

APACHE KAFKA ARCHITECTURE ANALYSIS REPORT

Group Name: NABER

- İlayda Akınet - 21050911023
- Hatice Rüveyda Akça - 21050911006
- Yahya Çakıcı - 22050951010

Contents

Part 1

1. Task I: Application and Tool Description

- 1.1. Introduction to Apache Kafka
- 1.2. Apache Kafka Use Cases
- 1.3. Apache Kafka in Action: Use Cases Across Industries
- 1.4. Kafka Functionality
- 1.5. When Not to Use Apache Kafka
- 1.6. Limitations and Considerations
- 1.7. Difference
- 1.8. Tools Intended to Use

2. Task II: Documented the Prescriptive (Intended) Architecture

- 2.1. Architectural Styles Used in Kafka's Evolution
- 2.2. Design Patterns
- 2.3. Key Concepts
- 2.4. Performance
- 2.5. Key Features of Apache Kafka Architecture

Part 2

3. Task III: Analyze the Descriptive Architecture of Apache Kafka

1. Introduction

2. Architectural Styles and Patterns

2.1. Event-Driven Architecture

2.2. Publish-Subscribe Model

2.3. Integration of Third-Party Dependencies

2.4. Pattern Determinations:

2.5. Component Analysis:

1. Producers
2. Consumers
3. Brokers
4. Topics and Partitions
5. Zookeeper and Metadata Management
6. Kafka Streams
7. Kafka Connect API

2.6. Graph of Components

3. Modularity, Cohesion and Coupling

3.1. Modularity

3.2. Cohesion

3.3. Coupling

4. Scalability, Reliability, and Performance

4.1. Scalability

4.2. Reliability

4.3. Performance

4. Task IV: Apache Kafka Architecture Review and Improvement Suggestions

1. Current Architecture Review

1.1. Drawing Architecture

1.2 Code Quality

1.3. Technical Debt and Quality Analysis

1.4. Duplicated Code

1.5. Code Smells

2. Maintainability

2.1. Evaluation of Maintainability

2.2. Suggestions For Increasing Maintainability

3. Security Review

3.1. Analyzing With SonarQube

3.2. Group Security Assessment

4. Fault Tolerance

5. Detailed Visual and Analytical Support

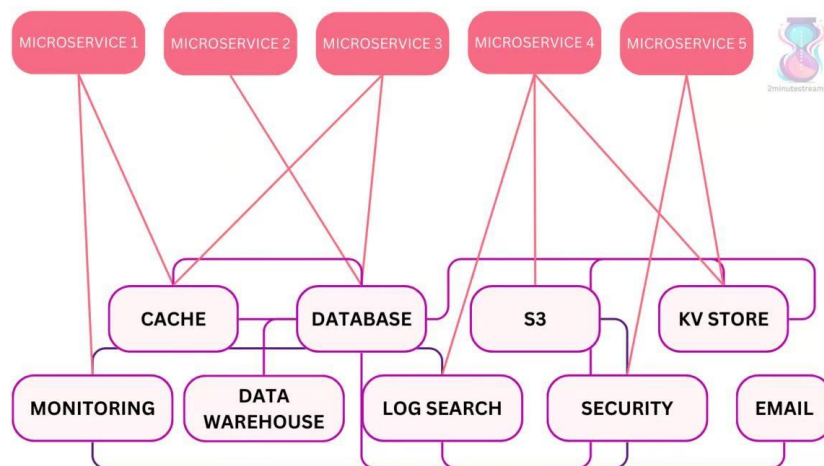
6. Conclusion and Action Plan

REFERENCES

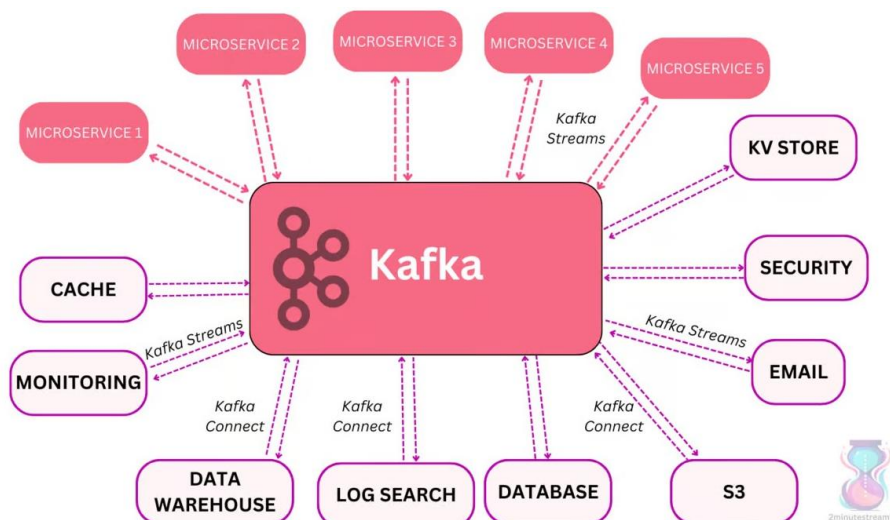
TASK I: Application and Tool Description

Introduction to Apache Kafka

Apache Kafka is an open-source stream-processing software platform developed by LinkedIn and donated to the Apache Software Foundation. It is written in Scala and Java. The basic architectural concept of the solution is an immutable log of messages that can be organized in topics to be consumed by several users or applications. Its development is path-dependent on the problems which hit LinkedIn at that time. Since they were among the first companies to encounter large-scale distributed systems, they noted the common problem of uncontrolled microservice proliferation.



In an effort to resolve the growing complexity of service-to-service and persistence store permutations, they developed a single platform that could become a source of truth.



Today Kafka is a powerful resource that can operate huge volumes of information. It makes it scalable, fault-tolerant, and high data throughput capable, hence solving several problems of service coordination and proliferation of microservices. Its popularity has been raised for current modern systems in the line of real-time data processing.

Apache Kafka Use Cases

- **Messaging**

Kafka is great for use cases where you need to separate data producers from processors and store unprocessed messages temporarily. It stands out from other messaging systems because it handles large volumes of messages, supports data partitioning, and is fault-tolerant. This makes Kafka ideal for large-scale applications.

- **Tracking Web Activities**

Kafka was originally designed for real-time tracking of user activity, like page views and searches, by publishing each action as a message in a topic. Different applications can subscribe to these topics to process data in real-time, monitor activity, or load it into systems like Hadoop for offline analysis. Since each user action generates a message, activity tracking can lead to high message volumes.

- **Metrics**

Kafka also has a wide usage in operational data monitoring. It aggregates the statistics of multiple applications and offers a single spot where you can go to get your operational information.

- **Log Aggregation**

Many people use Kafka for log aggregation: collecting log files from many servers and centralizing them for easier processing. It does this greatly by conceptualizing log data as one long stream of messages, hence making it easier to process. It also easily integrates with various data sources. Compared to other log-centric systems like Scribe or Flume, Kafka has similar performance but with better durability and less latency.

- **Stream Processing**

Commonly, Kafka is used within data processing pipelines: raw data is pulled from Kafka topics, transformed, and then published to new topics for further use. For example, a system might gather article content from various sources, clean it up, and recommend it to users. With Kafka Streams, an efficient stream-processing library, such data processing workflows can be easily built by users. Other alternatives for stream processing are Apache Storm and Apache Samza.

- **Event Sourcing**

Event sourcing is a design approach where state modifications are captured as a series of events. The capability of Kafka to handle massive volumes of log data makes it a perfect backend to use this approach.

- **Commit Log**

This provides a unified way to synchronize data between nodes and, if needed, restore the data of a failed node. Since it has log compaction, Kafka can be used for that purpose just like the Apache BookKeeper project.

Apache Kafka in Action: Use Cases Across Industries

Apache Kafka has emerged as a crucial technology for many organizations looking to harness the power of real-time data processing and analytics. Below are some notable companies that leverage Kafka in diverse applications:

- **LinkedIn**

LinkedIn originally developed Kafka to handle user activity tracking, aggregating massive data volumes from member interactions like profile views, messages, and content shares. Kafka enabled LinkedIn to process and analyze user behavior in real-time, driving features like personalized recommendations and targeted advertising while handling high traffic and continuous data streams.

- **Netflix**

Netflix employs Kafka for real-time monitoring and event-processing, ensuring that they can respond quickly to user demands and maintain high service quality. This use of Kafka allows them to streamline their operations and enhance the viewing experience for millions of subscribers.

- **Adidas**

Adidas integrates Kafka as the core of its Fast Data Streaming Platform. This enables the seamless connection of various source systems, allowing teams to implement real-time event processing for monitoring, analytics, and reporting solutions. Kafka's robust architecture supports Adidas in making data-driven decisions quickly and efficiently.

- **Cloudflare**

At Cloudflare, Kafka plays a pivotal role in their log processing and analytics pipeline. The platform collects hundreds of billions of events per day from thousands of servers, enabling Cloudflare to deliver insights and maintain high-performance standards for their services.

- **Coursera**

Kafka powers Coursera's educational platform at scale, serving as the backbone for real-time learning analytics and dashboards. This allows educators and students to access vital data on learning patterns and performance, enhancing the overall educational experience.

- **The New York Times**

The New York Times uses Apache Kafka and the Kafka Streams API to store and distribute published content in real-time. This system feeds various applications and systems, ensuring that readers receive timely updates and news articles seamlessly.

- **Oracle**

Oracle offers native connectivity to Kafka through its Enterprise Service Bus product, Oracle Service Bus (OSB). This integration allows developers to leverage OSB's built-in mediation capabilities to create staged data pipelines, enhancing data flow and processing.

- **PayPal**

At PayPal, Kafka is integral to multiple use cases, including first-party tracking, streaming application health metrics, database synchronization, and application log aggregation. With each of these use cases processing over 100 billion messages per day, Kafka empowers PayPal to ensure compliance and manage risks effectively.

- **Twitter**

Twitter employs Kafka as part of its Storm stream processing infrastructure, allowing the platform to handle real-time data efficiently. This use of Kafka enables Twitter to maintain the high speed and reliability that users expect from a leading social media service.

Kafka Functionality

1. Scalability

One of the reasons Apache Kafka is so phenomenal is that it can scale along manifold axes—producers, brokers, and consumers. Kafka handles the workload with ease, regardless of the number of consumer groups requesting data or the few producers spitting out large volumes of datasets. Kafka pulls this off by spreading data out across partitions and brokers, which enables high-throughput applications. Because Kafka is horizontally scalable, it can handle additional brokers with minimal performance degradation as more data producers or consumers are added. Hence, this also makes it quite ideal for big data or workloads that can become quite variable within organizations.

Use Case Example: While capturing customer interactions and order data in real time across different countries for a multinational e-commerce company, scaling a Kafka cluster easily by adding brokers is realized.

2. Volume of Data-handling Capacity

Kafka is designed to handle enormous volumes of data, from small byte-sized messages to petabytes of data daily. It will process large-scale, high-volume flows of data with ensured performance for applications requiring high-velocity streams of data, including IoT, stock trading, and real-time monitoring. Kafka achieves this through the storage of data across partitions so that data can be parallelly processed to achieve much higher throughput rates.

Use Case Example: In a financial trading application, Kafka can ingest millions of financial transactions per second, thereby allowing real-time market analysis and decision-making.

3. Guaranteed Exactly-Once Processing

Kafka guarantees that each message is being processed exactly once. In this respect, Kafka prevents duplicate delivery through orchestrated use of its distributed brokers. The prevention of duplicate delivery is one of those things that enterprise systems appreciate in situations where some types of data cannot be fake. For instance, financial transaction records or critical system logs. Since Kafka operates in a distributed fashion, it would be aware of those messages that have been successfully processed and hence would avoid duplicates even under conditions of system failure.

Use Case Example: A banking application using Kafka guarantees that transactions are processed once, which avoids issues such as double charging or other forms of data misprocessing of critical records.

4. Guaranteed Ordering of Data

Many real-time applications regard the preservation of message order as key to their functions. Kafka allows one to do so, it guarantees to preserve message order in a partition. Kafka horizontally scales across a large number of servers and partitions, maintaining the order of messages in a partition. Consumers reading from a particular partition will always get the messages in order that they were produced.

Use Case Example: Logistics tracking system-Kafka ensures that event data is processed in the correct order; for instance, an event indicating that a package has been scanned will result in discrepancies in the tracking history.

5. Highly Reliable and Fault-Tolerant

Kafka natively supports strong reliability features by way of replication. Thus, each partition can be replicated across multiple brokers, which means that in the event of failure of a broker, the data is still available from another broker. The number of replicas is configurable, with a balance between fault tolerance and storage costs. This makes it a critical feature in environments where data loss is not an option. It allows for high availability and disaster recovery.

Use Case Example: For example, in social media, Kafka can be used to provide real-time replication and storage of user-generated content across various data centers with ensured high availability even in case a server fails.

6. Offsite Replication for Disaster Recovery

Kafka can also replicate data across data centers or different geographical locations. This provides the promise of data availability even in case of major failure at one site. This off-site replication helps an organization to keep its business continuity and data loss at minimum in such natural disasters or data center outage.

Use Case Example: An internationally operating company may use Kafka to replicate transaction logs across various regions' data centers. This way, it ensures that fundamental systems remain operational in case a region goes down.

7. High Performance with Minimum Resource Usage

Kafka is designed for production usage to offer high-performance irrespective of the volume of handled data. Kafka works effectively with large streams of data without needing additional server resources. High throughput, low latency, and durability are facilitated by the log-based storage mechanism in Kafka. Besides, it employs zero-copy techniques so that the disks' I / O is optimized to ensure better performance.

Use Case Example: In an ad-tech company, which needs to process millions of events incoming every second from web interactions, Kafka handles huge volumes of real-time data at low latency. It thus enables real-time bidding and personalization.

8. Zero Downtime for Upgrades

Kafka supports rolling upgrades, which means that an organization can upgrade the Kafka clusters without causing any downtime. That's a big deal with mission-critical applications that cannot afford to be interrupted. The administrators would be in a position to carry out upgrades or other forms of maintenance without affecting the Kafka cluster's availability, reducing service disruption to a minimum.

Use Case Example: Within a large-scale CDN, Kafka ensures that data-streaming-related services are not disrupted during upgrade cycles or patching of an infrastructure.

9. Extensibility and Integration with the Ecosystem

The popularity of Kafka has brought out a huge ecosystem of integrations wherein Kafka extends itself with ease to connect and integrate with other tools and platforms. Examples include Kafka Connect, which features pre-built connectors for databases, file systems, and cloud platforms. The developers now can easily build robust data pipelines by connecting Kafka with external systems, both as sources and data sinks.

Use Case Example: This could enable a data analytics company to use Kafka Connect to integrate seamlessly with either Hadoop or Spark and provide the ability to build powerful real-time data pipelines for both machine learning and analytics.

10. Open Source and Free to Use

Kafka is open-source software, and because of that, no licensing fees are relevant, nor any restrictions to its usage. Initially developed by LinkedIn and then donated to the Apache Software Foundation, it has since grown into a truly community-driven project with active contributions from the biggest tech companies. The open-sourced nature is one of the key drivers towards Kafka's adoption. Being open source makes Kafka more attractive for startups who seek solutions that do not involve high costs.

Use Case Example: For startups on shoestring budgets, Kafka's open-source platform provides an entry point to create sophisticated data pipelines without having to pay expensive licensing fees, thus scaling infrastructure with the growth of the business.

When Not to Use Apache Kafka

Apache Kafka is an all-powerful platform for real-time data processing and streaming; it does feel overkill for each use case. Following are the key points one should keep in mind when not using Kafka:

1. Small Data Sets

Kafka has been designed to handle high throughput and large volumes of data. If the environment running Kafka is only dealing with a few thousand messages per day, then you may have problems with "too many open files" or "out of memory" just due to the overhead with the system.

2. Complex Message Transformations

Kafka provides a Stream API for processing data; however, it is not the appropriate tool for complex message transformations or complex ETL. The heavy transformations should be implemented, along with their maintenance, including cumbersome pipelines.

3. Task Queue Use Cases

Kafka was never designed to be a basic task queue. If your use case involves only simple management of tasks, then this can and should be better handled by tools designed for the purpose. Solutions actually focused on that will be far easier to work with for managing tasks.

4. Replacing a Database

Kafka is not intended to replace database systems. It lacks all the core features that any regular database would have: data storage, indexing, transaction support-a lot of things. In fact, most systems encounter great difficulties when they try to use Kafka as a database. Mainly, this will be because of the lack of data consistency and/or expression mechanisms for defining how to query the data.

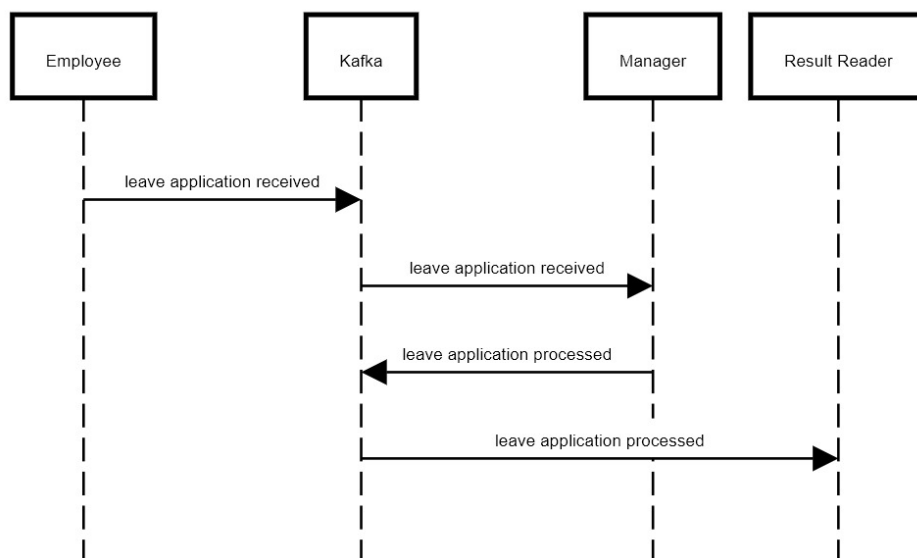
Limitations and Considerations

- **Overhead Issues:** Kafka has very high overhead; hence, for little data, it is not suitable, and therefore it causes resource challenges.
- **Complexity in Transformations:** The architecture is sensitive to heavy ETL. It is not suitable for complex message transformations.
- **Database Features Not Present:** It doesn't have common database features such as indexes and transaction guarantees, and hence it's not well suited for tasks that are like those of databases.

