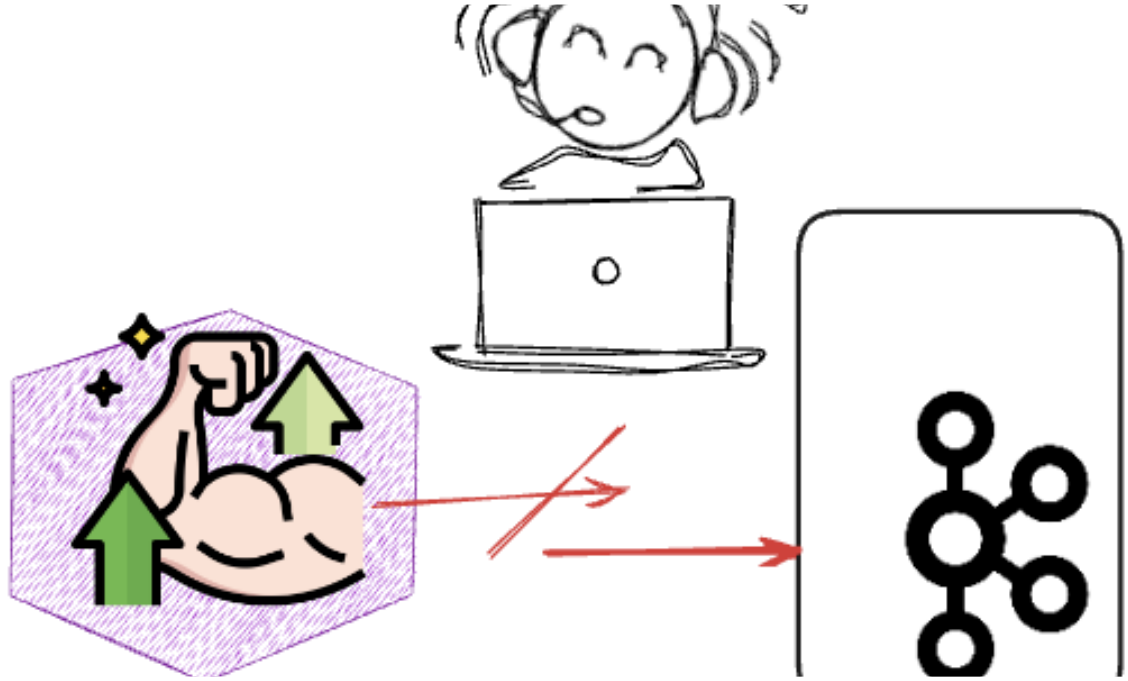


Jorge Saenz

Kafka Consumer Responsibilities

Kafka consumer requires precise handling of offsets, minimization of lag, schema updates, and tombstone management.

