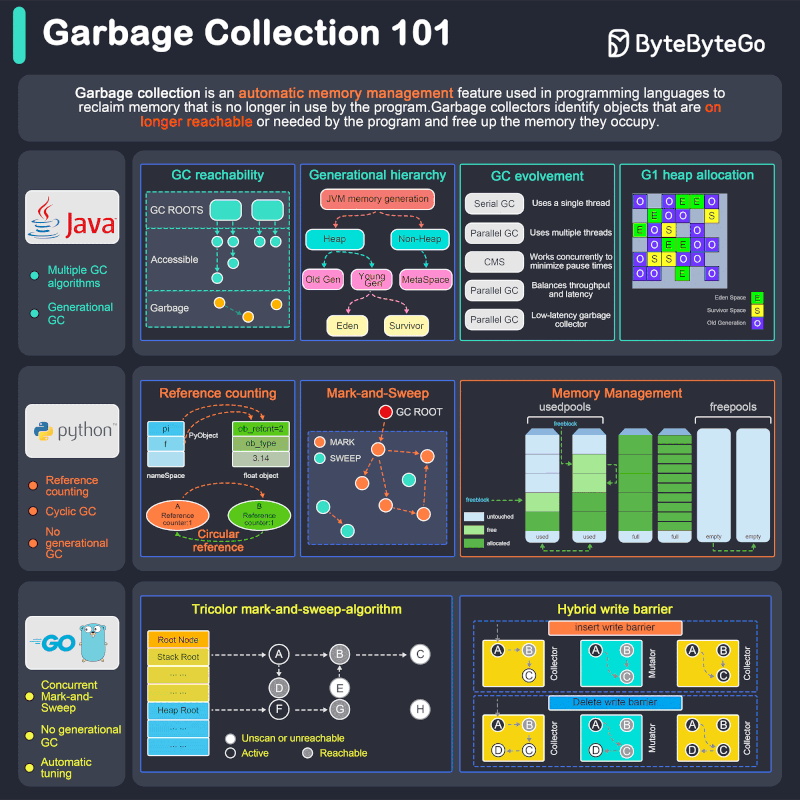
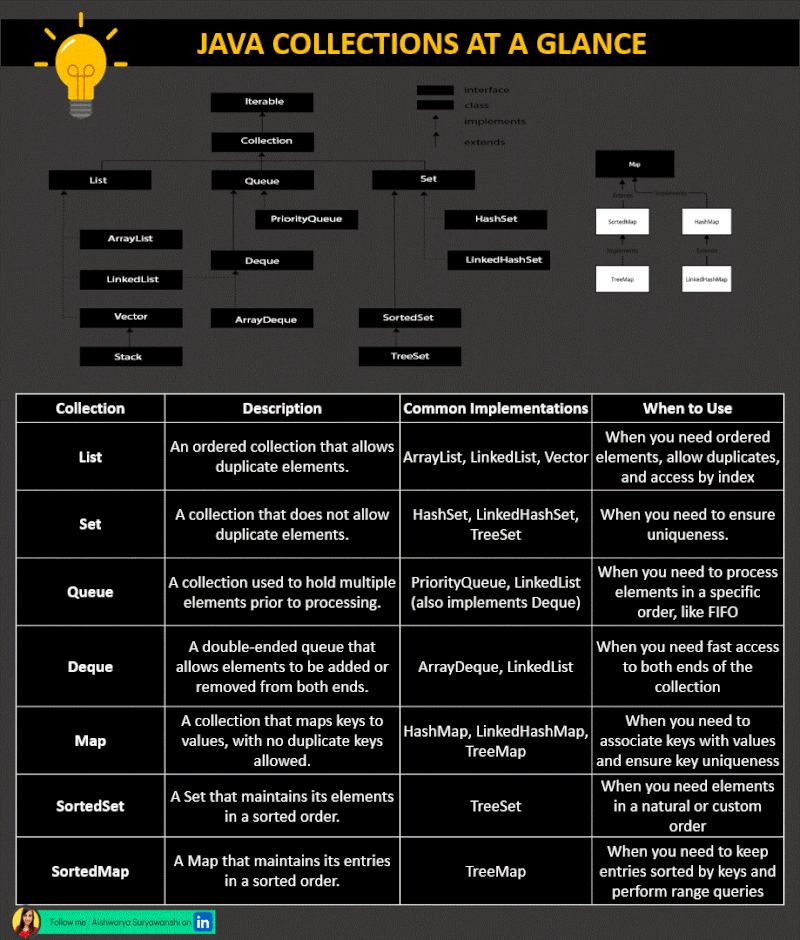
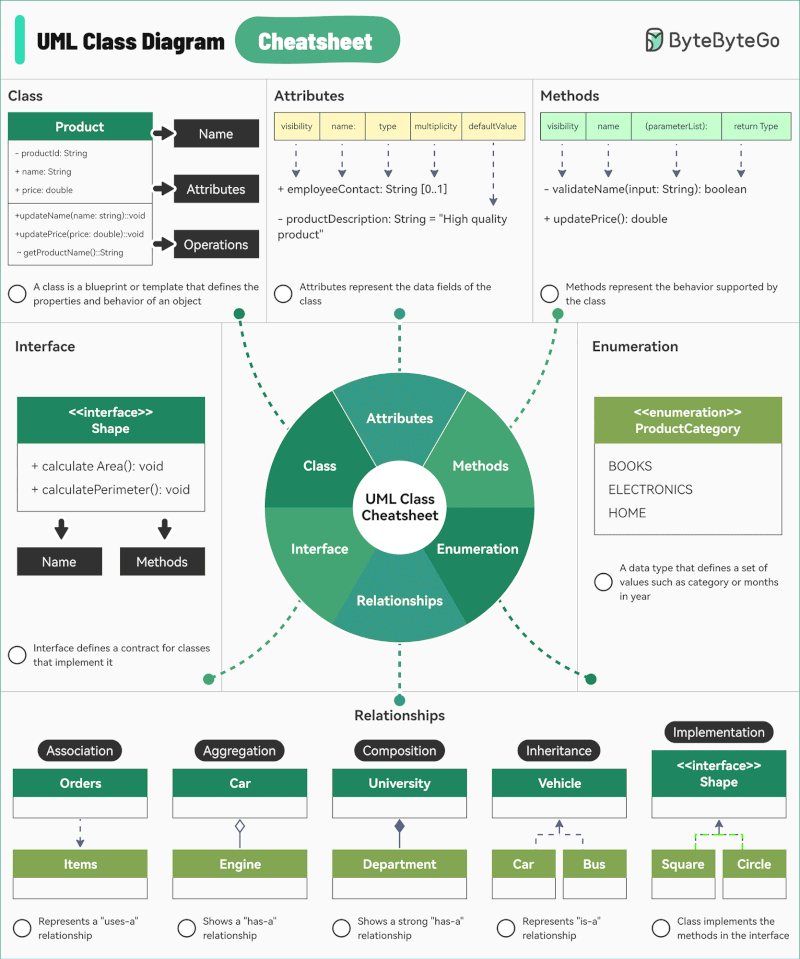