Kafka's Difference from Other Applications

Unlike traditional message queues, Kafka does a few things differently. A big difference is that messages in Kafka are kept for a certain time after being consumed-usually up to seven days. Most traditional messaging queues delete messages immediately after clients acknowledge them. Traditional queues also push the messages to the client and keep the load of maintaining the clients and deciding how many messages to serve each client.

On the other hand, Kafka allows a pull model where clients request messages at their own pace, such that when the client is ready to process messages. Kafka is built to support horizontal scaling, and hence, anybody can easily add more nodes in it to increase its capacity. Traditional messaging queues let one do vertical scaling, which increases the power of already existing servers. These are the basic differences between Kafka and traditional messaging systems, which reflect how Kafka is built to capture large volumes of data, unlike typical messaging systems.



Tools Intended to Use

1-Kafka Command-Line Tools

- **Analysis Type:** Basic architectural inspection of Kafka topics, partitions, and consumer groups.
- **Pros:** Direct access to Kafka's components, allowing a detailed understanding of partitioning, replication, and consumer behavior.
- **Cons:** Limited visualization capabilities and requires manual interaction.
- **Usage:** `kafka-topics.sh` for topic and partition details, `kafka-consumer-groups.sh` for consumer group analysis, and `kafka-configs.sh` for broker configuration insights.

2-Prometheus and Grafana

- **Analysis Type:** Real-time monitoring of Kafka's performance metrics like throughput, latency, and consumer lag.
- **Pros:** Powerful for tracking operational health, with customizable dashboards.
- **Cons:** Requires setup and configuration, and Prometheus storage is limited for long-term data.
- **Usage:** Prometheus captures Kafka metrics; Grafana visualizes them in dashboards to identify architectural patterns and potential issues like broker imbalance.

3-ELK Stack (Elasticsearch, Logstash, Kibana)

- **Analysis Type:** In-depth log analysis for detecting errors and architectural patterns.
- **Pros:** Comprehensive log aggregation and visualization, useful for identifying architectural flaws.
- **Cons:** Resource-intensive and may require tuning for high log volumes.
- **Usage:** Logstash aggregates Kafka logs, and Kibana visualizes log data to identify common issues like frequent rebalancing.

4-Confluent Control Center

- **Analysis Type:** Kafka-specific metrics tracking, especially for consumer lag and partition health.
- **Pros:** Centralized view of Kafka-specific metrics and alerts.
- **Cons:** Requires Confluent licensing for full features.
- **Usage:** Monitors Kafka's consumer lag and broker load, highlighting inefficiencies in partitioning and data flow.

5-Jaeger for Tracing

- **Analysis Type:** Distributed tracing to visualize message flow and latency across Kafka and connected services.
- **Pros:** Provides visibility into Kafka's role in microservice architectures.
- **Cons:** Setup can be complex, and tracing can generate additional data overhead.
- **Usage:** Tracks message propagation, helping to identify latency and bottlenecks in Kafka's integration with other systems.

TASK II: Documented the Prescriptive (Intended) Architecture

Architectural Styles Used in Kafka's Evolution

Apache Kafka was initially developed by LinkedIn, and has evolved to meet the demands of scalable, distributed data streaming. Kafka's architecture transitioned from a client-server model with ZooKeeper to event-driven, publish-subscribe, and microservices-oriented styles, enabling high-performance, real-time applications.

1. Client-Server Architecture (Initial Phase)

Kafka's early architecture closely resembled a client-server model. Brokers acted as servers handling requests from client producers and consumers, while ZooKeeper managed metadata and coordinated broker state. This centralized management by ZooKeeper allowed Kafka to distribute client requests across brokers, handling partition leadership and consumer offsets. However, as clusters scaled, the reliance on ZooKeeper introduced performance bottlenecks and operational complexities.

- **Role of ZooKeeper:** Managed metadata, including partition leadership and cluster state.
- **Client-Broker Interaction:** Producers and consumers communicate with brokers for reading and writing messages, while brokers relied on ZooKeeper for metadata

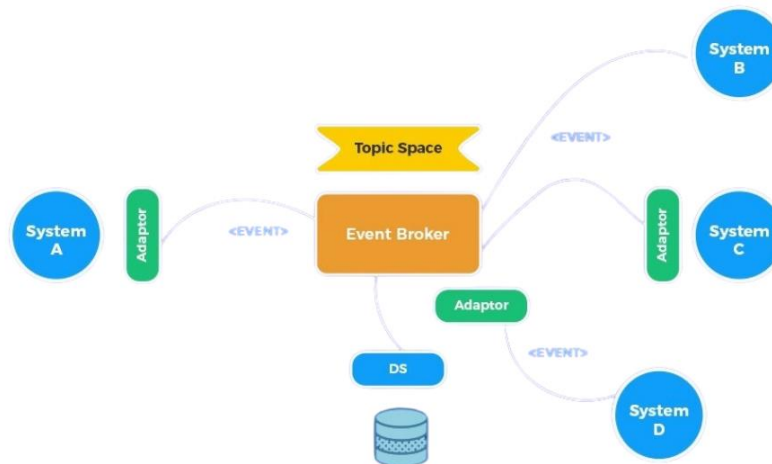
2. Event-Driven Architecture

As Kafka's demand for real-time data streaming grew, it adopted an **event-driven architecture**. In this model, Kafka brokers and clients operate asynchronously, emitting and consuming events independently via a distributed log. The event-driven style allows Kafka to support high-throughput, concurrent data processing, where brokers communicate state changes through event streams rather than relying solely on centralized control.

- **Event-Driven Processing:** Producers publish events to topics, which consumers can independently subscribe to and process at their own pace.

- **Scalability:** This architecture enables Kafka to handle massive event streams in parallel, ensuring low latency and high reliability for real-time applications.

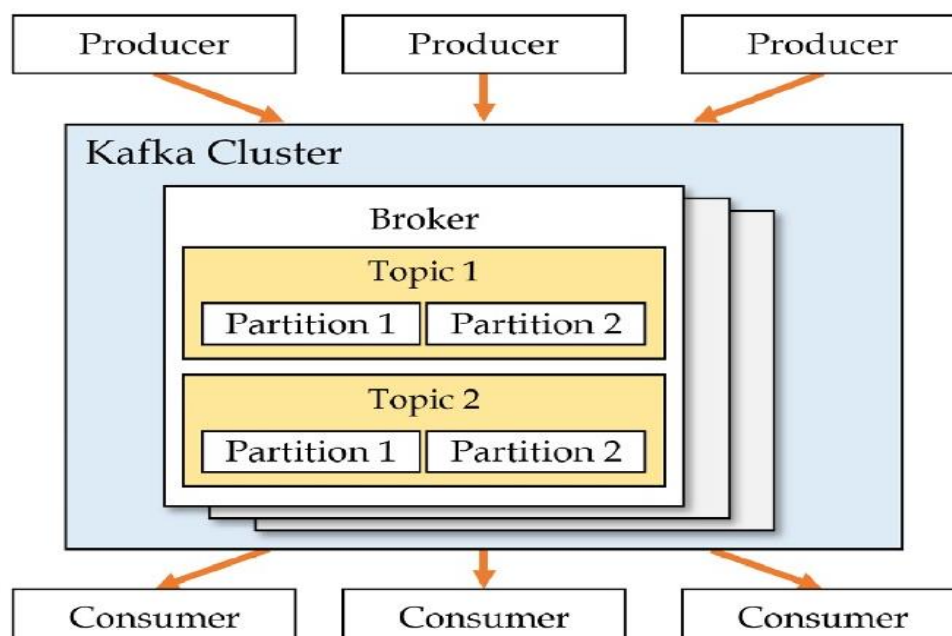
3.



Publish-Subscribe Model

Kafka's topic-based structure supports a **publish-subscribe** model, where multiple consumers can subscribe to a single topic, allowing for parallel data processing and enhanced scalability. This style enables Kafka to efficiently disseminate data from a single producer to multiple consumers without duplicating messages, optimizing resource use and supporting large-scale data distribution.

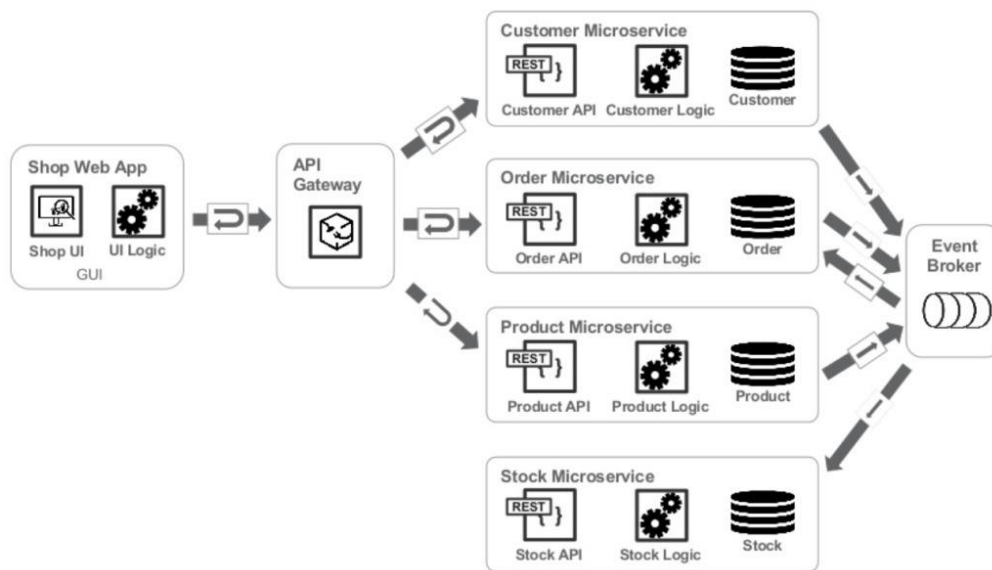
- **Topic and Partitioning:** Topics are divided into partitions, with each consumer in a group consuming from specific partitions.
- **Low Coupling:** Producers and consumers are decoupled, allowing each to operate independently and enhancing Kafka's reliability in distributed systems



4. Microservices-Oriented Architecture

Kafka's integration with **microservices** marks an evolution towards modular, flexible data pipelines where services communicate asynchronously through Kafka topics. This microservices approach is ideal for modern, cloud-native applications that require real-time data flows across distributed services.

- **Asynchronous Communication:** Microservices use Kafka topics to exchange data, allowing each service to process information independently without direct coupling.
- **Resilience and Scalability:** Kafka's distributed log structure ensures fault tolerance and allows microservices to scale independently, a crucial feature for dynamic and resilient architectures.



Design Patterns

Apache Kafka's architecture is a sophisticated assembly of several clear-cut patterns that drive its robustness, scalability, and efficiency in data streaming and real-time analytics. Here, we focus on the primary architectural patterns explicitly embedded in Kafka's core infrastructure: the **Publish-Subscribe Pattern**, **Messaging Bridge**, **Event-Monitor**, and **Load Balancing**.

1. Publish-Subscribe Pattern

Overview: The publish-subscribe pattern is fundamental in Kafka's design. This pattern facilitates communication between data producers and consumers through topics, ensuring multiple consumers can access and process the same data in parallel without affecting each other. In Kafka, topics function as channels where producers publish messages and consumers subscribe to receive them.

Implementation in Kafka: Kafka topics are divided into partitions, allowing each consumer in a group to handle messages from specific partitions. This structure enables Kafka to support high-throughput data processing, with messages available to all subscribed consumers, creating a decoupled data flow between producers and consumers.

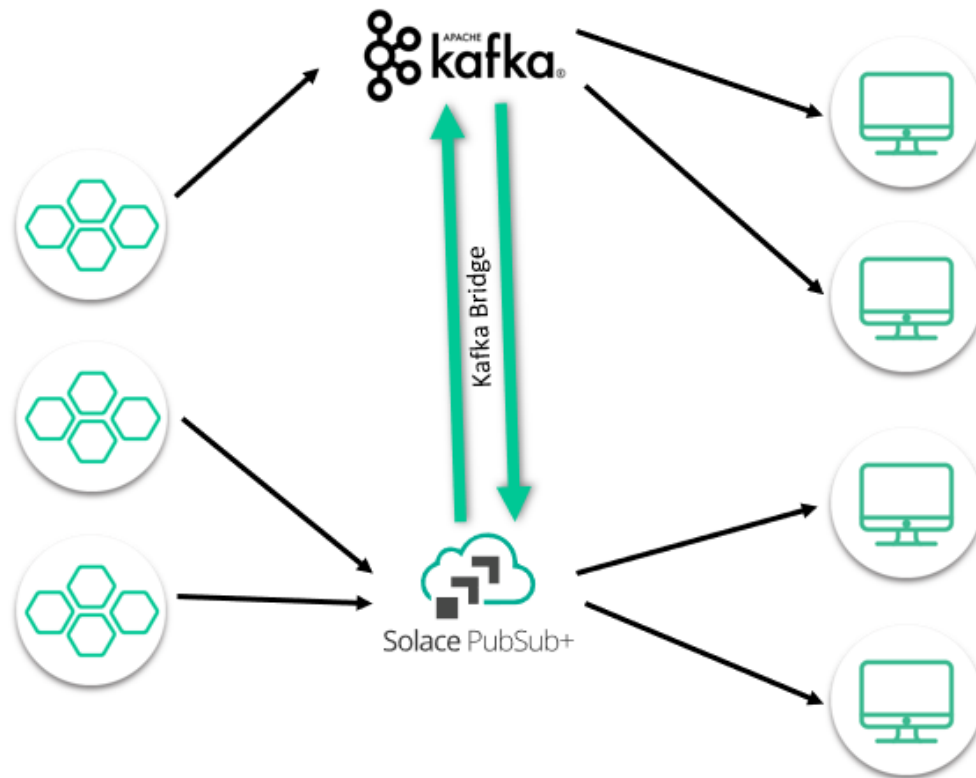
Impact: The publish-subscribe pattern in Kafka enables asynchronous communication, resilience, and scalability, making it ideal for large-scale, real-time data processing systems like IoT monitoring and user activity tracking on platforms like e-commerce or streaming services

2. Messaging Bridge Pattern

Overview: Kafka's **Kafka Connect** framework embodies the messaging bridge pattern by serving as a conduit between Kafka and external data sources or sinks. This allows data to move fluidly between Kafka and other storage or processing systems (such as databases, data lakes, or cloud storage).

Implementation in Kafka: Kafka Connect standardizes the integration of Kafka with other systems. Source connectors pull data into Kafka topics from external sources, while sink connectors push data from Kafka topics into destination systems. This decoupling and bridging mechanism is crucial for data integration, particularly in environments requiring data synchronization across disparate systems.

Impact: The messaging bridge pattern enables Kafka to integrate seamlessly with various data ecosystems, facilitating unified data flows and simplifying data management in distributed architectures. This approach is essential for organizations seeking to consolidate data from legacy and modern systems into Kafka for real-time processing



3. Load Balancing Pattern

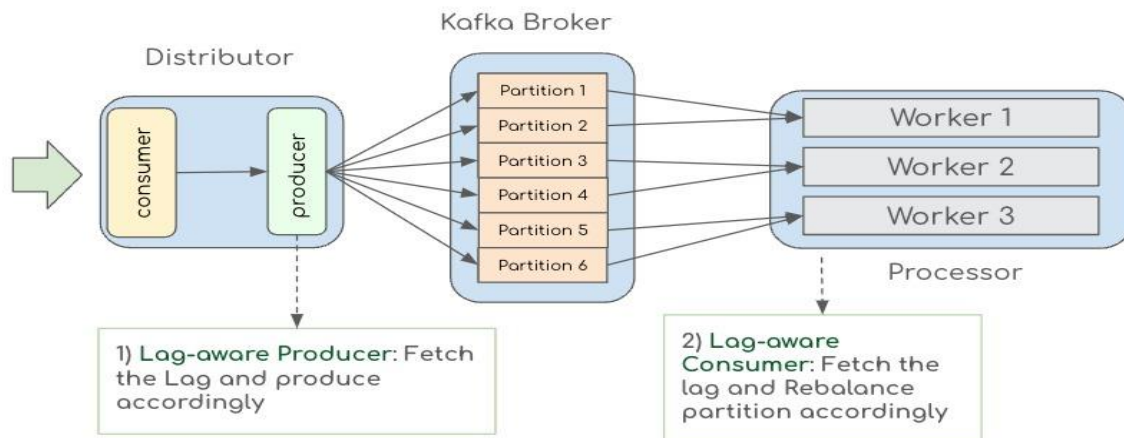
Overview: Kafka's load balancing is achieved through its partitioning strategy within topics. By distributing messages across multiple partitions and balancing these partitions across consumer instances, Kafka ensures an even workload distribution and prevents individual consumers from becoming overwhelmed.

Implementation in Kafka: Kafka topics are segmented into partitions, which allows data to be spread across brokers and read in parallel by consumer groups. When a new consumer joins or leaves a group, Kafka automatically reassigns partitions to balance the workload dynamically.