Jul 10 4

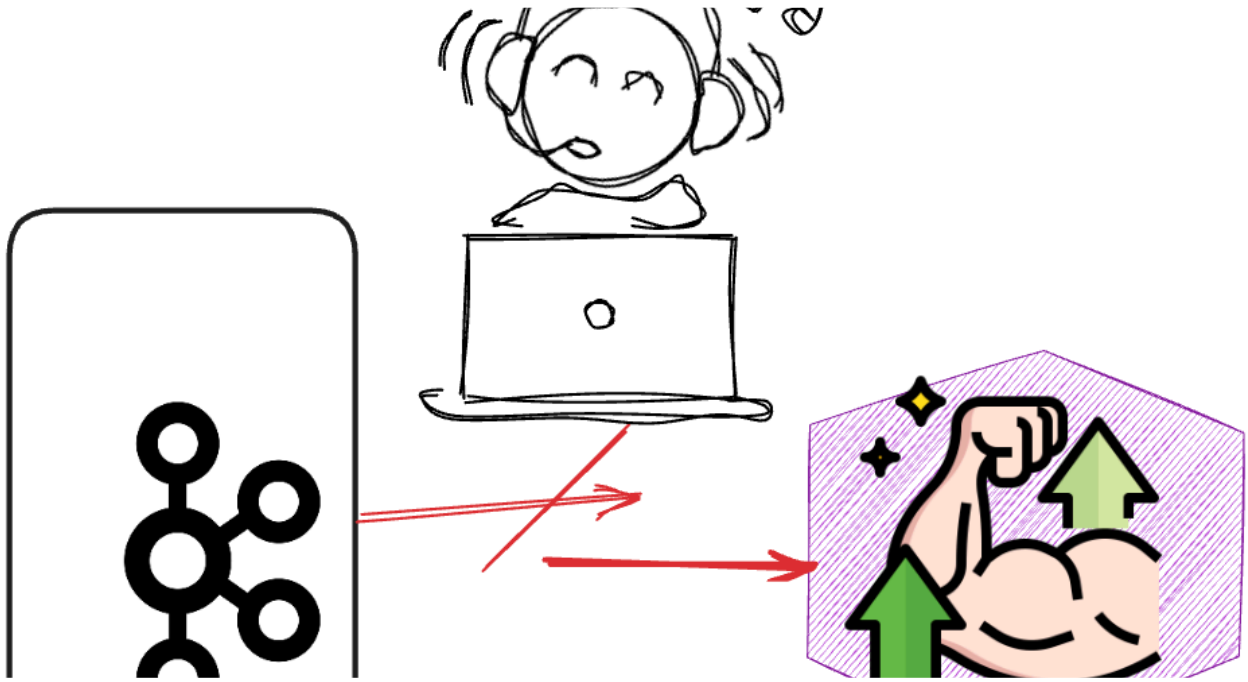


Jorge Saenz

Building Resilient Kafka Producers: Strategies and Best Practices

Errors in Kafka producers and how to manage them

Apr 15 51

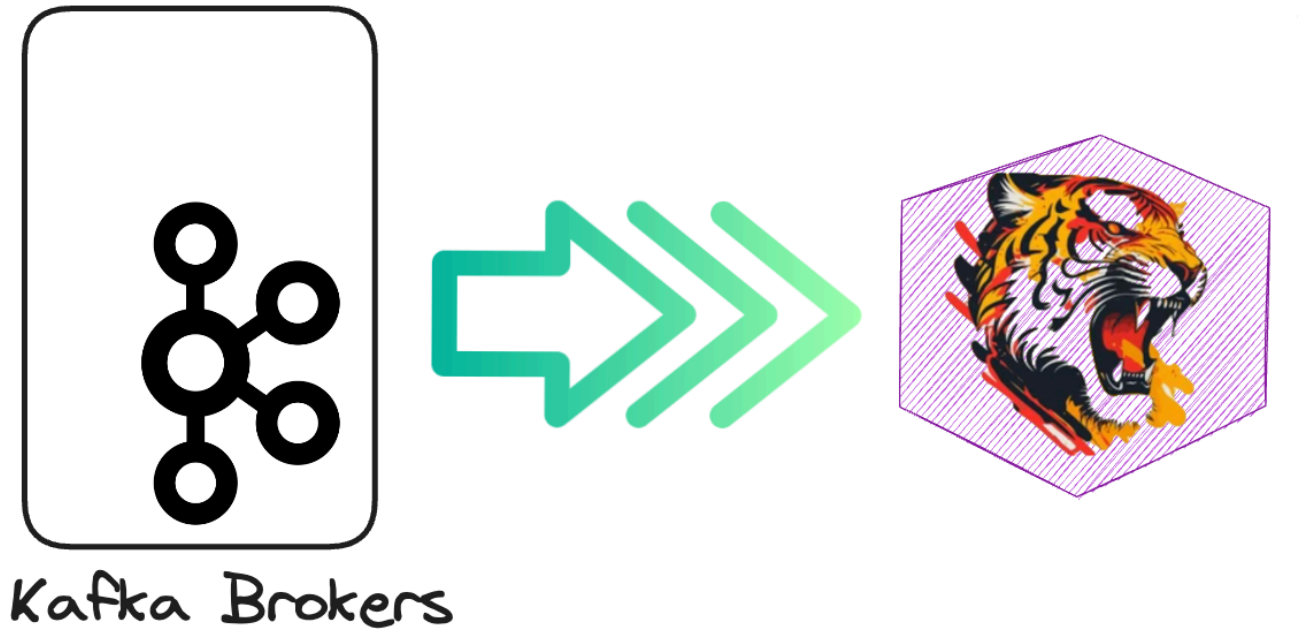


Jorge Saenz

Building Resilient Kafka Consumers: Strategies and Best Practices

Strategies and Best Practices for Increasing the Resilience of Your Kafka Consumers