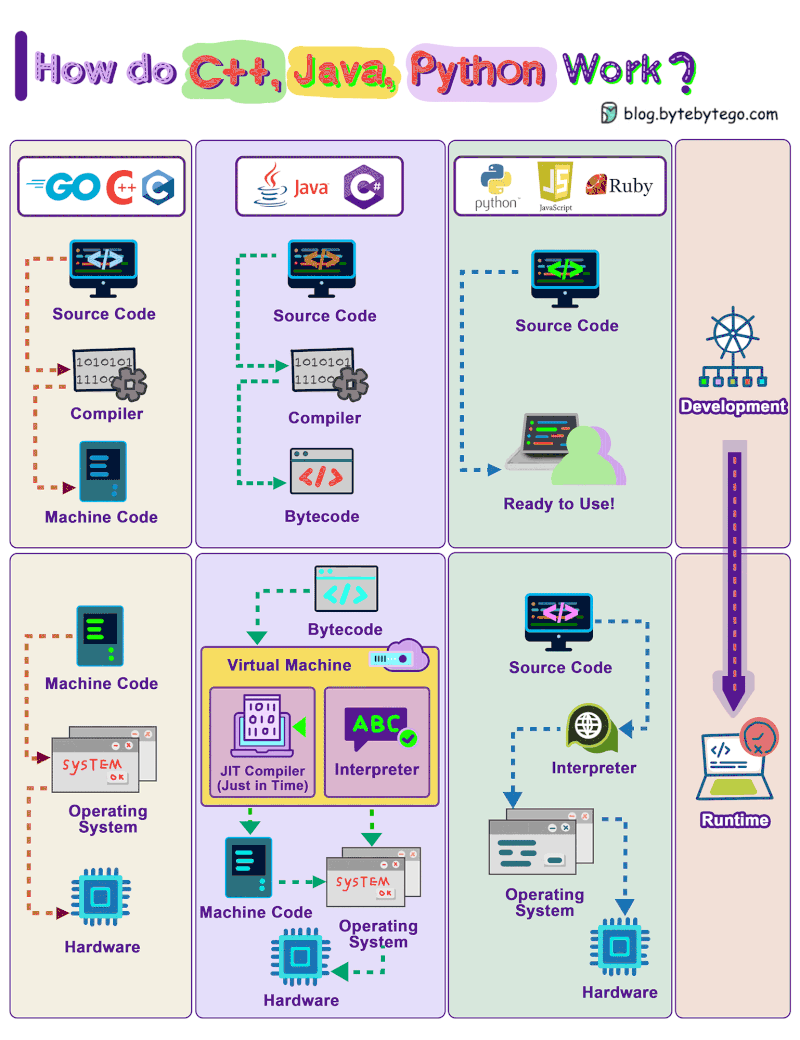
𝙃𝙤𝙬 𝘿𝙤𝙚𝙨 𝙩𝙝𝙚 𝙅𝙖𝙫𝙖 𝙂𝙖𝙧𝙗𝙖𝙜𝙚 𝘾𝙤𝙡𝙡𝙚𝙘𝙩𝙤𝙧 𝙒𝙤𝙧𝙠?   
  
The Java Garbage Collector automatically handles memory allocation and deallocation, making Java applications more robust and easier to develop. Here are the main concepts:  
  
1. Heap Memory:  
 - The area of memory where objects are dynamically allocated.  
 - Divided into different regions: Young Generation, Old Generation, and Permanent Generation (Metaspace in newer versions).  
  
2. Young Generation:  
 - Consists of Eden space and two Survivor spaces (S0 and S1).  
 - Newly created objects are allocated in the Eden space.  
 - Minor Garbage Collection occurs here.  
  
3. Old Generation:  
 - Stores long-lived objects that survived multiple Minor GCs.  
 - Major Garbage Collection (Full GC) occurs here.  
  
4. Permanent Generation (Metaspace):  
 - Stores metadata about classes and methods.  
 - Metaspace is used instead of Permanent Generation in Java 8 and later.



How does Garbage Collection work?   
.   
.   
Garbage collection is an automatic memory management feature used in programming languages to reclaim memory no longer used by the program.   
   