Impact: This load balancing approach enables Kafka to manage high volumes of data efficiently, with increased fault tolerance and reliability, as data is spread and managed across

multiple consumers. It is particularly beneficial in scenarios where scalable, real-time data processing is essential, such as financial market data feeds and content recommendation engines

4. Event-Monitor Pattern



Overview: Kafka's architecture functions as an event monitor, continuously capturing, recording, and processing data events in real time. This pattern is a cornerstone in Kafka, which acts as an event hub, where messages (events) are stored in topics for consumption by various services.

Implementation in Kafka: Every action in Kafka generates an event that is stored persistently in a distributed log. Consumers act as event monitors, polling Kafka topics for new data and processing it based on application requirements. This constant event monitoring is vital for maintaining data flow continuity in real-time applications.

Impact: With this event-monitoring model, Kafka provides reliable, continuous data feeds crucial for applications that depend on uninterrupted data streams, such as real-time analytics, fraud detection, and sensor data processing

Overview of Kafka:

Kafka is designed as a messaging-based log aggregator system that handles high-throughput, fault-tolerant, and distributed data streams.

Key Concepts:

1. Topics:

- A topic is a stream of messages of a particular type. Messages within the same topic are part of the same stream. Producers publish messages to topics, and consumers subscribe to them to pull out the data.

2. Producers:

- A producer writes a message to a Kafka topic. The producer can also collect a number of messages into a single batch to better leverage performance.
- Kafka is flexible in that it allows producers to decide upon their own serialization to use when encoding messages.

3. Brokers:

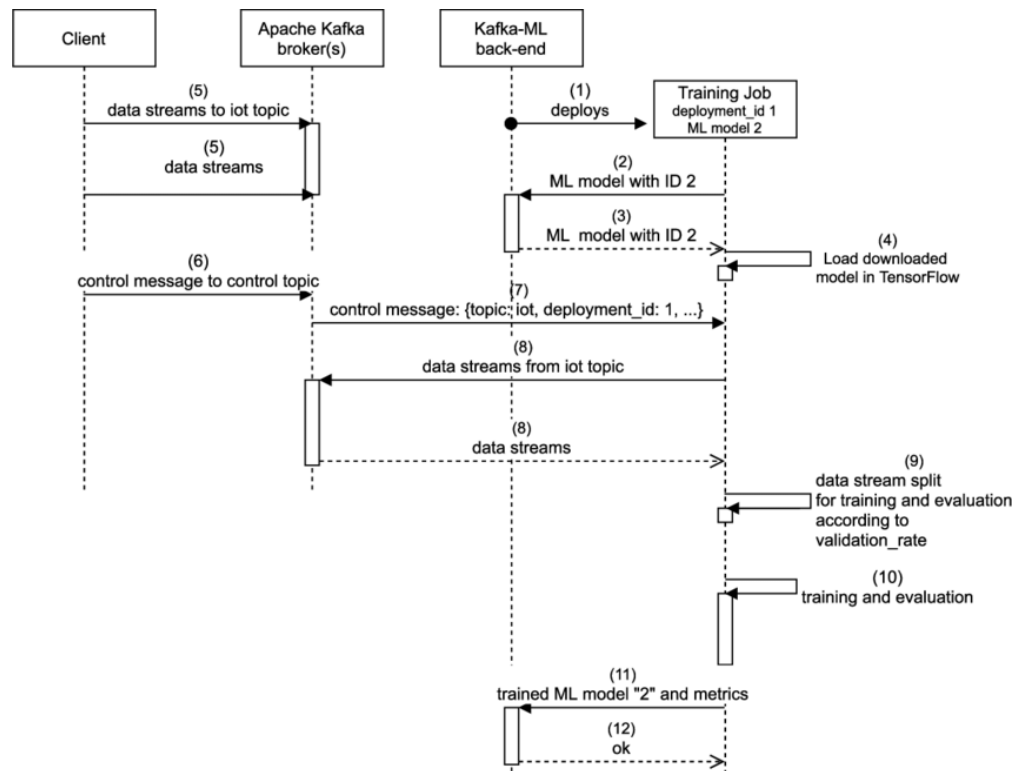
- Kafka uses brokers, which are servers responsible for the storage of published messages.
- Kafka clusters are distributed; in other words, they consist of several brokers. Every broker is going to handle a partition of a topic.
- Kafka divides a subject into partitions in order to distribute the load; brokers store these partitions. Messages in every partition get an identifier with the purpose of tracking the order of messages.

4. Consumers:

- A consumer subscribes to one or more topics, pulls data from brokers, and then processes it. Kafka supports two consumption models:
- Point-to-point: It allows several consumers to cooperate that consume a single copy of all messages.
- Publish/subscribe: Each consumer receives its own copy of the messages from the topic.
- A consumer iterates over the messages in a message stream, processing the payloads. By convention, Kafka's message streams are continuous; that is, they block if no new messages are available and wait until more are published.

Communication:

In Apache Kafka, **producers** publish messages to specific **topics**, which are divided into **partitions** across **brokers** to balance the load and support parallel processing. **Brokers** store these messages with unique offsets to maintain order, enabling efficient data retrieval. **Consumers** subscribe to topics and pull data independently, supporting both point-to-point (shared messages) and publish/subscribe (individual message copies) models. This architecture allows asynchronous communication between producers and consumers, ensuring high throughput, scalability, and fault tolerance in distributed data streaming.



Performance:

1. Efficiency in Data Storage and Transfer:

Simple Storage Layout: Kafka utilizes a straightforward log-based storage mechanism where each partition corresponds to a logical log implemented as segment files. This minimizes complexity and allows for efficient appending of messages.

Optimized Data Transfer: Kafka can handle multiple messages in a single pull request, improving throughput. It avoids double buffering by leveraging the underlying file system's page cache, leading to reduced memory overhead and improved performance.

2. Consumer Control and Flexibility:

Stateless Broker Design: By offloading the responsibility of tracking consumer offsets to the consumers themselves, Kafka simplifies the broker's design and reduces overhead. This allows consumers to manage their consumption states flexibly.

Rewind Capability: Consumers can rewind to older offsets to re-consume messages, a feature that, while atypical for queues, provides essential functionality for many use cases.

3. Retention and Message Management:

Kafka implements a time-based retention policy to manage message deletion, which addresses the challenge of ensuring messages are only deleted after all consumers have processed them. Messages are automatically deleted after a specified period, typically seven days, which balances storage efficiency with consumer needs.

4. Sequential Access and Cache Efficiency:

Both producers and consumers access segment files sequentially, which aligns well with operating system caching strategies. This design choice enhances performance through effective use of caching mechanisms, such as write-through caching and readahead.

Key Features of Apache Kafka Architecture:

- **Scalability:** Kafka's distributed and partitioned design allows it to scale horizontally, handling large data streams with high availability, ideal for real-time applications.
- **Fault Tolerance:** Leader-follower replication ensures data redundancy and continuity, allowing uninterrupted operations even if a broker fails.
- **High Throughput:** Log-based storage and zero-copy optimization enable fast read/write speeds, supporting high-volume tasks like analytics.
- **Efficient Data Handling:** Kafka's pull-based model allows consumers to process data at their own pace and reprocess as needed, ensuring balanced data flow.
- **Low Latency:** Zero-copy data transfer reduces delay, making Kafka suitable for latency-sensitive applications like fraud detection and IoT.

TASK-III Analyze the Descriptive Architecture of Apache Kafka

1. Introduction

In this report, the source code of Kafka has been examined, and its architectural structure has been analyzed using tools such as NDepend and SonarQube. Metrics such as code complexity, dependency management, technical debt, and security have been evaluated to identify strengths and weaknesses, and architectural improvement suggestions have been provided.

2. Architectural Styles and Patterns

2.1. Event-Driven Architecture

In fact, by looking deep inside the core of this application, into its source code, it would appear that in the system's message flow, the Kafka Broker class takes center stage, virtually a message bus. If our observations are correct, it manages a number of EndPoints that are defined by the names of listeners and corresponding security protocols. Such probably serves the purpose of supporting transparent communication between producers and consumers and other brokers. The fact that the Broker maintains connections, routes traffic, and keeps metadata at the cluster level would indicate that, in the distributed architecture of Kafka, it acts as a hub, which enables reliable and efficient message exchange. Based on this, we think Kafka uses an Event-Driven Architecture in its infrastructure.

The source code provided below emphasizes methods like `getNode()` and `brokerEndPoint()`, which seem to return node-level information and broker endpoints, respectively, given listener names. This modular approach might allow Kafka to retain flexibility in how communication paths are defined and managed. For example, explicit error handling in methods like `getNode(listenerName)` throws `BrokerEndPointNotAvailableException` in case of an endpoint not found, which indicates a focus on ensuring the reliability of communication and precise fault detection. We think this reinforces Kafka's robustness in managing distributed systems.

Furthermore, the presence of multiple constructors and method overloads to handle `listenerName`, `protocol`, and other parameters shows that it has been deliberately designed for adaptability. This may allow Kafka to support a variety of configurations and security protocols without needing major changes to the core Broker logic. In that case, the main Broker class would have an implementation design with modular logic and elaborate exception handling mechanisms that ensure

Kafka dynamically self-adjusts for runtime conditions at any moment with guaranteed reliability across message flows within the cluster.

```
case class Broker(id: Int, endpoints: Seq[EndPoint], rack: Option[String], features: Features[SupportedVersionRange]) {

  private val endpointsMap = endpoints.map { endPoint =>
    endPoint.listenerName -> endPoint
  }.toMap

  if (endpointsMap.size != endpoints.size)
    throw new IllegalArgumentException(s"There is more than one end point with the same listener name: ${endpoints.mkString(",")}")

  override def toString: String =
    s"$id : ${endpointsMap.values.mkString("(", ",", ")")} : ${rack.orNull} : $features"

  def this(id: Int, host: String, port: Int, listenerName: ListenerName, protocol: SecurityProtocol) = {
    this(id, Seq(EndPoint(host, port, listenerName, protocol)), None, emptySupportedFeatures)
  }

  def this(bep: BrokerEndPoint, listenerName: ListenerName, protocol: SecurityProtocol) = {
    this(bep.id, bep.host, bep.port, listenerName, protocol)
  }

  def node(listenerName: ListenerName): Node =
    getNode(listenerName).getOrElse {
      throw new BrokerEndPointNotAvailableException(s"End point with listener name ${listenerName.value} not found " +
        s"for broker $id")
    }
  }

  def getNode(listenerName: ListenerName): Option[Node] =
    endpointsMap.get(listenerName).map(endpoint => new Node(id, endpoint.host, endpoint.port, rack.orNull))

  def brokerEndPoint(listenerName: ListenerName): BrokerEndPoint = {
    val endpoint = endPoint(listenerName)
    new BrokerEndPoint(id, endpoint.host, endpoint.port)
  }

  def endPoint(listenerName: ListenerName): EndPoint = {
    endpointsMap.getOrElse(listenerName,
      throw new BrokerEndPointNotAvailableException(s"End point with listener name ${listenerName.value} not found for broker $id"))
  }

  def toServerInfo(clusterId: String, config: KafkaConfig): AuthorizerServerInfo = {
    val clusterResource: ClusterResource = new ClusterResource(clusterId)
    val interBrokerEndpoint: Endpoint = endPoint(config.interBrokerListenerName).toJava
    val brokerEndpoints: util.List[EndPoint] = endpoints.toList.map(_.toJava).asJava
    Broker.ServerInfo(clusterResource, id, brokerEndpoints, interBrokerEndpoint,
      config.earlyStartListeners.map(_.value()).asJava)
  }
}
```

Screenshot 1 Broker Class From Github Kafka Source Code

2.2. Publish-Subscribe Model

While we have been inspecting the source code of Kafka, we stumbled upon the SubscriptionPattern class. The class name and the manner in which it is structured seem to add credence to the argument whereby Kafka uses a publish-subscribe architecture. It shows that this class is defined in a way where some subscription patterns are created with the use of regular expressions, and that probably allows filtering of messages as well as dynamic subscription. Such functionalities might come handy in letting consumers subscribe to topics flexibly, that is, consistent with the other properties of Kafka as a decoupled messaging system.

SubscriptionPattern seems to represent a design that would allow consumers to subscribe based on patterns instead of one specific topic, which reflects the principles of the publish-subscribe model very much. Probably, it will facilitate supporting dynamic workloads and heterogeneous use cases. From our review, it looks like SubscriptionPattern plays an important role in what would be Kafka's flexible and scalable messaging infrastructure.

```
public class SubscriptionPattern {

    /**
     * String representation the regular expression, compatible with RE2/J.
     */
    private final String pattern;

    public SubscriptionPattern(String pattern) {
        this.pattern = pattern;
    }

    /**
     * @return Regular expression pattern compatible with RE2/J.
     */
    public String pattern() {
        return this.pattern;
    }

    @Override
    public String toString() {
        return pattern;
    }

    @Override
    public int hashCode() {
        return pattern.hashCode();
    }

    @Override
    public boolean equals(Object obj) {
        return obj instanceof SubscriptionPattern &&
            Objects.equals(pattern, ((SubscriptionPattern) obj).pattern);
    }
}
```

Screenshot 2 SubscriptionPattern Class From Github Kafka Source Code

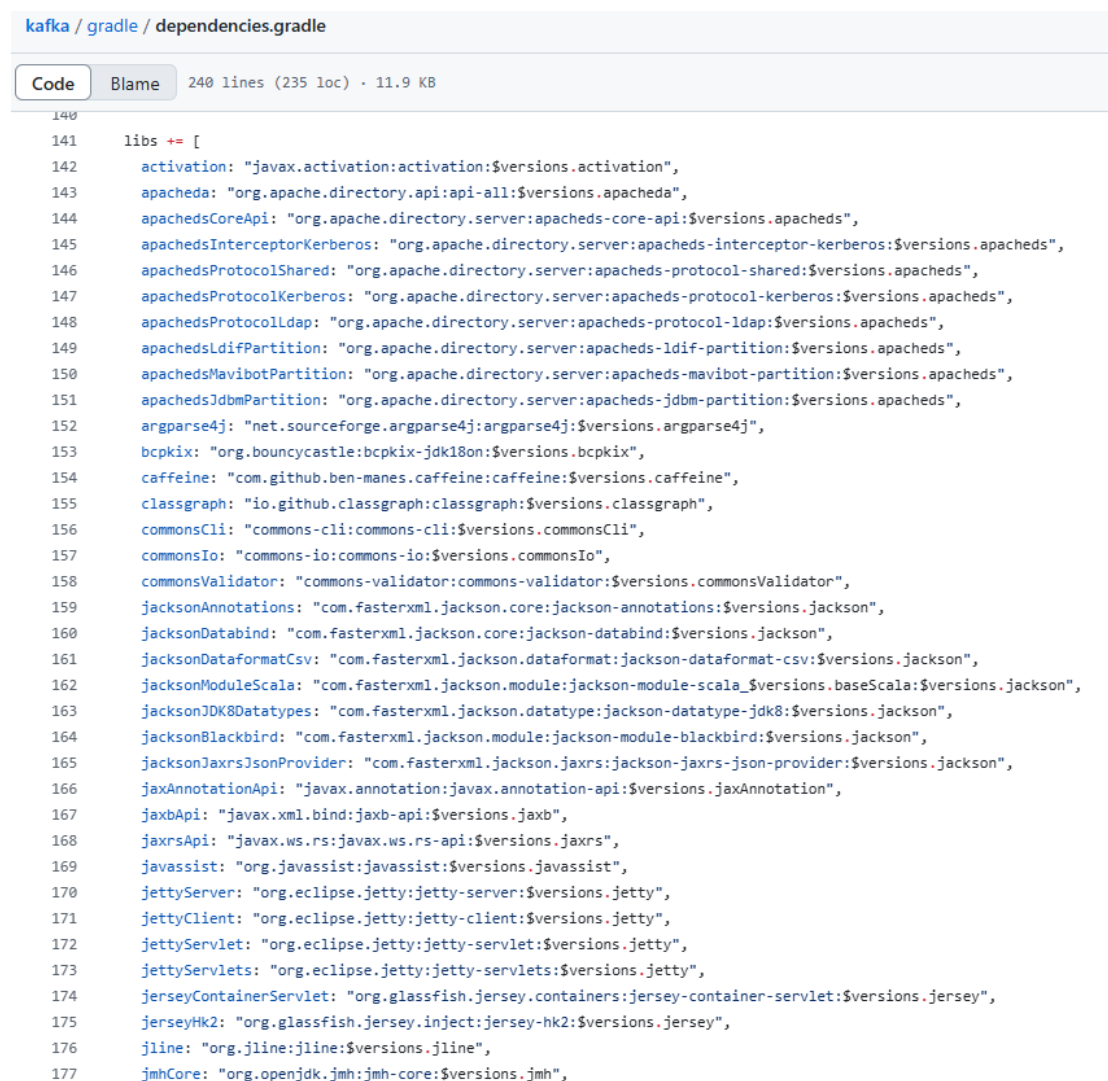
2.3. Integration of Third-Party Dependencies

The Kafka codebase has explicitly integrated several third-party dependencies to extend its capabilities and make development easier. Judging by the Gradle configuration, these dependencies appear to play a critical role in varied aspects of the Kafka distributed system.

Incorporated among other things is Apache Directory Server, a.k.a. ApacheDS: org.apache.directory.server:apacheds-core-api. This can perform directory operations like LDAP authentication and the like. Similarly, Jackson- com.fasterxml.jackson.core:jackson-annotations-it

probably handles all JSON-related processes, heavily leveraged during serialization and deserialization of data in Kafka's data pipelines. These integrations would probably provide flexibility and efficient handling of data structure-unstructured data.