May 18 95



Jorge Saenz

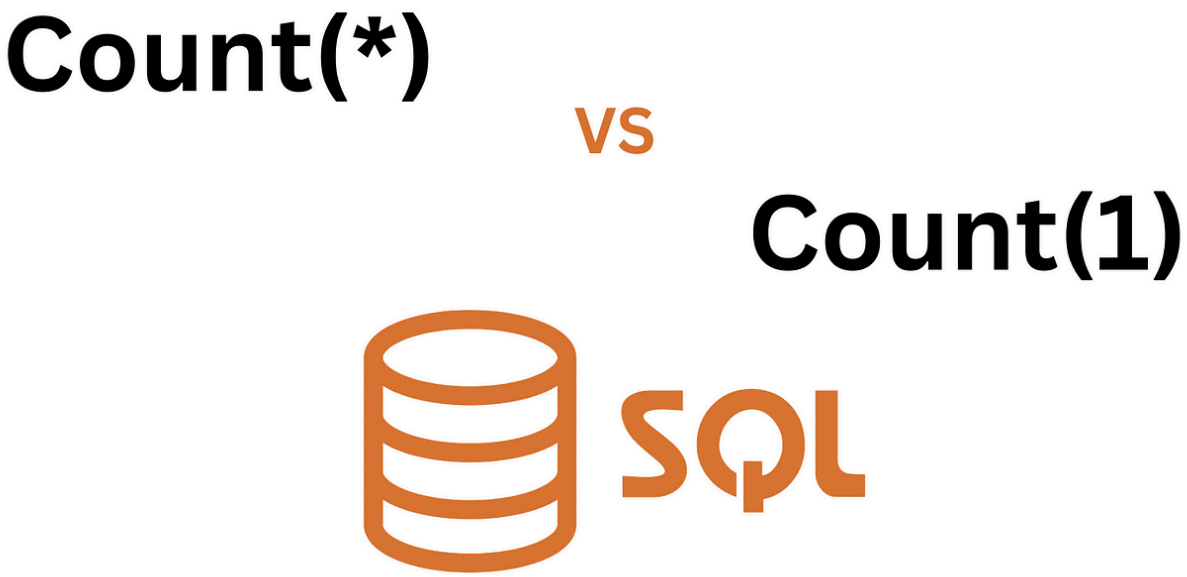
Kafka Consumer Deep Dive

Consumer Group, Leader & Coordinator, partition assignment, and consumer internals.