🔹 Java   
Java provides several garbage collectors, each suited for different use cases:   
   
1. Serial Garbage Collector: Best for single-threaded environments or small applications.   
   
2. Parallel Garbage Collector: Also known as the "Throughput Collector."   
   
3. CMS (Concurrent Mark-Sweep) Garbage Collector: Low-latency collector aiming to minimize pause times.   
   
4. G1 (Garbage-First) Garbage Collector: Aims to balance throughput and latency.   
   
5. Z Garbage Collector (ZGC): A low-latency garbage collector designed for applications that require large heap sizes and minimal pause times.   
   
🔹 Python   
Python's garbage collection is based on reference counting and a cyclic garbage collector:   
   
1. Reference Counting: Each object has a reference count; when it reaches zero, the memory is freed.   
   
2. Cyclic Garbage Collector: Handles circular references that can't be resolved by reference counting.   
   
🔹 GoLang   
Concurrent Mark-and-Sweep Garbage Collector: Go's garbage collector operates concurrently with the application, minimizing stop-the-world pauses.



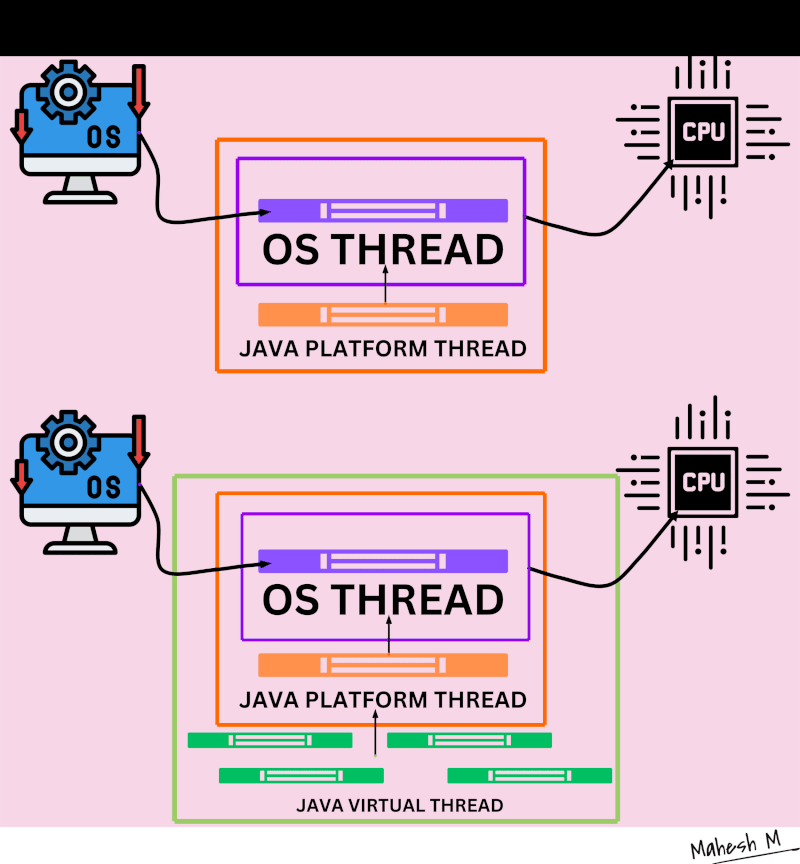






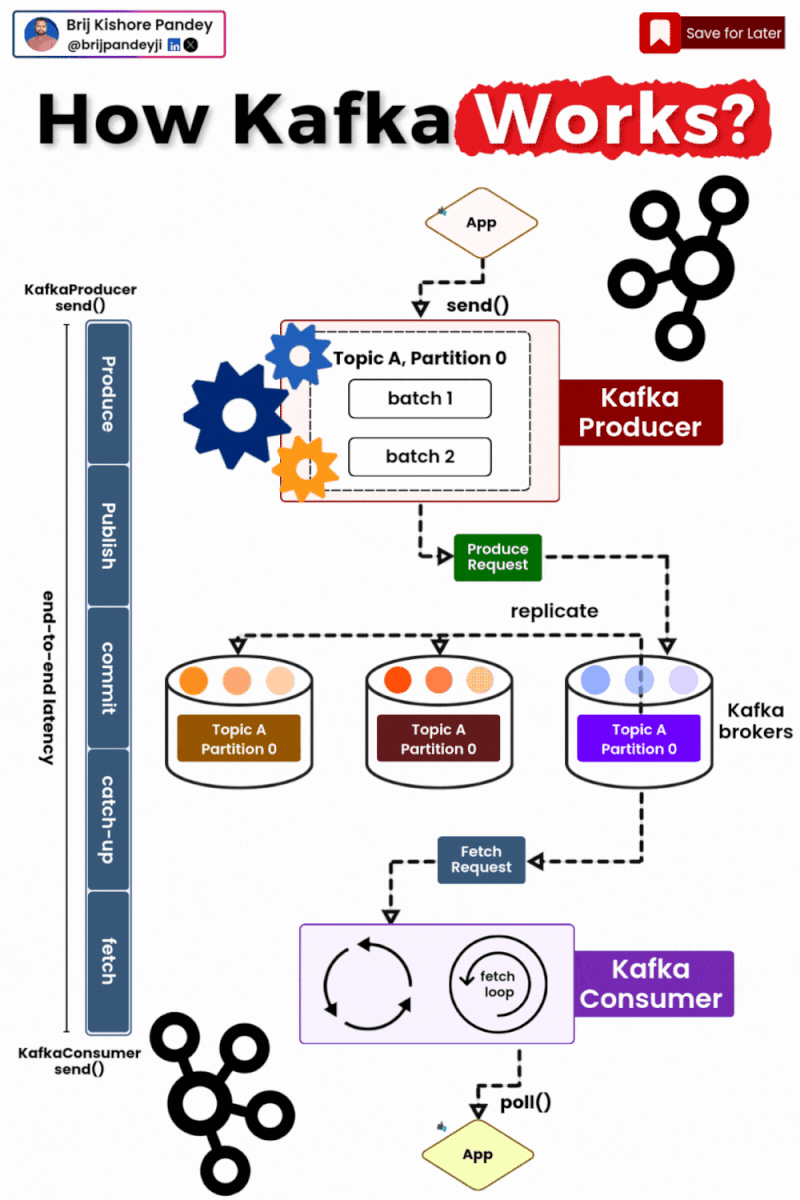
Java 17 brings a breath of fresh air to developers with its LTS (Long-Term Support) tag! 🌟 If you're like me, always striving for cleaner code, optimized performance, and enhanced security, then you're going to love what this latest release has to offer. Let’s break down some of the coolest features:  
  