In addition, Netty indicates a significant part in high-throughput, low-latency networking I/O, which is a significant building block for resolving high-performance and distributed messaging processing from Kafka. Other dependencies include Google Protocol Buffers, or Protobuf, at `com.google.protobuf:protobuf-java`, which may further contribute to the efficient serialization of structured data into Kafka Streams or some internal protocols. Kafka uses Mockito from `org.mockito:mockito-core` for testing, which simplifies unit testing by allowing dependency mocking. What seems feasible, then, is a third-party tool that augments the scalability, reliability, and maintainability of Kafka, consequently allowing it to handle complex distributed systems with high ease.



```
kafka / gradle / dependencies.gradle

Code Blame 240 lines (235 loc) · 11.9 KB

140
141     libs += [
142         activation: "javax.activation:activation:$versions.activation",
143         apacheda: "org.apache.directory.api:api-all:$versions.apacheda",
144         apachedsCoreApi: "org.apache.directory.server:apacheds-core-api:$versions.apacheds",
145         apachedsInterceptorKerberos: "org.apache.directory.server:apacheds-interceptor-kerberos:$versions.apacheds",
146         apachedsProtocolShared: "org.apache.directory.server:apacheds-protocol-shared:$versions.apacheds",
147         apachedsProtocolKerberos: "org.apache.directory.server:apacheds-protocol-kerberos:$versions.apacheds",
148         apachedsProtocolLdap: "org.apache.directory.server:apacheds-protocol-ldap:$versions.apacheds",
149         apachedsLdifPartition: "org.apache.directory.server:apacheds-ldif-partition:$versions.apacheds",
150         apachedsMavibotPartition: "org.apache.directory.server:apacheds-mavibot-partition:$versions.apacheds",
151         apachedsJdbmPartition: "org.apache.directory.server:apacheds-jdbm-partition:$versions.apacheds",
152         argparse4j: "net.sourceforge.argparse4j:argparse4j:$versions.argparse4j",
153         bcpkix: "org.bouncycastle:bcpkix-jdk18on:$versions.bcpkix",
154         caffeine: "com.github.ben-manes.caffeine:caffeine:$versions.caffeine",
155         classgraph: "io.github.classgraph:classgraph:$versions.classgraph",
156         commonsCli: "commons-cli:commons-cli:$versions.commonsCli",
157         commonsIo: "commons-io:commons-io:$versions.commonsIo",
158         commonsValidator: "commons-validator:commons-validator:$versions.commonsValidator",
159         jacksonAnnotations: "com.fasterxml.jackson.core:jackson-annotations:$versions.jackson",
160         jacksonDatabind: "com.fasterxml.jackson.core:jackson-databind:$versions.jackson",
161         jacksonDataformatCsv: "com.fasterxml.jackson.dataformat:jackson-dataformat-csv:$versions.jackson",
162         jacksonModuleScala: "com.fasterxml.jackson.module:jackson-module-scala_$versions.baseScala:$versions.jackson",
163         jacksonJDK8Datatypes: "com.fasterxml.jackson.datatype:jackson-datatype-jdk8:$versions.jackson",
164         jacksonBlackbird: "com.fasterxml.jackson.module:jackson-module-blackbird:$versions.jackson",
165         jacksonJaxrsJsonProvider: "com.fasterxml.jackson.jaxrs:jackson-jaxrs-json-provider:$versions.jackson",
166         jaxAnnotationApi: "javax.annotation:javax.annotation-api:$versions.jaxAnnotation",
167         jaxbApi: "javax.xml.bind:jaxb-api:$versions.jaxb",
168         jaxrsApi: "javax.ws.rs:javax.ws.rs-api:$versions.jaxrs",
169         javassist: "org.javassist:javassist:$versions.javassist",
170         jettyServer: "org.eclipse.jetty:jetty-server:$versions.jetty",
171         jettyClient: "org.eclipse.jetty:jetty-client:$versions.jetty",
172         jettyServlet: "org.eclipse.jetty:jetty-servlet:$versions.jetty",
173         jettyServlets: "org.eclipse.jetty:jetty-servlets:$versions.jetty",
174         jerseyContainerServlet: "org.glassfish.jersey.containers:jersey-container-servlet:$versions.jersey",
175         jerseyHk2: "org.glassfish.jersey.inject:jersey-hk2:$versions.jersey",
176         jline: "org.jline:jline:$versions.jline",
177         jmhCore: "org.openjdk.jmh:jmh-core:$versions.jmh",
```

Screenshot 3 Dependencies From Github Kafka Source Code

2.4. Pattern Determinations:

The provided `PipeDemo` class demonstrates the **Pipe-and-Filter** architectural design pattern. This pattern is widely used in stream processing systems such as Apache Kafka Streams.

```
public class PipeDemo {

    public static void main(final String[] args) {
        final Properties props = new Properties();
        props.put(StreamsConfig.APPLICATION_ID_CONFIG, "streams-pipe");
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.StringSerde.class);
        props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.StringSerde.class);

        // setting offset reset to earliest so that we can re-run the demo code with the same pre-loaded data
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

        final StreamsBuilder builder = new StreamsBuilder();

        builder.stream("streams-plaintext-input").to("streams-pipe-output");

        final KafkaStreams streams = new KafkaStreams(builder.build(), props);
        final CountDownLatch latch = new CountDownLatch(1);

        // attach shutdown handler to catch control-c
        Runtime.getRuntime().addShutdownHook(new Thread("streams-pipe-shutdown-hook") {
            @Override
            public void run() {
                streams.close();
                latch.countDown();
            }
        });

        try {
            streams.start();
            latch.await();
        } catch (final Throwable e) {
            System.exit(1);
        }
        System.exit(0);
    }
}
```

Screenshot 4 PipeDemo Class From Github Kafka Source Code

The provided `UnsubscribeEvent` class demonstrates the use of the **Publish-Subscribe pattern**, a design pattern where senders (publishers) and receivers (subscribers) are decoupled. This pattern allows for efficient and scalable communication within an event-driven architecture.

```
public class UnsubscribeEvent extends CompletableApplicationEvent<Void> {

    public UnsubscribeEvent(final long deadlineMs) {
        super(Type.UNSUBSCRIBE, deadlineMs);
    }
}
```

Screenshot 5 UnsubscribeEvent Class From Github Kafka Source Code

The Metrics class looks likely to be implementing the Singleton Pattern since its design throughout appears to be a centralized repository for metrics throughout a system. Its designs through concurrent maps like metrics and sensors probably imply that this class highly experiments with single-instance management and coordinative efforts to keep information collection standardized across

threads. The use of a shared `MetricConfig` and a `ScheduledThreadPoolExecutor` further seems to validate that theory, which we think will add thread safety while preventing redundancy.

Also, reports and childrenSensors' inclusion in the lists suggests that it is a metric reporter with an aggregated collection of such reporters in a possible hierarchy of sensors. This design should enable efficient metric collection and reporting across the system. From its implementation, we believe that the `Metrics` class is primarily for tracking performance and system health: consistent with Singleton in the architecture of Kafka that ensures standard behavior.

```
70  public final class Metrics implements Closeable {
71
72      private final MetricConfig config;
73      private final ConcurrentMap<MetricName, KafkaMetric> metrics;
74      private final ConcurrentMap<String, Sensor> sensors;
75      private final ConcurrentMap<Sensor, List<Sensor>> childrenSensors;
76      private final List<MetricsReporter> reporters;
77      private final Time time;
78      private final ScheduledThreadPoolExecutor metricsScheduler;
79      private static final Logger log = LoggerFactory.getLogger(Metrics.class);
80
81      /**
82       * Create a metrics repository with no metric reporters and default configuration.
83       * Expiration of Sensors is disabled.
84       */
85      public Metrics() {
86          this(new MetricConfig());
87      }
```

Screenshot 6 Singleton Pattern From Github Kafka Source Code

The `forId` method probably shows a Factory Pattern in that it dynamically returns to the caller the `CompressionType` instance related to the provided ID. As far as we have observed, this probably wraps the logic necessary to choose the right compression strategy with some modularity and flexibility of the code. This is rather done by doing a switch over integer ids for compression types like GZIP, SNAPPY, and ZSTD, thus centralizing the method for deciding this rather than embedding it elsewhere all over the code.

This is complemented with the default case to handle an exception for unknown IDs, which implies a focus on robustness and error handling. This we assume enables Kafka to acquire new compression formats easily. It would simply add logic to `forId` and not alter existing code. This modular approach appears to go hand-in-hand with the Factory Pattern principles for scalable maintainable architecture with Kafka.

```

144  public static CompressionType forId(int id) {
145      switch (id) {
146          case 0:
147              return NONE;
148          case 1:
149              return GZIP;
150          case 2:
151              return SNAPPY;
152          case 3:
153              return LZ4;
154          case 4:
155              return ZSTD;
156          default:
157              throw new IllegalArgumentException("Unknown compression type id: " + id);
158      }
159  }

```

Screenshot 7 Factory Pattern From Github Kafka Source Code

The ConsumerCoordinatorMetrics class, by checking for run-time changes like commit latency, would seem to present behaviour resembling the Observer Pattern, feeding updates into their respective metrics dynamically. The event-inferring metrics sensed using commitSensor for tracking average and maximum commit latency continue to suggest that this class responds to and collects performance data as real-time performance events happen.

This might be the design to monitor and react dynamically to system performance without letting go of valuable runtime information that can later be captured and stored for analysis in the future. The additional metrics of the number of assigned partitions add weight to the observation of such a class as well as its activity about consumers. We suppose such an approach is consistent not only with the principles of the Observer Pattern, but also with enhancing the observable behaviour of Kafka in offering possible actions to be taken based on its operational performance for real-time monitoring and debugging.

```

1558  private class ConsumerCoordinatorMetrics {
1559      private final Sensor commitSensor;
1560
1561  private ConsumerCoordinatorMetrics(Metrics metrics, String metricGrpPrefix) {
1562      String metricGrpName = metricGrpPrefix + COORDINATOR_METRICS_SUFFIX;
1563
1564      this.commitSensor = metrics.sensor("commit-latency");
1565      this.commitSensor.add(metrics.metricName("commit-latency-avg",
1566          metricGrpName,
1567          "The average time taken for a commit request"), new Avg());
1568      this.commitSensor.add(metrics.metricName("commit-latency-max",
1569          metricGrpName,
1570          "The max time taken for a commit request"), new Max());
1571      this.commitSensor.add(createMeter(metrics, metricGrpName, "commit", "commit calls"));
1572
1573      Measurable numParts = (config, now) -> subscriptions.numAssignedPartitions();
1574      metrics.addMetric(metrics.metricName("assigned-partitions",
1575          metricGrpName,
1576          "The number of partitions currently assigned to this consumer"), numParts);
1577  }
1578  }

```


Screenshot 8 Observer Pattern From Github Kafka Source Code

StateStore interface seems to have state-related operations within it aligning to the Repository Pattern. According to our observation, this abstraction lets separate storage and state access logic from implementation details. Thus developers might focus on specific types of storage-in such cases-as RocksDB or in-memory storage. With such general interface provision, we can presume the simplification of state management across various stream processing tasks by StateStore.

Furthermore, the init method, which takes a StateStoreContext and a root StateStore object, also suggests quite a flexible scheme for initializing and restoring state from a changelog. It could allow implementations to deal with, for instance, bulk-load or restore callback commits outside of the core storage logic. All in all, we believe such an approach guarantees consistent state management across different stream processing operations with reduced complexity and better maintainability in Kafka's stream processing framework.

```
48 public interface StateStore {
49
50     /**
51      * The name of this store.
52      * @return the storage name
53      */
54     String name();
55
56     /**
57      * Initializes this state store.
58      * <p>
59      * The implementation of this function must register the root store in the stateStoreContext via the
60      * {@link StateStoreContext#register(StateStore, StateRestoreCallback, CommitCallback)} function, where the
61      * first {@link StateStore} parameter should always be the passed-in {@code root} object, and
62      * the second parameter should be an object of user's implementation
63      * of the {@link StateRestoreCallback} interface used for restoring the state store from the changelog.
64      * <p>
65      * Note that if the state store engine itself supports bulk writes, users can implement another
66      * interface {@link BatchingStateRestoreCallback} which extends {@link StateRestoreCallback} to
67      * let users implement bulk-load restoration logic instead of restoring one record at a time.
68      *
69      * @throws IllegalStateException If store gets registered after initialized is already finished
70      * @throws StreamsException if the store's change log does not contain the partition
71      */
72     void init(final StateStoreContext stateStoreContext, final StateStore root);
```

Screenshot 9 Repository Pattern From Github Kafka Source Code

2.5. Component Analysis:

It appears that the key components of Apache Kafka-from the source code analysis over GitHub-are core, clients, streams, and connect directories, which seem to constitute the foundation of its architecture. The goal of investigating these areas was to understand the structure and main functionalities of the system. The source code states that the above components probably work together to enable Kafka's distributed messaging and data streaming platform.

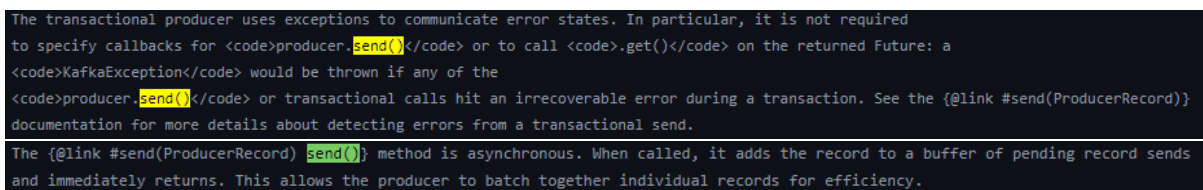
Thus, we could study the most important classes and configuration files and how they are related by exploring these directories. We were then trying to understand how all this is connected and what it adds to the overall functionality of Kafka. Some conclusions have been reached through direct observation of the code base, others are tentative, based on inferences about the intended design and

purpose of these elements. This way seems to form a picture of what might be Kafka's architectural framework and operational flow.

Through this exploring process, we believe we can identify what is the heart of Kafka which probably concerned with message production and consumption as well as stream processing and integration toward a different environment. These will not be conclusive but rather a peek into Kafka's possibilities for a scalable and reliable data streaming platform.

1. Producers

- **Files Reviewed:**
 - KafkaProducer.java

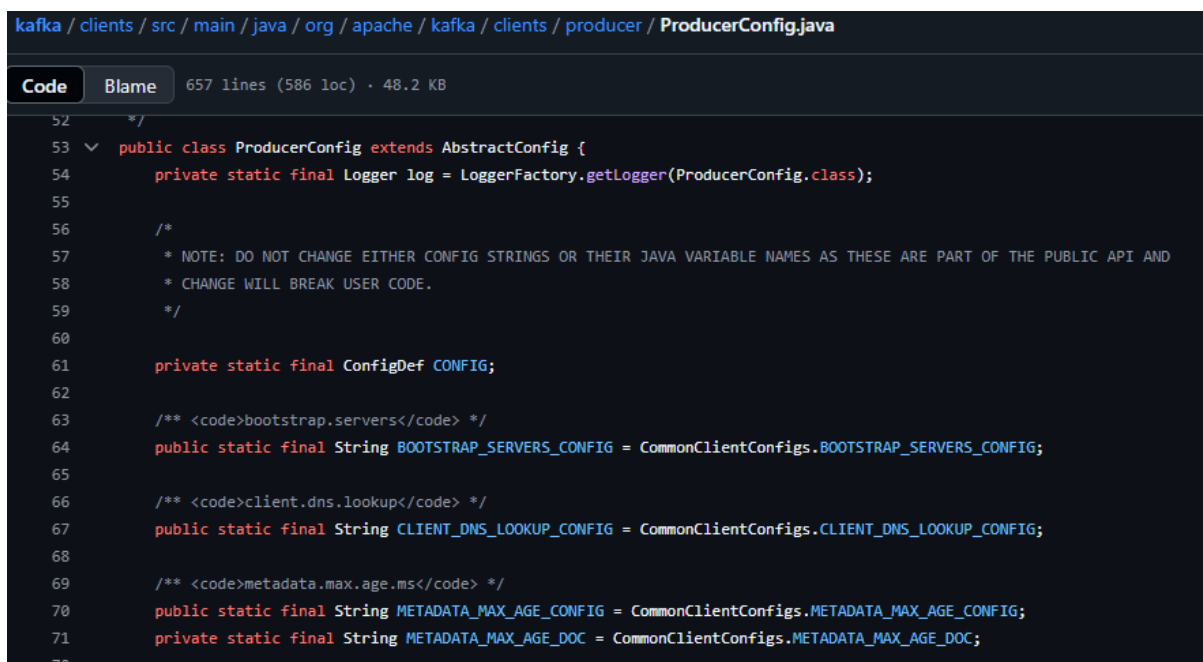


The transactional producer uses exceptions to communicate error states. In particular, it is not required to specify callbacks for `producer.send()` or to call `get()` on the returned Future: a `KafkaException` would be thrown if any of the `producer.send()` or transactional calls hit an irrecoverable error during a transaction. See the [{@link #send\(ProducerRecord\)}](#) documentation for more details about detecting errors from a transactional send.

The [{@link #send\(ProducerRecord\) send\(\)}](#) method is asynchronous. When called, it adds the record to a buffer of pending record sends and immediately returns. This allows the producer to batch together individual records for efficiency.