Jun 24 3

See all from Jorge Saenz

Recommended from Medium

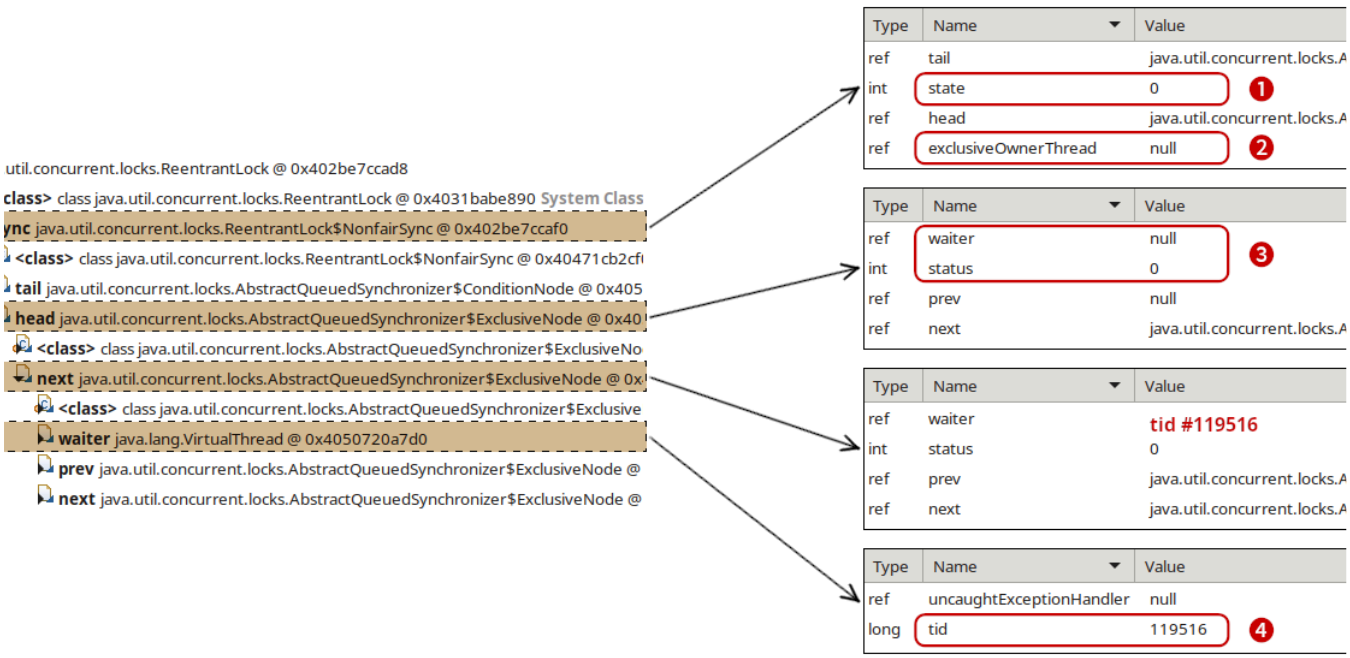


Vishal Barvaliya in Data Engineer

Count(*) vs Count(1) in SQL.

If you've spent any time writing SQL queries, you've probably seen both `COUNT(*)` and `COUNT(1)` used to count rows in a table. But what's...

Mar 9 722 15



Netflix Technology Blog in Netflix TechBlog

Java 21 Virtual Threads - Dude, Where's My Lock?

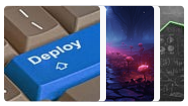
Getting real with virtual threads

6d ago 1.3K 12

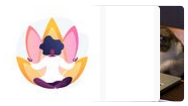
Lists

General Coding Knowledge
20 stories · 1448 saves

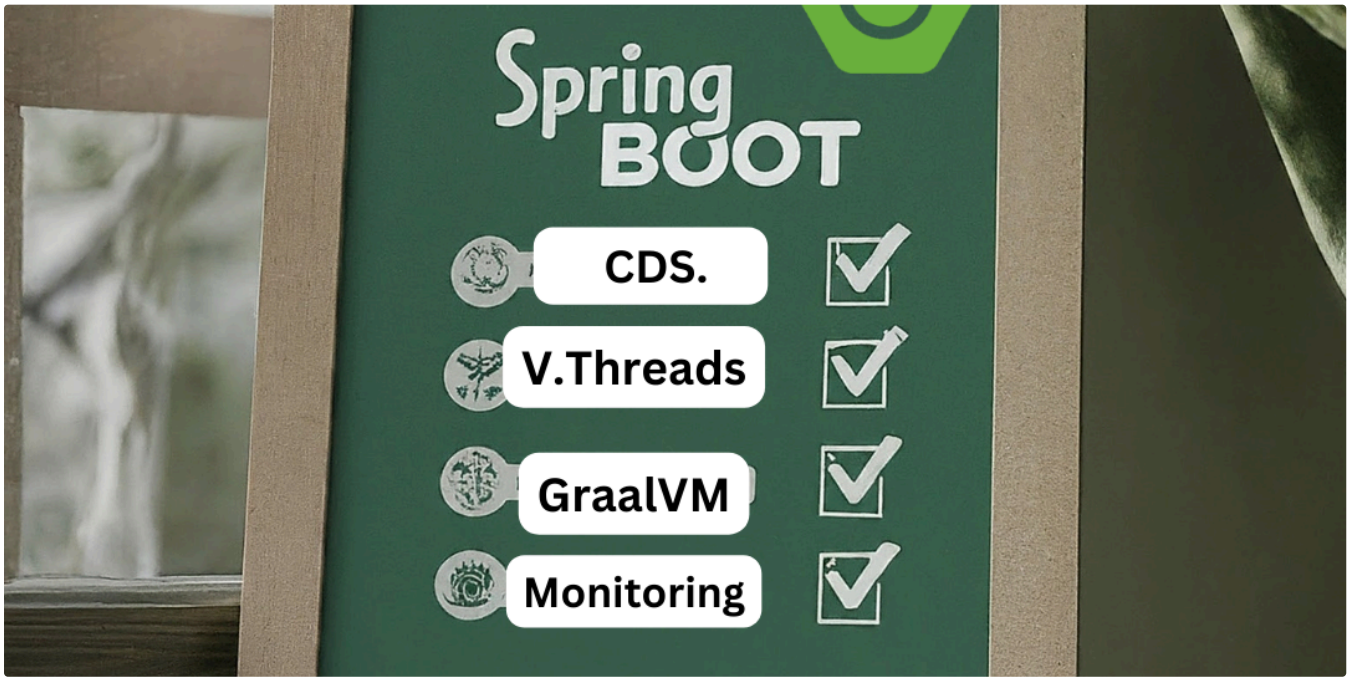
Coding & Development
11 stories · 726 saves