1️⃣ Sealed Classes:  
  
Tired of worrying about which classes can extend yours? With Sealed Classes, you can now control which classes are permitted to inherit from your base class, making your hierarchy more secure and your code easier to maintain.  
  
🎯 Use case: Limit your API's extension points and protect your core logic.  
  
2️⃣ Pattern Matching for instanceof:  
  
No more redundant casting! Java 17 simplifies type-checking by introducing Pattern Matching with instanceof, making your code cleaner and easier to read.  
  
📈 Benefit: Reduces boilerplate code, improving code readability.  
  
3️⃣ Text Blocks:  
  
Long, multi-line strings? Say no more. Text Blocks in Java 17 make handling multi-line strings more intuitive and readable.  
  
💡 Impact: Improves readability, especially when dealing with JSON/XML data or SQL queries.  
  
4️⃣ Enhanced Performance with ZGC:  
  
Meet the Z Garbage Collector! It’s designed to handle large heaps with minimal latency. Whether you're building a high-throughput system or dealing with massive datasets, ZGC improves overall performance without affecting your app’s responsiveness.  
  
🚀 Performance boost: ZGC delivers low-latency garbage collection, making it perfect for modern, real-time applications.  
  
5️⃣ Strong Encapsulation with Security Manager Deprecation:  
  
Java 17 moves toward stronger security by deprecating the Security Manager and encouraging strong encapsulation by default. This means safer, more secure apps with fewer vulnerabilities.  
  
🔒 Result: Enhanced security with less overhead for developers!  
  
As we explore Java 17, it’s clear that this release is all about writing clearer, more efficient code with added security and performance optimizations. Whether you're building microservices, large-scale applications, or APIs, these features make a significant difference!  
  
🔑 Pro tip: If you're still on an older LTS version, now’s the perfect time to migrate and take advantage of these powerful new features!

Core Java Topics to clear next interview.  
  
  
Core Java is a fundamental part of Java programming, and interviewers often focus on these topics:  
  
𝗢𝗢𝗣 𝗖𝗼𝗻𝗰𝗲𝗽𝘁𝘀:   
 - Understand the principles of Object-Oriented Programming, like classes, objects, inheritance, encapsulation, and polymorphism.  
 - <https://lnkd.in/djNeVRex>  
 - <https://lnkd.in/dVFeWbnp>  
  
𝗗𝗮𝘁𝗮 𝗧𝘆𝗽𝗲𝘀 𝗮𝗻𝗱 𝗩𝗮𝗿𝗶𝗮𝗯𝗹𝗲𝘀:   
 - Know Java's primitive data types, their sizes, and usage. Understand how to declare and initialize variables.  
 - <https://lnkd.in/dC4WmRP3>  
  
𝗢𝗽𝗲𝗿𝗮𝘁𝗼𝗿𝘀:   
 - Learn about arithmetic, relational, logical, and bitwise operators.  
 - <https://lnkd.in/dTqrrEtp>  
 - <https://lnkd.in/dXtbFXHg>  
  
𝗖𝗼𝗻𝘁𝗿𝗼𝗹 𝗦𝘁𝗮𝘁𝗲𝗺𝗲𝗻𝘁𝘀:   
 - Be familiar with if-else, switch, and looping constructs (for, while, do-while).  
 - <https://lnkd.in/dBUgcWhN>  
 - <https://lnkd.in/dGbEM5TZ>  
  
𝗠𝗲𝘁𝗵𝗼𝗱𝘀:   
 - Understand method declaration, overloading, and overriding. Know how to pass arguments and return values.  
 - <https://lnkd.in/dfjkvm4X>  
 - <https://lnkd.in/dSRkNa8y>  
  
𝗔𝗿𝗿𝗮𝘆𝘀:   
 - Learn about single and multi-dimensional arrays, and how to manipulate them.  
 - <https://lnkd.in/dMDTpPm2>  
  
𝗘𝘅𝗰𝗲𝗽𝘁𝗶𝗼𝗻 𝗛𝗮𝗻𝗱𝗹𝗶𝗻𝗴:   
 - Know how to handle exceptions using try-catch blocks, and understand the hierarchy of exception classes.  
 - <https://lnkd.in/d7pgMRCJ>  
 - <https://lnkd.in/dE7MCH8j>  
  
𝗦𝘁𝗿𝗶𝗻𝗴 𝗛𝗮𝗻𝗱𝗹𝗶𝗻𝗴:   
 - Understand String class, string manipulation, and the importance of immutability.  
 - <https://lnkd.in/dA2nn79A>  
  