Screenshot 10 Developer Notes About KafkaProducer Class From Github Kafka Source Code

- ProducerConfig.java



kafka / clients / src / main / java / org / apache / kafka / clients / producer / **ProducerConfig.java**

Code Blame 657 lines (586 loc) · 48.2 KB

```
52  */
53  public class ProducerConfig extends AbstractConfig {
54      private static final Logger log = LoggerFactory.getLogger(ProducerConfig.class);
55
56      /*
57       * NOTE: DO NOT CHANGE EITHER CONFIG STRINGS OR THEIR JAVA VARIABLE NAMES AS THESE ARE PART OF THE PUBLIC API AND
58       * CHANGE WILL BREAK USER CODE.
59       */
60
61      private static final ConfigDef CONFIG;
62
63      /** <code>bootstrap.servers</code> */
64      public static final String BOOTSTRAP_SERVERS_CONFIG = CommonClientConfigs.BOOTSTRAP_SERVERS_CONFIG;
65
66      /** <code>client.dns.lookup</code> */
67      public static final String CLIENT_DNS_LOOKUP_CONFIG = CommonClientConfigs.CLIENT_DNS_LOOKUP_CONFIG;
68
69      /** <code>metadata.max.age.ms</code> */
70      public static final String METADATA_MAX_AGE_CONFIG = CommonClientConfigs.METADATA_MAX_AGE_CONFIG;
71      private static final String METADATA_MAX_AGE_DOC = CommonClientConfigs.METADATA_MAX_AGE_DOC;
72  }
```

Screenshot 11 ProducerConfig Class From Github Kafka Source Code

- **Observations:** These observations indicate that Kafka producer class should be central to the management of records production into Kafka topics. Send() function likely works for asynchronous transmission, immediately queuing records in a buffer we believe would batch multiple records for efficiency. It furthermore seems the transactional producer uses exceptions as a mechanism toward reliable delivery since these notify errors during send operations. This design could probably allow producers to fail-safe without requiring add-on callbacks.

This ProducerConfig class appears to be an important class in our example since it

should manage producer-specific properties, for example bootstrap servers, retry mechanism, and metadata refresh intervals. This means that all its configuration should allow producer setup simplified usage while also ensuring standardized configuration values among several producer instances. The annotations and constants in the class hint toward making configuration parameters extensible or amenable to change.

All of them seem to join together to make this a quite solid and flexible system for message production to Kafka topics. Well, `KafkaProducer` and `ProducerConfig` are expected to have a modular design that is efficient, reliable in message transmission, as it pertains to the scalability of a distributed architecture like Kafka.

2. Consumers

- **Files Reviewed:**
 - `KafkaConsumer.java`

```
After subscribing to a set of topics, the consumer will automatically join the group when {@link #poll(Duration)} is invoked. The poll API is designed to ensure consumer liveness. As long as you continue to call poll, the consumer will stay in the group and continue to receive messages from the partitions it was assigned. Underneath the covers, the consumer sends periodic heartbeats to the server. If the consumer crashes or is unable to send heartbeats for a duration of {@code session.timeout.ms}, then the consumer will be considered dead and its partitions will be reassigned.

@Override
public void commitAsync() {
    delegate.commitAsync();
}

/**
 * Commit offsets returned on the last {@link #poll(Duration) poll()} for the subscribed list of topics and partitions.
```

Screenshot 12 Developer Notes About `KafkaConsumer` Class From Github Kafka Source Code

- `ConsumerConfig.java`

```
/**
 * The consumer configuration keys
 */
public class ConsumerConfig extends AbstractConfig {
    private static final ConfigDef CONFIG;

    // a list contains all the assignor names that only assign subscribed topics to consumer. Should be updated when new assignor added.
    // This is to help optimize ConsumerCoordinator#performAssignment method
    public static final List<String> ASSIGN_FROM_SUBSCRIBED_ASSIGNORS = List.of(
        RANGE_ASSIGNOR_NAME,
        ROUNDROBIN_ASSIGNOR_NAME,
        STICKY_ASSIGNOR_NAME,
        COOPERATIVE_STICKY_ASSIGNOR_NAME
    );
};
```

Screenshot 13 Developer Notes About `ConsumerConfig` Class From Github Kafka Source Code

- **Observations:** It can be observed that the `KafkaConsumer` class plays a very important role in the message consuming process from Kafka topics. The `poll(Duration)` method most probably assures the consumer's liveness by making it synchronize with its group every time the messages are consumed with periodic heartbeats. This mechanism may help to maintain the integrity of the group without unnecessary reassignments. Also, it seems that methods like

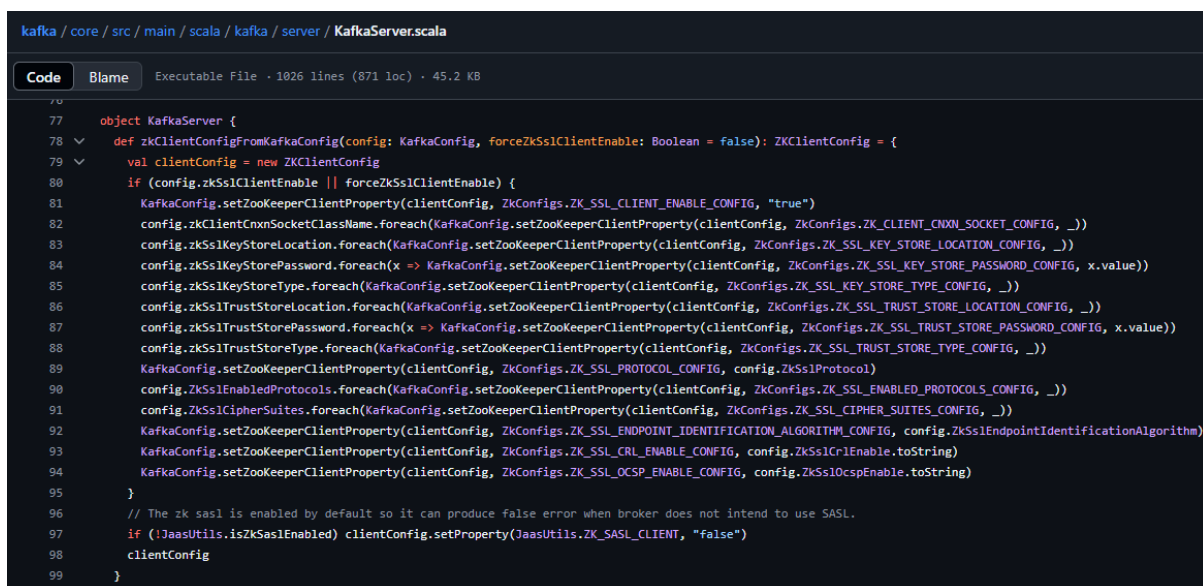
commitAsync() and commitSync() provide very flexible mechanisms for offset management, thus enabling consumers to either confirm the message as processed asynchronously or synchronously, depending upon the requirement.

Hence, the ConsumerConfig class, as we could make out from the context, would define the most critical configuration parameters, which would optimize the behavior of consumer groups. For example, the list of assignor names like RANGE_ASSIGNOR_NAME and ROUNDROBIN_ASSIGNOR_NAME indicates such random methods to assign partitions in consumers. Obviously, these would make Kafka more efficient than it already is in unevenly balancing assignments among consumer group members.

These components seem to ensure thus that Kafka's consumers are reliable but can also easily adapt to new requirements. This combination of dynamic strategies for assigning partitions and offset management tools stands out in KafkaConsumer and ConsumerConfig for a seamless yet scalable message consumption process within Kafka's distributed architecture.

3. Brokers

- **Files Reviewed:**
 - KafkaServer.scala



```
kafka / core / src / main / scala / kafka / server / KafkaServer.scala

Code Blame Executable File · 1026 lines (871 loc) · 45.2 KB

77 object KafkaServer {
78   def zkClientConfigFromKafkaConfig(config: KafkaConfig, forceZkSslClientEnable: Boolean = false): ZKClientConfig = {
79     val clientConfig = new ZKClientConfig
80     if (config.zkSslClientEnable || forceZkSslClientEnable) {
81       KafkaConfig.setZooKeeperClientProperty(clientConfig, ZkConfigs.ZK_SSL_CLIENT_ENABLE_CONFIG, "true")
82       config.zkClientCnxnSocketClassName.foreach(KafkaConfig.setZooKeeperClientProperty(clientConfig, ZkConfigs.ZK_CLIENT_CNXN_SOCKET_CONFIG, _))
83       config.zkSslKeyStoreLocation.foreach(KafkaConfig.setZooKeeperClientProperty(clientConfig, ZkConfigs.ZK_SSL_KEY_STORE_LOCATION_CONFIG, _))
84       config.zkSslKeyStorePassword.foreach(x => KafkaConfig.setZooKeeperClientProperty(clientConfig, ZkConfigs.ZK_SSL_KEY_STORE_PASSWORD_CONFIG, x.value))
85       config.zkSslKeyStoreType.foreach(KafkaConfig.setZooKeeperClientProperty(clientConfig, ZkConfigs.ZK_SSL_KEY_STORE_TYPE_CONFIG, _))
86       config.zkSslTrustStoreLocation.foreach(KafkaConfig.setZooKeeperClientProperty(clientConfig, ZkConfigs.ZK_SSL_TRUST_STORE_LOCATION_CONFIG, _))
87       config.zkSslTrustStorePassword.foreach(x => KafkaConfig.setZooKeeperClientProperty(clientConfig, ZkConfigs.ZK_SSL_TRUST_STORE_PASSWORD_CONFIG, x.value))
88       config.zkSslTrustStoreType.foreach(KafkaConfig.setZooKeeperClientProperty(clientConfig, ZkConfigs.ZK_SSL_TRUST_STORE_TYPE_CONFIG, _))
89       KafkaConfig.setZooKeeperClientProperty(clientConfig, ZkConfigs.ZK_SSL_PROTOCOL_CONFIG, config.ZkSslProtocol)
90       config.zkSslEnabledProtocols.foreach(KafkaConfig.setZooKeeperClientProperty(clientConfig, ZkConfigs.ZK_SSL_ENABLED_PROTOCOLS_CONFIG, _))
91       config.zkSslCipherSuites.foreach(KafkaConfig.setZooKeeperClientProperty(clientConfig, ZkConfigs.ZK_SSL_CIPHER_SUITES_CONFIG, _))
92       KafkaConfig.setZooKeeperClientProperty(clientConfig, ZkConfigs.ZK_SSL_ENDPOINT_IDENTIFICATION_ALGORITHM_CONFIG, config.ZkSslEndpointIdentificationAlgorithm)
93       KafkaConfig.setZooKeeperClientProperty(clientConfig, ZkConfigs.ZK_SSL_CRL_ENABLE_CONFIG, config.ZkSslCrlEnable.toString)
94       KafkaConfig.setZooKeeperClientProperty(clientConfig, ZkConfigs.ZK_SSL_OCSP_ENABLE_CONFIG, config.ZkSslOcspEnable.toString)
95     }
96     // The zk sasl is enabled by default so it can produce false error when broker does not intend to use SASL.
97     if (!JaasUtils.isZkSaslEnabled) clientConfig.setProperty(JaasUtils.ZK_SASL_CLIENT, "false")
98     clientConfig
99   }
100 }
```

Screenshot 14 KafkaServer Object From Github Kafka Source Code

- ReplicaManager.scala

```
kafka / core / src / main / scala / kafka / server / ReplicaManager.scala

Code Blame 3083 lines (2783 loc) · 157 KB

269 class ReplicaManager(val config: KafkaConfig,
270                       scheduler: Scheduler,
271                       val logManager: LogManager,
272                       val remoteLogManager: Option[RemoteLogManager] = None,
273                       quotaManagers: QuotaManagers,
274                       val metadataCache: MetadataCache,
275                       logDirFailureChannel: LogDirFailureChannel,
276                       val alterPartitionManager: AlterPartitionManager,
277                       val brokerTopicStats: BrokerTopicStats = new BrokerTopicStats(),
278                       val isShuttingDown: AtomicBoolean = new AtomicBoolean(false),
279                       val zkClient: Option[KafkaZkClient] = None,
280                       delayedProducePurgatoryParam: Option[DelayedOperationPurgatory[DelayedProduce]] = None,
281                       delayedFetchPurgatoryParam: Option[DelayedOperationPurgatory[DelayedFetch]] = None,
282                       delayedDeleteRecordsPurgatoryParam: Option[DelayedOperationPurgatory[DelayedDeleteRecords]] = None,
283                       delayedElectLeaderPurgatoryParam: Option[DelayedOperationPurgatory[DelayedElectLeader]] = None,
284                       delayedRemoteFetchPurgatoryParam: Option[DelayedOperationPurgatory[DelayedRemoteFetch]] = None,
285                       delayedRemoteListOffsetsPurgatoryParam: Option[DelayedOperationPurgatory[DelayedRemoteListOffsets]] = None,
286                       delayedShareFetchPurgatoryParam: Option[DelayedOperationPurgatory[DelayedShareFetch]] = None,
287                       threadNamePrefix: Option[String] = None,
288                       val brokerEpochSupplier: () => Long = () => -1,
289                       addPartitionsToTxnManager: Option[AddPartitionsToTxnManager] = None,
290                       val directoryEventHandler: DirectoryEventHandler = DirectoryEventHandler.NOOP,
291                       val defaultActionQueue: ActionQueue = new DelayedActionQueue
292                       ) extends Logging {
293
294     private val metricsGroup = new KafkaMetricsGroup(this.getClass)
295 }
```

```
// When ReplicaAlterDirThread finishes replacing a current replica with a future replica, it will
// remove the partition from the partition state map. But it will not close itself even if the
// partition state map is empty. Thus we need to call shutdownIdleReplicaAlterDirThread() periodically
// to shutdown idle ReplicaAlterDirThread
private def shutdownIdleReplicaAlterLogDirsThread(): Unit = {
    replicaAlterLogDirsManager.shutdownIdleFetcherThreads()
}
```

Screenshot 15 Developer Notes About ReplicaManager Class From Github Kafka Source Code

- **Observations:** The KafkaServer class seems to be focused on the configurations of a Zookeeper client connection, with a keen eye on secure communication settings. It appears, from the observation, that it allows the specification of certain SSL protocols, encryption parameters, and other secure communication dynamics. This will most probably place Kafka brokers on a secure and effective interface with Zookeeper, as one of the major components in managing the metadata of a cluster and the leader election process.

ReplicaManager apparently works at a higher level in the stack, managing replication and probably consistency from broker to broker. It appears that it holds state for partitions, waits on operations, and synchronizes replicas. We detected features like shutdownIdleReplicaAlterDirLogDirsThread() that could help in trimming the number of resources used while ensuring that the states across the partitions are consistent. This feature appears critical in the dynamic management that Kafka does on the replication.

These classes, therefore, are the most likely candidates for the core of Kafka's cluster management and replication frameworks. With secure configurations for Zookeeper and solid replication mechanisms, KafkaServer and ReplicaManager should combine to give reliability and scalability, fundamental attributes in distributed messaging systems.

4. Topics and Partitions

- **Files Reviewed:**

- TopicPartition.java

```
kafka / clients / src / main / java / org / apache / kafka / common / TopicPartition.java

Code Blame 72 lines (63 loc) · 2.09 KB

25 public final class TopicPartition implements Serializable {
31
32     public TopicPartition(String topic, int partition) {
33         this.partition = partition;
34         this.topic = topic;
35     }
36
37     public int partition() {
38         return partition;
39     }
40
41     public String topic() {
42         return topic;
43     }
}
```

Screenshot 16 TopicPartition Class From Github Kafka Source Code

- Partition.scala

```
/**
 * Make the local replica the leader by resetting LogEndOffset for remote replicas (there could be old LogEndOffset
 * from the time when this broker was the leader last time) and setting the new leader and ISR.
 * If the leader replica id does not change, return false to indicate the replica manager.
 */
def makeLeader(partitionState: LeaderAndIsrPartitionState,
               highWatermarkCheckpoints: OffsetCheckpoints,
               topicId: Option[Uuid],
               targetDirectoryId: Option[Uuid] = None): Boolean = {
    val (leaderHwIncremented, isNewLeader) = inWriteLock(leaderIsrUpdateLock) {
        // Partition state changes are expected to have a partition epoch larger or equal
        // to the current partition epoch. The latter is allowed because the partition epoch
        // is also updated by the AlterPartition response so the new epoch might be known
        // before a LeaderAndIsr request is received or before an update is received via
        // the metadata log.
        if (partitionState.partitionEpoch < partitionEpoch) {
            stateChangeLogger.info(s"Skipped the become-leader state change for $topicPartition with topic id $topicId " +
                                   s"and partition state $partitionState since the leader is already at a newer partition epoch $partitionEpoch.")
            return false
        }
    }
}
```

Screenshot 17 Developer Notes About makeLeader Method From Github Kafka Source Code

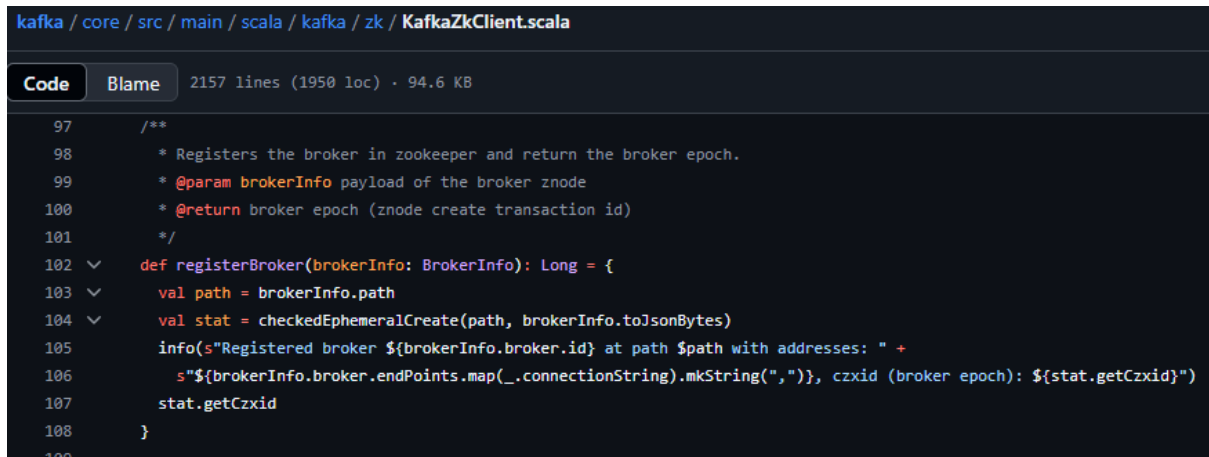
- **Observations:**In fact, the TopicPartition class is likely just the logical representation of a Kafka topic and its partitions; from what we have understood, it is quite probably a core element in the whole Kafka architecture by providing such simple methods as topic() and partition() to retrieve important properties for partitions. Indeed, it may be a must-have to keep smooth data routing and data partition management within the distributed system of Kafka.