Predictive Modeling w/ Python
20 stories · 1423 saves



Stories to Help You Grow as a Software Developer
19 stories · 1248 saves



Amit Himani

Spring Boot 3 : Top 5 Features You Need to Know

Revving Up Your Apps: 5 Spring Boot Features That Supercharged Our Apps (and How You Can Too)

Jul 7 191 2

Mpavani

Know these before u attend a Kafka Interview| Kafka Consumer Group Interview Questions

1. What is a Kafka Consumer Group? —Consumer groups are a way to parallelize processing and load balance consumption of messages from...

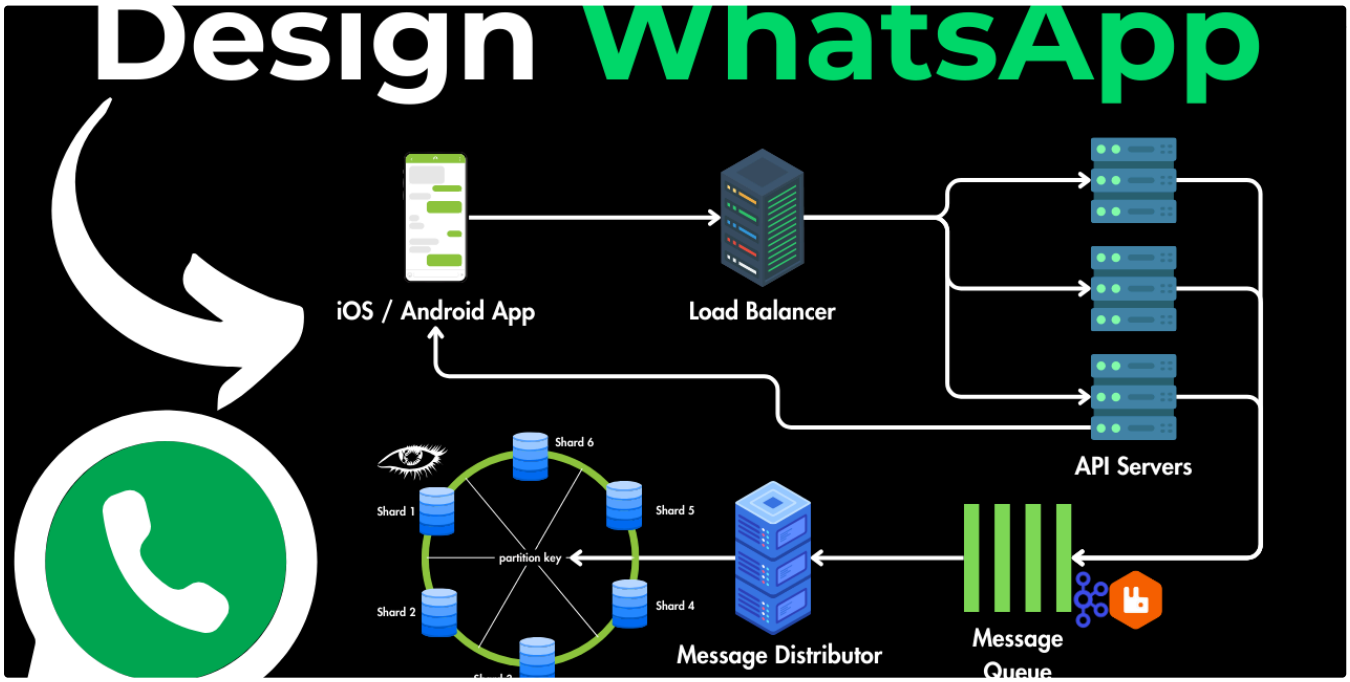
Feb 25 12

Matúš Seidl in Dev Genius

10 Java Concurrency Tricks Every Developer Should Know

Java concurrency and multithreading are essential skills for any developer aiming to build high-performance and responsive applications...

Jul 18 47 3



Hayk Simonyan in Level Up Coding

System Design Interview: Design WhatsApp

Prepare for system design interviews with this guide to designing a WhatsApp-like messaging app.

Jun 18 2.7K 22