𝗖𝗼𝗹𝗹𝗲𝗰𝘁𝗶𝗼𝗻𝘀 𝗙𝗿𝗮𝗺𝗲𝘄𝗼𝗿𝗸:   
 - Familiarize yourself with collections like ArrayList, HashMap, and their usage.  
 - <https://lnkd.in/dJPdQhXK>  
  
𝗠𝘂𝗹𝘁𝗶-𝗧𝗵𝗿𝗲𝗮𝗱𝗶𝗻𝗴:   
 - Learn about creating and managing threads, synchronization, and thread safety.  
 - <https://lnkd.in/ddKr9rgz>  
  
𝗙𝗶𝗹𝗲 𝗛𝗮𝗻𝗱𝗹𝗶𝗻𝗴:   
 - Know how to read and write files in Java using input/output streams.  
 - <https://lnkd.in/d8g8f45b>  
  
𝗚𝗲𝗻𝗲𝗿𝗶𝗰𝘀:   
 - Understand how to use generic types to write more reusable and type-safe code.  
 - <https://lnkd.in/dwD7Bzss>  
  
𝗝𝗮𝘃𝗮 𝗜/𝗢:   
 - Learn about input and output streams, file handling, and serialization.  
  
𝗝𝗩𝗠 𝗮𝗻𝗱 𝗠𝗲𝗺𝗼𝗿𝘆 𝗠𝗮𝗻𝗮𝗴𝗲𝗺𝗲𝗻𝘁:   
 - Understand the Java Virtual Machine, garbage collection, and memory management.  
 - <https://lnkd.in/dEEAasFa>  
  
𝗝𝗮𝘃𝗮 𝗞𝗲𝘆𝘄𝗼𝗿𝗱𝘀:   
 - Be aware of important Java keywords like final, static, synchronized, etc.  
  
𝗟𝗮𝗺𝗯𝗱𝗮 𝗘𝘅𝗽𝗿𝗲𝘀𝘀𝗶𝗼𝗻𝘀:   
 - Understand how to use lambda expressions for functional programming.  
  
𝗝𝗮𝘃𝗮 𝗔𝗣𝗜:   
 - Familiarize yourself with important classes and packages in the Java API, such as java.lang, java.util, and [java.io](http://java.io/).  
  
𝗝𝗮𝘃𝗮 𝟴 𝗙𝗲𝗮𝘁𝘂𝗿𝗲𝘀:   
 - Learn about Java 8 features like Stream API, default methods, and the new Date and Time API.  
 - <https://lnkd.in/dMqx4dve>

Platform Thread Vs Virtual Thread  
  
✔️ A platform thread is a direct wrapper around an OS thread, executing Java code on the OS thread and maintaining this association throughout its lifetime. Therefore, the number of OS threads limits the number of available platform threads.  
  
✔️ Similar to a platform thread, a virtual thread is also an instance of java.lang.Thread. However, a virtual thread isn't tied to a specific OS thread, offering a new level of flexibility that empowers developers. When code running in a virtual thread calls a blocking I/O operation, the Java runtime suspends the virtual thread until it can be resumed, allowing the associated OS thread to perform operations for other virtual threads.  
  
The Java runtime, which is the part of the Java Virtual Machine that executes Java programs, efficiently manages a large number of virtual threads by mapping them to a small number of OS threads, effectively simulating a high thread count.  
  
𝗕𝗲𝗹𝗼𝘄 𝗮𝗿𝗲 𝘀𝗼𝗺𝗲 𝗼𝗳 𝘁𝗵𝗲 𝗯𝗲𝗻𝗲𝗳𝗶𝘁𝘀 𝗼𝗳 𝗩𝗶𝗿𝘁𝘂𝗮𝗹 𝗧𝗵𝗿𝗲𝗮𝗱𝘀   
  
 ⏹️Improved scalability of the applications  
 ⏹️Improved resource efficiency  
 ⏹️Reduces the overall complexity of dealing with concurrency  
 ⏹️Virtual threads are lightweight to work with  
 ⏹️Seamless integration between Virtual Threads and CompletableFuture  
 ⏹️Develop high-throughput concurrent applications  
 ⏹️It provides high throughput and lower latency.  


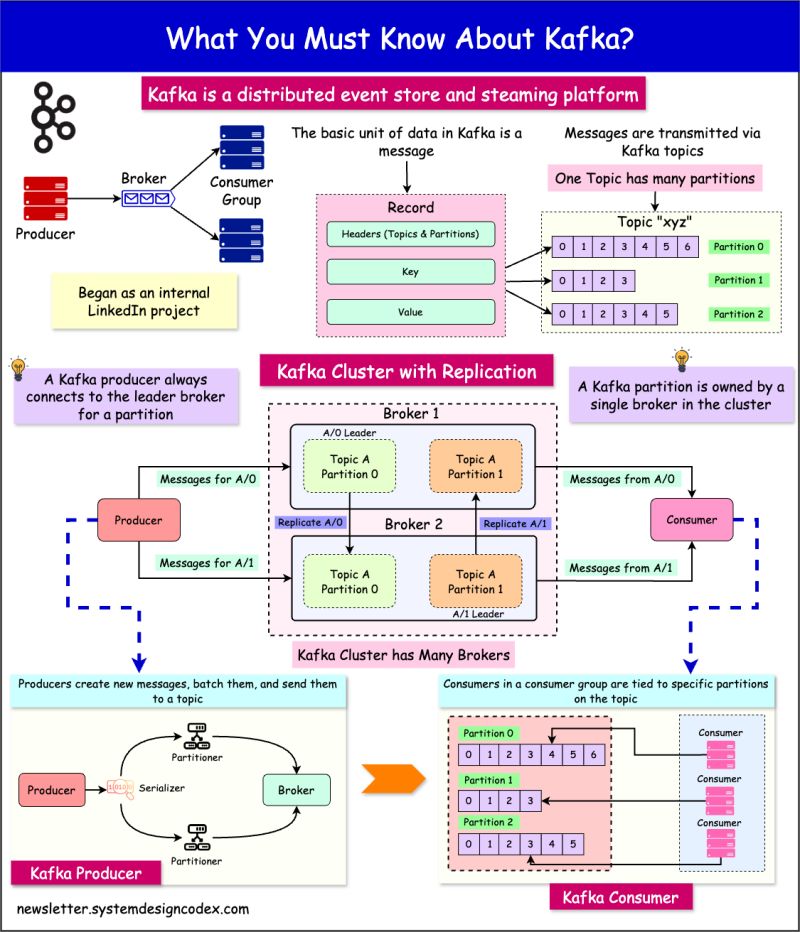
When it comes to data streaming, [hashtag#𝐫𝐞𝐬𝐢𝐥𝐢𝐞𝐧𝐜𝐲](https://www.linkedin.com/feed/hashtag/?keywords=resiliency&highlightedUpdateUrns=urn%3Ali%3Aactivity%3A7244900292033462272) is non-negotiable. Kafka, being a distributed system, offers multiple ways to ensure high availability and fault tolerance. Let's dive into some essential Kafka resiliency techniques that ensure your design is ready to handle failures.  
  