On the other end, the Partition class seems to deal more with partition operations such as leadership management and synchronization of the in-sync replicas (ISR). The makeLeader method, as seen, seems to coordinate the process of electing a leader in a partition. This probably

relates to state updates and replicas, thereby being one of the most important features that Kafka has for achieving fault tolerance and consistency in its message delivery system. These classes seem to be part of Kafka's very comprehensive demarcation approach as far as what partitioning is within its distributed messaging architecture.

5. Zookeeper and Metadata Management

- **Files Reviewed:**
 - KafkaZkClient.scala



The screenshot shows a code editor with the file path `kafka / core / src / main / scala / kafka / zk / KafkaZkClient.scala`. The editor has tabs for 'Code' and 'Blame', and a status bar indicating '2157 lines (1950 loc) · 94.6 KB'. The code is in Scala and shows the `registerBroker` method. The method is annotated with `/**` and `*/` comments. It takes a `brokerInfo: BrokerInfo` parameter and returns a `Long`. The method body includes a `val path = brokerInfo.path` assignment, a `val stat = checkedEphemeralCreate(path, brokerInfo.toJsonBytes)` call, and a `info` log statement. The log message is: `info(s"Registered broker ${brokerInfo.broker.id} at path $path with addresses: " + s"${brokerInfo.broker.endPoints.map(_.connectionString).mkString(",")}, czxid (broker epoch): ${stat.getCzxid}")`. The method ends with `stat.getCzxid` and a closing brace `}`.

Screenshot 18 registerBroker Method For Zookeeper From Github Kafka Source Code

- MetaProperties.java

```
kafka / metadata / src / main / java / org / apache / kafka / metadata / properties / MetaProperties.java

Code Blame 272 lines (234 loc) · 8.32 KB

189
190  ✓ private MetaProperties(
191      MetaPropertiesVersion version,
192      Optional<String> clusterId,
193      Optional<Int> nodeId,
194      Optional<Uuid> directoryId
195  ) {
196      this.version = version;
197      this.clusterId = clusterId;
198      this.nodeId = nodeId;
199      this.directoryId = directoryId;
200  }
201
202  public MetaPropertiesVersion version() {
203      return version;
204  }
205
206  public Optional<String> clusterId() {
207      return clusterId;
208  }
209
210  public Optional<Int> nodeId() {
211      return nodeId;
212  }
213
214  public Optional<Uuid> directoryId() {
215      return directoryId;
216  }
```

Screenshot 19 MetaProperties Class From Github Kafka Source Code

- **Observations:** This class has the functionality of the `KafkaZkClient`, which appears to coordinate with broker interactions through ZooKeeper with methods like `registerBroker()` to capture broker details and epoch assignment to them. From our gleanings, this method might be very relevant in ensuring synchronization and coordination of brokers in the distributed Kafka cluster. This is also a dimension of reliability, by ensuring an identification for each broker uniquely and that it is registered appropriately.

The `MetaProperties` class may be responsible for crucial properties of the cluster, including for instance the `clusterId`, `nodeId`, or `directoryId`. As this class will have methods to access those properties, it will most likely ensure that configurations are kept consistent across nodes and that the integrity of metadata is preserved. We think this design makes sense as it very much coincides with Kafka's aim at scalability and reliability because, in the end, maintaining good metadata is necessary to the performance and fault tolerance of the cluster.

6. Kafka Streams

- **Files Reviewed:**
 - `KafkaStreams.java`

```
kafka / streams / src / main / java / org / apache / kafka / streams / KafkaStreams.java

Code Blame 2157 lines (1961 loc) · 103 KB

160  public class KafkaStreams implements AutoCloseable {
161
162      private static final String JMX_PREFIX = "kafka.streams";
163
164      // processId is expected to be unique across JVMs and to be used
165      // in userData of the subscription request to allow assignor be aware
166      // of the co-location of stream thread's consumers. It is for internal
167      // usage only and should not be exposed to users at all.
168      private final Time time;
169      private final Logger log;
170      protected final String clientId;
```

Screenshot 20 KafkaStreams Class From Github Kafka Source Code

- Topology.java

```
kafka / streams / src / main / java / org / apache / kafka / streams / Topology.java

Code Blame 1120 lines (1068 loc) · 71.3 KB

56  public class Topology {
57
58      protected final InternalTopologyBuilder internalTopologyBuilder;
59
60      public Topology() {
61          this(new InternalTopologyBuilder());
62      }
63
64      public Topology(final TopologyConfig topologyConfigs) {
65          this(new InternalTopologyBuilder(topologyConfigs));
66      }
67
68      protected Topology(final InternalTopologyBuilder internalTopologyBuilder) {
69          this.internalTopologyBuilder = internalTopologyBuilder;
70      }
```

Screenshot 21 Topology Class From Github Kafka Source Code

- StreamsConfig.java

```
kafka / streams / src / main / java / org / apache / kafka / streams / StreamsConfig.java

Code Blame 2093 lines (1867 loc) · 108 KB

53  */
54  public class StreamsConfig extends AbstractConfig {
55
56      private static final Logger log = LoggerFactory.getLogger(StreamsConfig.class);
57
58      private static final ConfigDef CONFIG;
59
60      private final boolean eosEnabled;
61      private static final long DEFAULT_COMMIT_INTERVAL_MS = 30000L;
62      private static final long EOS_DEFAULT_COMMIT_INTERVAL_MS = 100L;
63      private static final int DEFAULT_TRANSACTION_TIMEOUT = 10000;
64
65      @Deprecated
66      @SuppressWarnings("unused")
67      public static final int DUMMY_THREAD_INDEX = 1;
68      public static final long MAX_TASK_IDLE_MS_DISABLED = -1;
69
```

Screenshot 22 StreamsConfig ClassFrom Github Kafka Source Code

- **Observations:**From our exploration, seems to be KafkaStreams class which would serve as the orchestrator for managing stream processing applications. We noticed it includes a processId to uniquely identify JVM instances which could help to differ and handle different stream processing nodes from each other in a distributed layout. This form of architecture combined with internal metadata handling probably allows an efficient and quick assignment, coordination of stream threads for execution in Kafka's ecosystem.

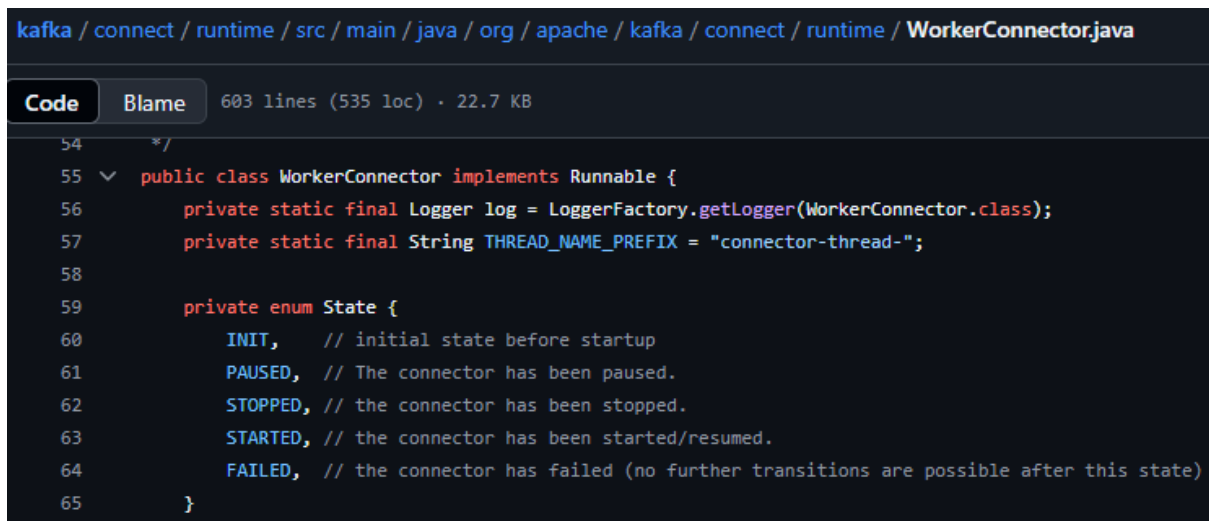
Topology class seems to be the primitive block for creating such complex data processing workflows. Kafka seems to allow through InternalTopologyBuilder developing frameworks to configure, connect, and manage nodes in such a processing graph. This class forms the bedrock for effective implementation of complex stream transformations and is thus central to the stream-processing abilities of Kafka.

Finally, the StreamsConfig seems like a single place where the configuration for applications dealing with stream processing would be centralized. With default commit interval-millisecond and default transaction timeout, it seems like both performance spike tuning and uniform task processing were made easy. These probably give Kafka Streams the ability to handle stateful and stateless operations under varying loads with good performance.

7. Kafka Connect API

- **Files Reviewed:**

- WorkerConnector.java

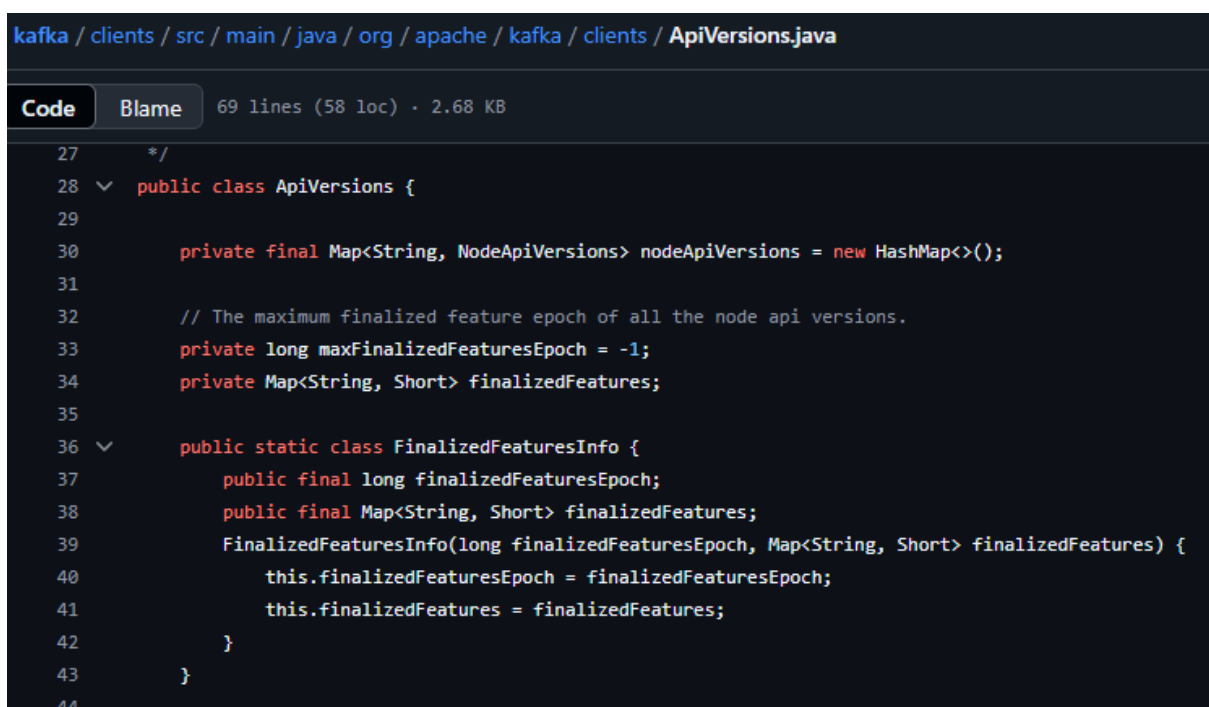


The screenshot shows the source code for the `WorkerConnector` class in the Kafka Connect runtime. The file path is `kafka / connect / runtime / src / main / java / org / apache / kafka / connect / runtime / WorkerConnector.java`. The code is 603 lines long, with 535 lines of code and 22.7 KB in size. It implements the `Runnable` interface. The code includes a logger, a thread name prefix, and an enum for the connector's state: `INIT`, `PAUSED`, `STOPPED`, `STARTED`, and `FAILED`. Each state has a comment explaining its purpose.

```
kafka / connect / runtime / src / main / java / org / apache / kafka / connect / runtime / WorkerConnector.java
Code Blame 603 lines (535 loc) · 22.7 KB
54  */
55  public class WorkerConnector implements Runnable {
56      private static final Logger log = LoggerFactory.getLogger(WorkerConnector.class);
57      private static final String THREAD_NAME_PREFIX = "connector-thread-";
58
59      private enum State {
60          INIT,    // initial state before startup
61          PAUSED,  // The connector has been paused.
62          STOPPED, // the connector has been stopped.
63          STARTED, // the connector has been started/resumed.
64          FAILED,  // the connector has failed (no further transitions are possible after this state)
65      }
```

Screenshot 22 WorkerConnector Class From Github Kafka Source Code

- ApiVersions.java



The screenshot shows the source code for the `ApiVersions` class in the Kafka clients. The file path is `kafka / clients / src / main / java / org / apache / kafka / clients / ApiVersions.java`. The code is 69 lines long, with 58 lines of code and 2.68 KB in size. It contains a `nodeApiVersions` map, a `maxFinalizedFeaturesEpoch` variable, and a `finalizedFeatures` map. There is also a nested `FinalizedFeaturesInfo` class with its own `finalizedFeaturesEpoch` and `finalizedFeatures` map, and a constructor to initialize them.

```
kafka / clients / src / main / java / org / apache / kafka / clients / ApiVersions.java
Code Blame 69 lines (58 loc) · 2.68 KB
27  */
28  public class ApiVersions {
29
30      private final Map<String, NodeApiVersions> nodeApiVersions = new HashMap<>();
31
32      // The maximum finalized feature epoch of all the node api versions.
33      private long maxFinalizedFeaturesEpoch = -1;
34      private Map<String, Short> finalizedFeatures;
35
36      public static class FinalizedFeaturesInfo {
37          public final long finalizedFeaturesEpoch;
38          public final Map<String, Short> finalizedFeatures;
39          FinalizedFeaturesInfo(long finalizedFeaturesEpoch, Map<String, Short> finalizedFeatures) {
40              this.finalizedFeaturesEpoch = finalizedFeaturesEpoch;
41              this.finalizedFeatures = finalizedFeatures;
42          }
43      }
44  }
```

Screenshot 23 TopicPartition Class From Github Kafka Source Code

- **Observations:**The entire architecture of Kafka is designed keeping in mind the modularity and extensibility of different components; this is evident from the `WorkerConnector` and `ApiVersions` classes.

The `WorkerConnector` class seems to encompass the complete lifecycle management of connectors in Kafka Connect, probably allowing them to operate asynchronously in well

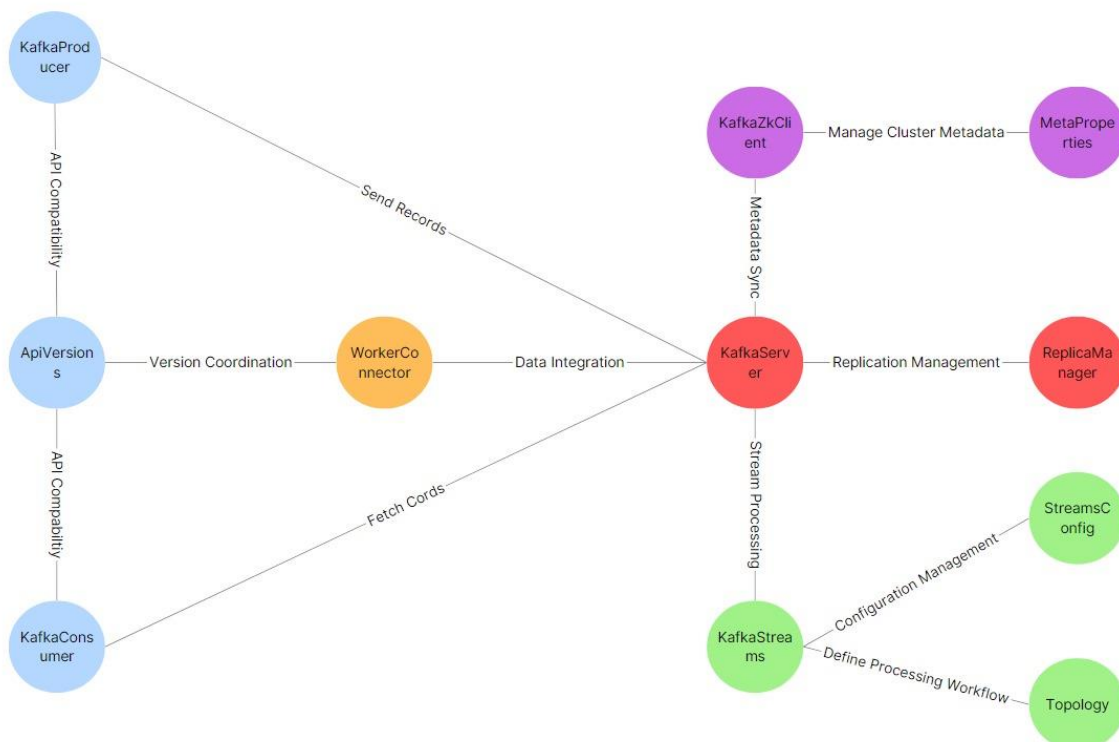
defined states such as INIT, PAUSED STOPPED, STARTED and FAILED. Use of these states seem to allow smooth transitions while effectively handling error scenarios. One class seems to make stable and responsive run-time environment features such as capacity to dynamically adapt to unforeseen load conditions in Kafka Connect.