───── 𝐃𝐚𝐭𝐚 𝐑𝐞𝐬𝐢𝐥𝐢𝐞𝐧𝐜𝐲 ─────  
  
↳ 𝐑𝐞𝐩𝐥𝐢𝐜𝐚𝐭𝐢𝐨𝐧 𝐅𝐚𝐜𝐭𝐨𝐫: Always increase the number of replicas to safeguard against broker failures.  
↳ 𝐀𝐜𝐤𝐧𝐨𝐰𝐥𝐞𝐝𝐠𝐞𝐦𝐞𝐧𝐭 𝐒𝐞𝐭𝐭𝐢𝐧𝐠𝐬: Use 𝘢𝘤𝘬𝘴 settings wisely to ensure messages are fully replicated before committing.  
↳ 𝐌𝐢𝐧 𝐈𝐧-𝐒𝐲𝐧𝐜 𝐑𝐞𝐩𝐥𝐢𝐜𝐚𝐬: Set a minimum number of replicas that must be synced before acknowledging writes.  
↳ 𝐃𝐢𝐬𝐤 𝐒𝐩𝐚𝐜𝐞 𝐌𝐚𝐧𝐚𝐠𝐞𝐦𝐞𝐧𝐭: Monitor and manage disk usage to avoid data loss.  
  
───── 𝐁𝐫𝐨𝐤𝐞𝐫 𝐑𝐞𝐬𝐢𝐥𝐢𝐞𝐧𝐜𝐲 ─────  
  
↳ 𝐌𝐮𝐥𝐭𝐢-𝐃𝐚𝐭𝐚𝐜𝐞𝐧𝐭𝐞𝐫 𝐃𝐞𝐩𝐥𝐨𝐲𝐦𝐞𝐧𝐭: Spread your brokers across multiple data centres to prevent data loss in case of a failure.  
↳ 𝐑𝐚𝐜𝐤 𝐀𝐰𝐚𝐫𝐞𝐧𝐞𝐬𝐬: Ensure replicas are spread across availability zones to avoid single points of failure.  
↳ 𝐁𝐫𝐨𝐤𝐞𝐫 𝐇𝐞𝐚𝐥𝐭𝐡 𝐌𝐨𝐧𝐢𝐭𝐨𝐫𝐢𝐧𝐠: Monitor brokers' health to automate failovers.  
  
───── 𝐓𝐨𝐩𝐢𝐜 & 𝐏𝐫𝐨𝐝𝐮𝐜𝐞𝐫 𝐑𝐞𝐬𝐢𝐥𝐢𝐞𝐧𝐜𝐲 ─────  
  
↳ 𝐏𝐚𝐫𝐭𝐢𝐭𝐢𝐨𝐧 𝐌𝐚𝐧𝐚𝐠𝐞𝐦𝐞𝐧𝐭: Evenly distribute partitions to avoid overloading a single broker.  
↳ 𝐈𝐝𝐞𝐦𝐩𝐨𝐭𝐞𝐧𝐭 𝐏𝐫𝐨𝐝𝐮𝐜𝐞𝐫𝐬: Avoid duplicate messages with idempotent producers.  
↳ 𝐑𝐞𝐭𝐫𝐢𝐞𝐬 𝐚𝐧𝐝 𝐁𝐚𝐜𝐤𝐨𝐟𝐟: Configure retries and exponential backoff for transient errors.  
  
───── 𝐎𝐩𝐞𝐫𝐚𝐭𝐢𝐨𝐧𝐚𝐥 𝐑𝐞𝐬𝐢𝐥𝐢𝐞𝐧𝐜𝐲 ─────  
  
↳ 𝐁𝐚𝐜𝐤𝐮𝐩 𝐚𝐧𝐝 𝐑𝐞𝐬𝐭𝐨𝐫𝐞: Regular backups ensure quick recovery from failures.  
↳ 𝐃𝐢𝐬𝐚𝐬𝐭𝐞𝐫 𝐑𝐞𝐜𝐨𝐯𝐞𝐫𝐲: Regularly test your disaster recovery plans to keep your system resilient.