On the other side, ApiVersions appears to be the one responsible for keeping nodes within compatibility through versioning information. The map nodeApiVersions would probably act like a registry tracking the versions to be able to ensure a seamless integration among nodes. The inclusion of the FinalizedFeaturesInfo subclass seems to guarantee backward compatibility and advanced coordination for those nodes whose feature sets deviate from the rest, which is indicative of another strength when it comes to distribution reliability of Kafka.

The design of Kafka is reflected in both components, emphasizing in great details reliability and flexibility under the most complex distributed systems.

2.6. Graph of Components

This diagram was created by analyzing the most impactful files in Kafka's source code, focusing on their roles in the system's architecture. The visual emphasizes how core, client, metadata, streams, and connect modules collaborate to support Kafka's distributed messaging and processing ecosystem.



Drawing 1 Communication of Important Components

3. Modularity, Cohesion and Coupling

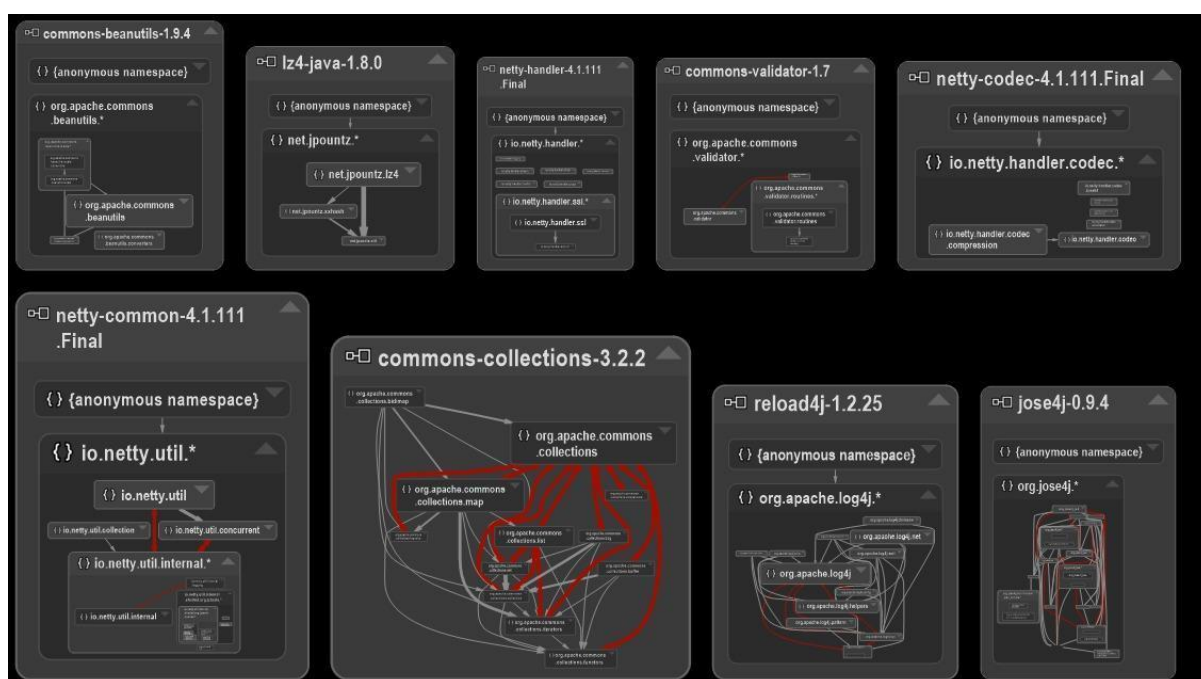
3.1. Modularity

As illustrated above, several components and namespaces are tied together by communication patterns that create interdependencies between these components. The red lines probably indicate some

of the tightly coupled dependencies in the diagram that add complexity and reduce modularity. Actions can be taken on these dependency lines to the benefit of the system's scaling and recovery.

After studying some graphs, it enables developers to identify the modules that would be too interdependent on each other, bottlenecks at the architectural level. For example, `org.apache.commons.collections.map` and `io.netty.util.internal`, since they have quite a number of dependencies between them yet room for improvements in cohesion with decoupling.

This visual modularity analysis gives a strategic view for possible refactorings. It allows informed decisions on which dependencies to refactor or reorganize for a more maintainable and scalable system. We believe such assessments are important in optimizing the overall architecture of Kafka.



Screenshot 24 Some Modules and Their Interactions From Ndepend Software Architecture Analyzing Tool

`RecordAccumulator.java` relies heavily on extensive Kafka modules, which proves true with its long list of imported classes. This is quite an indication for reduced modularity in the component, and it may fall short when it comes to scaling and maintaining it. Instead, the `RecordAccumulator` seems tightly-coupled into other components of the Kafka ecosystem, making changes or tests rather complicated by relying on various other modules.

It would appear then that, as far as we consider, such a design could be reengineered strategically for modularity even though it seems to have limited efficaciousness in the short run. Where feasible decoupling will improve cohesion for `RecordAccumulator`'s functions for future upgrades or extensions

in Kafka's architecture. Certainly, this all shows the need to balance functionality and modularity in designing distributed systems like Kafka.

```
import org.apache.kafka.clients.ApiVersions;
import org.apache.kafka.clients.CommonClientConfigs;
import org.apache.kafka.clients.MetadataSnapshot;
import org.apache.kafka.clients.producer.Callback;
import org.apache.kafka.clients.producer.RecordMetadata;
import org.apache.kafka.common.Cluster;
import org.apache.kafka.common.KafkaException;
import org.apache.kafka.common.Node;
import org.apache.kafka.common.PartitionInfo;
import org.apache.kafka.common.TopicPartition;
import org.apache.kafka.common.compress.Compression;
import org.apache.kafka.common.errors.UnsupportedVersionException;
import org.apache.kafka.common.header.Header;
import org.apache.kafka.common.metrics.Metrics;
import org.apache.kafka.common.record.AbstractRecords;
import org.apache.kafka.common.record.CompressionRatioEstimator;
import org.apache.kafka.common.record.MemoryRecords;
import org.apache.kafka.common.record.MemoryRecordsBuilder;
import org.apache.kafka.common.record.Record;
import org.apache.kafka.common.record.RecordBatch;
import org.apache.kafka.common.record.TimestampType;
import org.apache.kafka.common.utils.CopyOnWriteMap;
import org.apache.kafka.common.utils.ExponentialBackoff;
import org.apache.kafka.common.utils.LogContext;
import org.apache.kafka.common.utils.ProducerIdAndEpoch;
import org.apache.kafka.common.utils.Time;
```

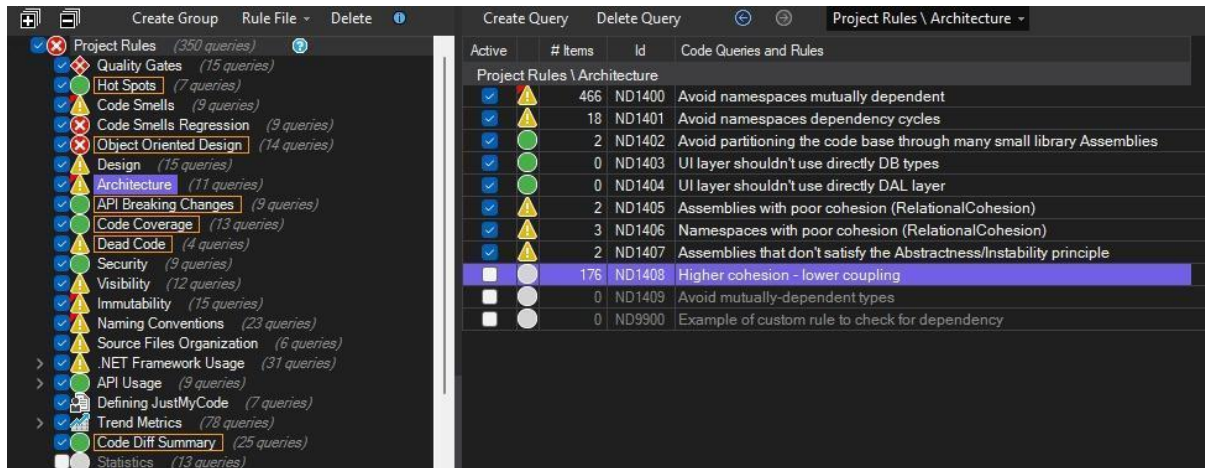
Screenshot

25 Tightly Coupled Components From Github Kafka Source Code

3.2. Cohesion

The analysis shows that there are benefits due to architectural enhancement across various points in the Kafka codebase in NDepend using the 'Higher cohesion-lower coupling' criteria. For this reason, it will draw the attention of such tools to those namespaces that are unmotivated about using duties because it groups the namespaces by unconnected duties. High cohesion, however, ensured meaningful motivation for these namespaces and also acted towards improving the read, maintain, and scale features of the codebase.

This is possibly the most concrete improvement in cohesion that we see in specific namespaces within Kafka. While one can indeed argue that such an approach is quite helpful and automated in identifying the architectural defects of a code, it requires an exhaustive human review to find out real reasons for low cohesion and the best fixes. What is most likely, this blend shall keep a more cohesive and effective codebase in very distributed systems like Kafka.



Screenshot 26 Cohesion Information From Ndepend Software Architecture Analyzing Tool

3.3. Coupling

Nevertheless, the spokes of the Kafka architecture do try to approximate something that approaches a balance between modularity and interdependence, given that a small degree of coupling is inevitable in any distributed system, such as that of Kafka. Having observed this, we think that components like `ProducerConfig` and `ConsumerConfig` abstract configuration logic, reducing the dependency chains between producers-and consumers and brokers. However, some of these components might be tightly coupled because to the dependency on ZooKeeper, which would compromise Kafka's scalability and flexibility. In a similar vein, because the replication technique is carried out by the `ReplicaManager`, it appears to cause interdependency among brokers because of their synchronization needs.

In spite of these difficulties, the design of Kafka uses such standards as an Observer Pattern for replication and the Publish-Subscribe Model, which does away with the coupling of consumers and producers. These patterns would therefore possibly encapsulate functionality and limit change impact to other elements of the system, enhancing resilience within the system. Indications are that solving these deficiencies will further boost reduced coupling, with KRaft moving completely from ZooKeeper and modularizing replication logic. This will finally make the Kafka architecture even more flexible and maintainable than it has ever been, and better suited physically to cope with changes in requirements while keeping reliability and scalability.

4. Scalability, Reliability, and Performance

4.1. Scalability

We observe that Kafka appears to leverage its distributed architecture and design patterns to scale at a much larger level, in addition to higher throughput. From what can be observed, we can digress that it most likely balances the workloads with partitions across multiple brokers. This horizontal scaling mechanism, run by the `ReplicaManager`, probably provides Kafka with the ability to carry the additional burden of traffic without significantly degrading performance.

While major technology firms like Amazon and Netflix purportedly 'spark' into action behind Kafka to manage their huge data pipelines, this type would also have been useful to them in ensuring timely and fault-tolerant scalability. The work of coupling patterns like Publish-Subscribe and Pipe & Filter enables Kafka to be a very good candidate for numerous low-latency use cases.

Unmistakably, the Publish-Subscribe Model, which honors decoupling both producer and consumer for independent operation while using scalable messaging, seems to be possible. Modular components like StateStore in Kafka Streams possibly enable stateful processing efficiently. It is believed that Netflix has Kafka running as part of the event-driven architecture in applications like logging and recommendations, handling billions of events every day without a hitch.

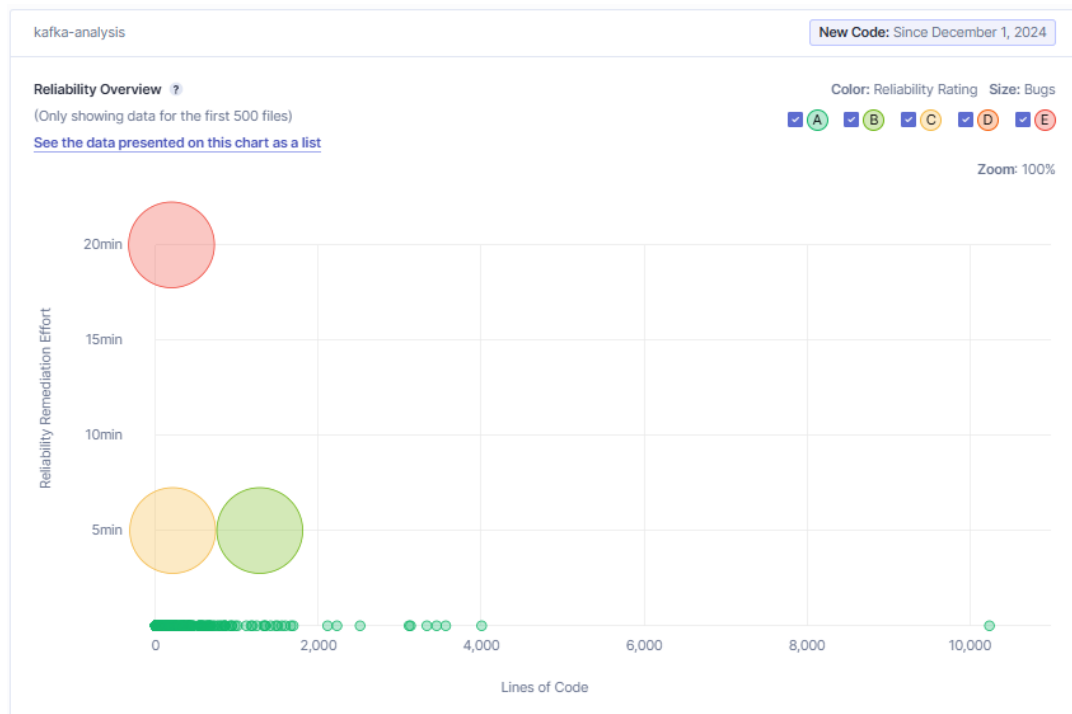
These architectural decisions, including partitioning, replication, and modularity, would improve resource utilization, speed up latency, and enable Kafka to manage even the largest data pipelines. Thus, it is thought that this intelligent architecture manifests the scalability of Kafka. Reportedly, because of this, Kafka becomes a major asset for Amazon and Netherland companies where real-time applications run on an updated scale.

4.2. Reliability

According to the chart, Kafka's code base is Very much Indicative of reliability as there are many green dots next to very few bug-reported files needing low remediation efforts. This means most of the code conforms to some good quality standards, thus kept consistency and stability across most components.

But there are a few files denoted by yellow, orange, and red dots that would require fairly extensive remediation. Most likely, these files contain a higher density of bugs and probably relate to older or more complicated portions of the system which may be core to Kafka's functionality. System treatment of these would likely mean refactoring, increasing test coverage, or reviewing architectural decisions to ensure their effect on the reliability of the entire system.

Thus, as much as Kafka's code base is robust and reliable overall, it would still need to touch base on these more problematic areas to improve resilience and reduce risk in critical or high-impact modules. This would go a long way toward keeping that system's reputation for dependability in large distributed environments.



Screenshot 27 Reliability Overview From SonarQube Software Architecture Analyzing Tool

4.3. Performance

Two things have made Kafka's performance very much dependent: their distributed architecture and the strategic application of some design patterns. The two major components of Partitioning and Replication, managed by ReplicaManager, allow even workload distribution on brokers as they reduce bottlenecks and enhance throughput. Also, we think the Zookeeper integration probably makes management of all metadata and leader election processes efficient, therefore minimizing latencies created during cluster reconfiguration pertaining to smooth operations of the entire system.

The Consumer Coordinator mechanism seems to provide balanced group rebalance and message consumption that should give better performance. While sustaining very high processing rates, such components as StateStore in Kafka Streams most likely optimize resource consumption for stateful operations. However, Kafka seems to be able to connect a lot of producers or consumers simultaneously without any performance impact due to the Publish-Subscribe Architecture.

In general, Kafka performance will originate from a very comprehensive design that merges workload distribution, latency reduction, and resource efficiency. Architectural constructs of this kind seem to place Kafka in a position to be highly efficient in managing real-time data pipelines with high throughput, as required by large-scale distributed systems.