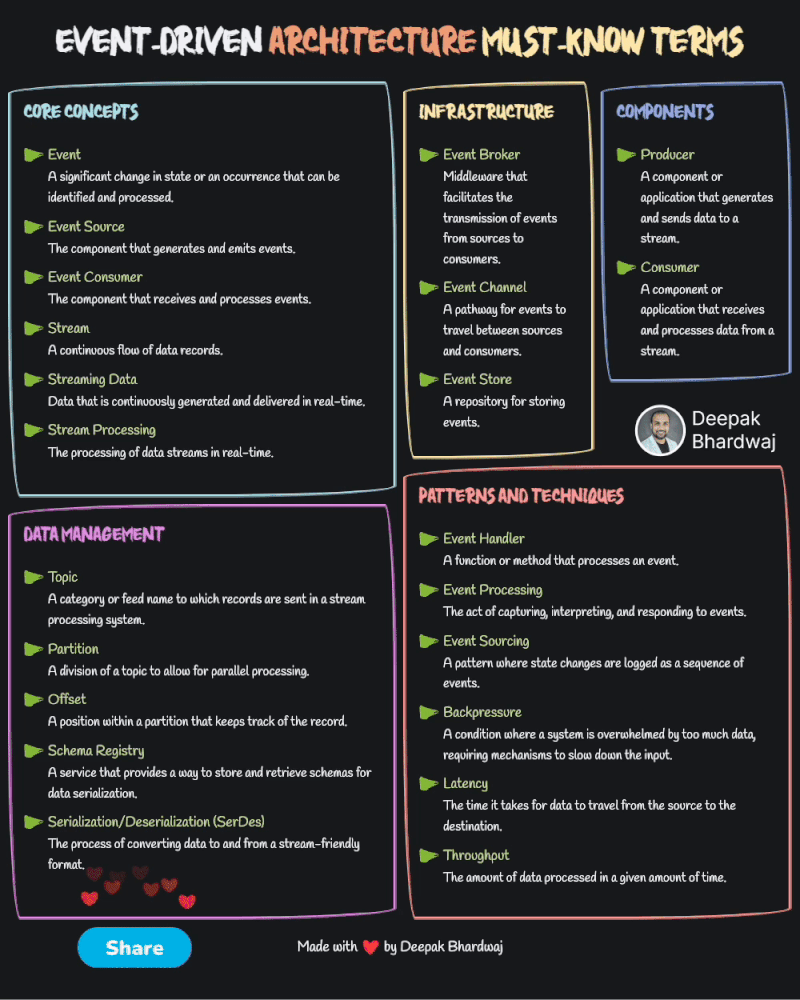




Apache Kafka has become the de facto standard for building real-time data pipelines and streaming applications.   
  
As a distributed event streaming platform, it has revolutionized  how companies handle data flows.   
  
Let's dive into what makes Kafka tick.  
  
 Key Concepts:  
  
1. Topics:   
 • Think of these as categories or feed names  
 • Messages are published to topics  
 • Can have multiple partitions for parallelism  
  
2. Partitions:  
 • Ordered, immutable sequence of records  
 • Each record assigned a sequential ID (offset)  
 • Enables massive scalability  
  
3. Producers:  
 • Publish messages to topics  
 • Can choose which partition to send messages to  
  
4. Consumers:  
 • Subscribe to topics and process the messages  
 • Track their position (offset) in each partition  
  
5. Brokers:  
 • Kafka servers that store and manage topics  
 • A cluster typically has multiple brokers for fault tolerance  
  
6. KRaft:  
 • Manages the Kafka cluster  
 • Tracks broker health, topic configuration, and more  
  
How It All Connects:  
  
1. Message Flow:  
 Producer → Broker → Consumer  
  
2. Partition Leadership:  
 • Each partition has one leader broker and multiple replicas  
 • Writes go to the leader, replicas stay in sync  
  
3. Consumer Groups:  
 • Multiple consumers can work together  
 • Each partition is read by only one consumer in a group  
  
4. Offset Management:  
 • Consumers commit their offset after processing  
 • Enables restart from last position if a consumer fails  
  
5. Retention:  
 • Messages can be retained for a configured time or size  
 • Enables replay and catch-up scenarios  
  
Key Features:  
  
• High Throughput: Can handle millions of messages per second  
• Fault Tolerance: Replication ensures data safety  
• Scalability: Easy to scale out by adding more brokers  
• Low Latency: Sub-10 ms latency in production environments  
• Durability: Data persisted to disk, surviving broker failures  
  
Use Cases:  
  
• Event Sourcing  
• Log Aggregation  
• Stream Processing  
• Metrics Collection  
• Activity Tracking  
  
  
Kafka's architecture enables decoupling of data streams and systems. This makes it invaluable for building real-time data pipelines and streaming applications.

✅ What is Apache Kafka?  
  
Apache Kafka is a distributed event store and streaming platform.  
  
It began as an internal project at LinkedIn.  
  
Over time, it grew rapidly and today, some of the largest data pipelines in the world use Kafka.  
  
Organizations like Netflix and Uber rely on it for their workflows.  
  
✅ Kafka Messages, Topics, and Partitions  
  
The basic unit of data in Kafka is a Message  
  
Think of a message like a record in a database table. It is transmitted as an array of bytes.  
  
Every message goes to a particular Topic.  
  
You can compare Kafka Topics to a database table or a folder on your computer.  
  
Topics are also made up of multiple partitions.  
  
Partitions improve the redundancy and make the topics horizontally scalable.  
  
✅ Kafka Producers and Kafka Consumer  
  
Producers in Kafka create new messages, batch them, and send them over to a Kafka topic.  
  
A producer also balances messages across the different partitions of a topic.  
  
You can provide a custom partitioning strategy to control the distribution of messages.  
  
Kafka Consumers read messages from a broker.  
  
One or more consumers work as a consumer group to consume messages from a topic.  
  
A consumer instance is tied to a particular partition.  
  
In other words, a partition is owned by a consumer instance.  
  
✅ Kafka Broker and Cluster  
  
A single Kafka server is known as a Broker.  
  
A broker can handle thousands of partitions and millions of messages/second.  
  
Think of the broker as a bridge between the producer and consumer.  
  
It receives messages from producers and handles fetch requests from the consumer.  
  
But the broker as part of a Kafka Cluster  
  
A Kafka Cluster consists of several brokers and provides features like replication.  
  
Every partition is replicated across multiple brokers ensuring high-availability and redundancy.  
  
✅ Common Use Cases of Kafka  
  
👉 Tracking user activity on front-end application  
  
👉 Messaging requirements in a distributed system such as notifications and emails to users  
  
👉 Metrics collection and logging





Here are some 𝐦𝐮𝐬𝐭-𝐤𝐧𝐨𝐰 𝐭𝐞𝐫𝐦𝐬 to get started:  
  
🔘 𝐂𝐨𝐫𝐞 𝐂𝐨𝐧𝐜𝐞𝐩𝐭𝐬:  
  
↳ 𝐄𝐯𝐞𝐧𝐭: A significant change in state or occurrence that can be identified and processed.  
↳ 𝐄𝐯𝐞𝐧𝐭 𝐒𝐨𝐮𝐫𝐜𝐞: The component that generates and emits events.  
↳ 𝐄𝐯𝐞𝐧𝐭 𝐂𝐨𝐧𝐬𝐮𝐦𝐞𝐫: The component that receives and processes events.  
↳ 𝐒𝐭𝐫𝐞𝐚𝐦: A continuous flow of data records.  
↳ 𝐒𝐭𝐫𝐞𝐚𝐦𝐢𝐧𝐠 𝐃𝐚𝐭𝐚: Data that is continuously generated in real-time.  
↳ 𝐒𝐭𝐫𝐞𝐚𝐦 𝐏𝐫𝐨𝐜𝐞𝐬𝐬𝐢𝐧𝐠: The real-time processing of data streams.  
  
🔘 𝐈𝐧𝐟𝐫𝐚𝐬𝐭𝐫𝐮𝐜𝐭𝐮𝐫𝐞:  
  
↳ 𝐄𝐯𝐞𝐧𝐭 𝐁𝐫𝐨𝐤𝐞𝐫: Middleware that facilitates the transmission of events from sources to consumers.  
↳ 𝐄𝐯𝐞𝐧𝐭 𝐂𝐡𝐚𝐧𝐧𝐞𝐥: A pathway for events to travel between sources and consumers.  
↳ 𝐄𝐯𝐞𝐧𝐭 𝐒𝐭𝐨𝐫𝐞: A repository for storing events.  
  
🔘 𝐂𝐨𝐦𝐩𝐨𝐧𝐞𝐧𝐭𝐬:  
  
↳ 𝐏𝐫𝐨𝐝𝐮𝐜𝐞𝐫: Generates and sends data to a stream.  
↳ 𝐂𝐨𝐧𝐬𝐮𝐦𝐞𝐫: Receives and processes data from a stream.  
  
🔘 𝐃𝐚𝐭𝐚 𝐌𝐚𝐧𝐚𝐠𝐞𝐦𝐞𝐧𝐭:  
  
↳ 𝐓𝐨𝐩𝐢𝐜: A category or feed for stream processing.  
↳ 𝐏𝐚𝐫𝐭𝐢𝐭𝐢𝐨𝐧: A topic division to allow for parallel processing.  
↳ 𝐎𝐟𝐟𝐬𝐞𝐭: A position that tracks a record in a partition.  
↳ 𝐒𝐜𝐡𝐞𝐦𝐚 𝐑𝐞𝐠𝐢𝐬𝐭𝐫𝐲: A service to store and retrieve schemas for data serialisation.  
↳ 𝐒𝐞𝐫𝐃𝐞𝐬 (Serialization/Deserialization): Converts data to/from stream-friendly formats.  
  
🔘 𝐏𝐚𝐭𝐭𝐞𝐫𝐧𝐬 𝐚𝐧𝐝 𝐓𝐞𝐜𝐡𝐧𝐢𝐪𝐮𝐞𝐬:  
  
↳ 𝐄𝐯𝐞𝐧𝐭 𝐇𝐚𝐧𝐝𝐥𝐞𝐫: Processes an event.  
↳ 𝐄𝐯𝐞𝐧𝐭 𝐏𝐫𝐨𝐜𝐞𝐬𝐬𝐢𝐧𝐠: Captures, interprets, and responds to events.  
↳ 𝐄𝐯𝐞𝐧𝐭 𝐒𝐨𝐮𝐫𝐜𝐢𝐧𝐠: Logs state changes as events.  
↳ 𝐁𝐚𝐜𝐤𝐩𝐫𝐞𝐬𝐬𝐮𝐫𝐞: Slows down input when overwhelmed by data.  
↳ 𝐋𝐚𝐭𝐞𝐧𝐜𝐲: Time taken for data to travel from source to destination.  
↳ 𝐓𝐡𝐫𝐨𝐮𝐠𝐡𝐩𝐮𝐭: The amount of data processed in a given time.  
  
EDA transforms our thoughts about 𝐬𝐲𝐬𝐭𝐞𝐦𝐬 𝐚𝐫𝐜𝐡𝐢𝐭𝐞𝐜𝐭𝐮𝐫𝐞, enabling more reactive, scalable, and decoupled solutions!