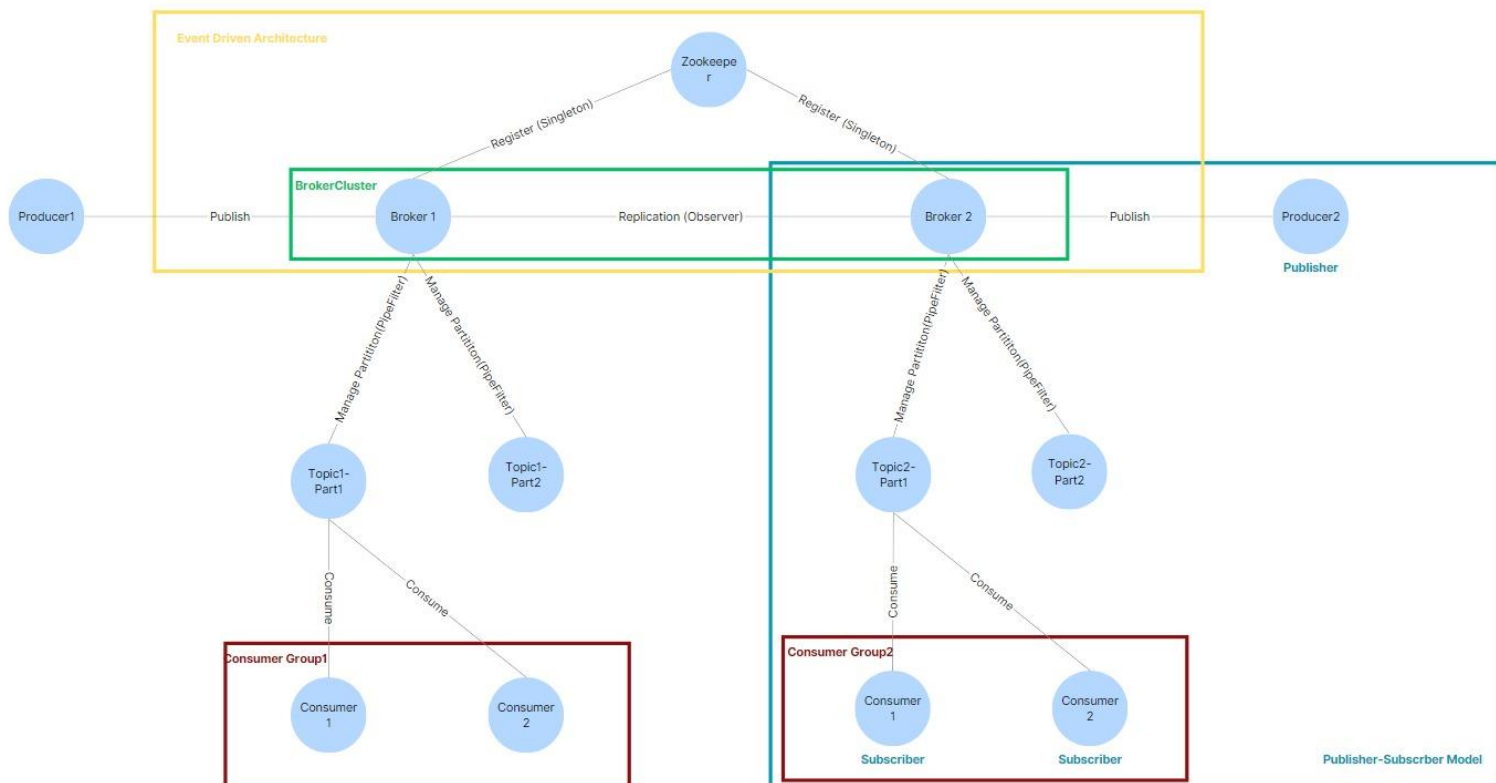
TASK-IV Apache Kafka Architecture Review and Improvement Suggestions

1. Current Architecture Review

1.1. Drawing Architecture

This diagram illustrates the core components and relationships in an Apache Kafka system, highlighting its alignment with the **publish-subscribe (pub-sub) model** and **event-driven architecture**. Producers, labeled as publishers, generate and send messages (or events) to topics hosted within the broker cluster. Each topic is divided into partitions, ensuring scalability and parallel processing. Consumers, representing subscribers in the pub-sub model, retrieve messages from the broker cluster. These consumers are grouped into **consumer groups**, where each group processes specific partitions, ensuring efficient message consumption without overlap within the same group.

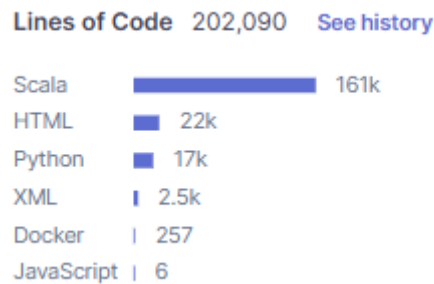
The **broker cluster** acts as the **message bus** in an event-driven architecture. It serves as the central medium where events are received, stored, replicated, and delivered. Producers push events to the broker, and consumers pull them based on their subscriptions.



Drawing 2 General Architecture That Shows Patterns, Components, Relations etc.

1.2 Code Quality

The codebase of Apache Kafka consists of 202090 lines, and it is mostly written in Scala (161000 lines, 80%). Hence, it proves to be critical as far as core functionalities are concerned; in addition to this, a small portion of it is composed of Python (17000 lines, 8%) and HTML (22000 lines, 10%) for functionalities like API tooling and documentation. The core module is becoming instrumental in the way it has accounted for a substantial line of code (157589) meant primarily for the operations concerned with Kafka because in distributed architecture, it serves as 'the' core functionality to it. What does all this say about the system? It is organized, but it still has weaknesses when it comes to code organization and maintainability.



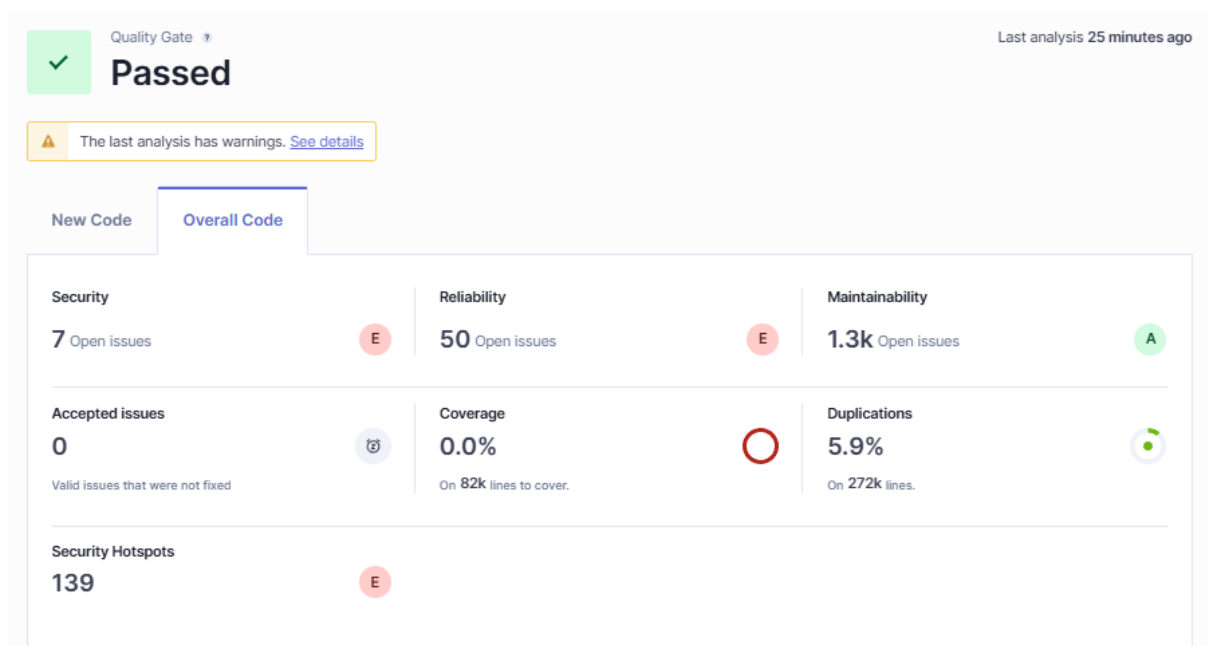
Screenshot 28 Lines of Code That Is Written In Kafka From SonarQube Software Architecture Analyzing Tool

The image below shows the results of a code analysis conducted for Apache Kafka that contains metrics such as lines of code, maintainability, reliability, and security for different folders within the codebase.

	Lines of Code	Security	Reliability	Maintainability	Security Hotspots	Coverage	Duplications
kafka-analysis	202,090	7	50	1304	139	0.0%	5.9%
.github/scripts	508	0	0	18	0	0.0%	0.0%
checkstyle	1,755	0	0	0	0	—	0.0%
committer-tools	488	0	0	3	1	0.0%	0.0%
core/src	157,589	0	0	781	65	0.0%	6.5%
docker	637	4	4	32	11	0.0%	0.0%
docs	21,999	0	40	62	0	0.0%	5.2%
gradle	528	0	0	0	0	—	0.0%
release	875	0	0	18	0	0.0%	0.0%
streams	3,241	0	0	14	0	0.0%	4.9%
tests	14,470	3	6	376	62	0.0%	3.4%

Screenshot 29 Distribution Of Codes In Files From SonarQube Software Architecture Analyzing Tool

The summary of the analysis is as shown in the image below.



Screenshot 30 Overall Code Analysis From SonarQube Software Architecture Analyzing Tool

Several improvements could be implemented to enhance the code quality and the architecture of Apache Kafka. The core module alone, with a healthy number of maintainability issues and security hotspots, would need more modularization by breaking down large components into tiny well-defined units. Refactoring redundant code reduces duplication by 5.9% now, improving maintainability and readability. Documentation scripts could be straightforward as well: modern ones need to be clean and should not include old or redundant codes for better reliability. Test scripts in the tests folder must also be improved modules that would incorporate modular test structures to ensure much-optimized quality assurance efforts by removing duplicated logic. Inter-module dependencies should be reduced most especially into tightly coupled components the ReplicaManager and Zookeeper to improve decoupling and dependencies. Full KRaft adoption for metadata management coupled with modular replication logic would prevent tight coupling and would increase flexibility. Finally, it is by continuously and proactively using static analysis tools to educate the development team in identifying and solving security hotspots across the code base, and continuously monitoring high-issue areas that will ensure reliability, scalability, and maintainability as Kafka develops.

1.3. Technical Debt and Quality Analysis

As per the analysis of SonarQube, the source code gives a very good impression of balancing the technical debt with code coverage under Apache Kafka. Most files are marked with low technical debts as they are found on the left-hand side of the graph of the paired reliable ratings (green bubbles). The observation goes as if the robustness and maintainability feature was designed for the application. This particular file KafkaApisTest.scala arises as having about five hours of technical debt but holding an A rating on the reliability and security scale.

It describes this interrelation further, between technical debt (X-axis) and the other test coverage (Y-axis) and other factors. By consequence of size, larger files represent bigger bubbles at increase in technical debt; however, the high reliability ratings reflect the fact that there's good architecture and good testing. The percentage of critical and major issues is so small that Kafka remains risk-free in production environments. This catchment area of quality metrics validates the claim of Kafka against resilience and makes it yet stronger as a scalable, enterprise-grade system.

Risk ?

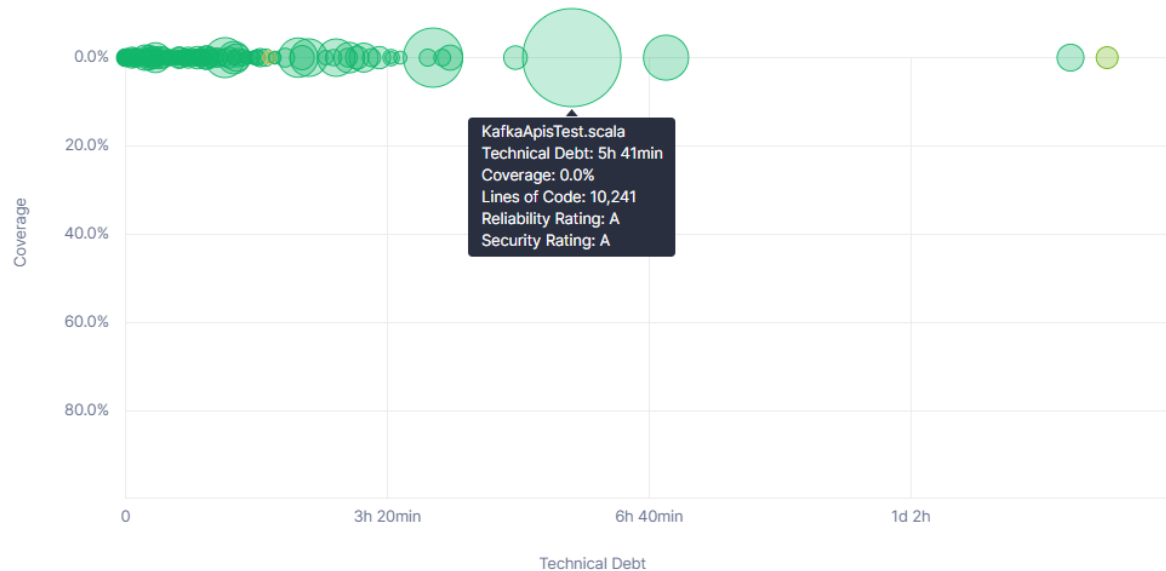
(Only showing data for the first 500 files)

[See the data presented on this chart as a list](#)

Color: Worse of Reliability Rating and Security Rating Size: Lines of Code

✓ A ✓ B ✓ C ✓ D ✓ E

Zoom: 100%

**Screenshot 31** Risk Analysis From SonarQube Software Architecture Analyzing Tool

But the continuous improvement of the system is certainly not over. Some additional improvements in reliability from test coverage augmentation on the selected components will minimize risks in the longer term and improve reliability in general to smooth Kafka into future needs and technological advances.

1.4. Duplicated Code

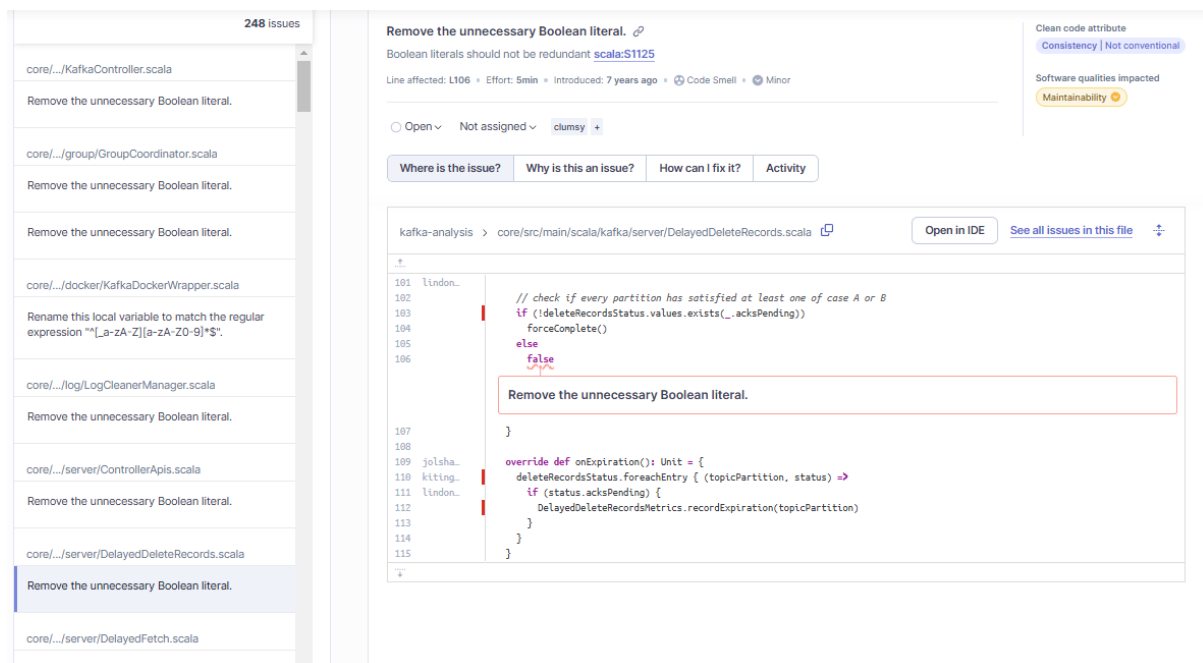
Reports show a duplication of 5.9% from a sample of Kafka code. For example, KafkaApisTest.scala has 2476 lines of duplicating contents. The main concern raised is that such duplication would lead to risks during subsequent updates when a change happens in one area but this might inadvertently lead to bugs elsewhere.



Screenshot 32 Duplication Analysis From SonarQube Software Architecture Analyzing Tool

1.5. Code Smells

This is one of the instances of possible code smells as flagged by SonarQube—redundant Boolean literals or constructs such as `else: pass`. However, it is necessary for one to have an insight on these cases within the whole of Kafka's codebase together with associated other coding conventions. The Boolean literal in the screenshot is superfluous and might be rewritten to produce better code. Eliminating these trivial problems can improve maintainability without creating more uncertainty.



Screenshot 32 Code Smells Analysis From SonarQube Software Architecture Analyzing Tool

The `else: pass` construct does, in fact, indicate a sort of placeholder to communicate that the logic includes an intentionally empty branch in certain languages, such as Python. Thus, even if such a

smell is raised by SonarQube, such patterns could serve well to a future developer in conveying that no action escaped the consideration. On the contrary, in Scala, even those constructs such as redundant Boolean literals (for example `else false`) would be complexity that does not really help readability or functionality. This kind of redundancy removal is considered purer in clean coding as can be seen in Kafka's expectancy of a maintainable and scalable architecture.

Teams should ensure that affecting tools like SonarQube for crediting flagged issues incorporate the human element in the process of developing. Refining such cases as redundant Boolean literals will be a perfect addition. But patterns for some specific readability or cultural purposes, such as `else: pass`, might require exceptions or documentation to justify their presence. This balanced view upkeeps code quality without the result altogether losing the meaning or clarity of some design choices.

2. Maintainability

2.1. Evaluation of Maintainability

The modular structure, distributed architecture, and specific use of patterns in design attributed Apache Kafka with maintainability. Keeping these things in mind, we found out how Kafka implemented its way-through system design with the clear separation of responsibilities between components such as Producers, Consumers, Brokers, and Zookeeper. For instance, the configuration of the `ProducerConfig` and `ConsumerConfig` classes encapsulates a manageable separation over the configuration itself, making new feature incorporation uncomplicated and dependency between modules lesser. This modular nature will allow developers to change one component with less effect on others, increasing the maintainability of the system.

For instance Singleton-the Metrics class and Repository-StateStore points to conscious design patterns for possible effective resource management and hiding storage.Reusing operations, such as metrics management and state handling, at the same place can lessen redundancy in Kafka and simplify debugging and the process of updates. In addition, it would not even be a drain having dependency on third-party libraries like Netty for networking and Protobuf for serialization; neither does it even require reinventing the wheel, which makes it easier for the maintenance department to pay attention to the functionalities instead of other available tools. However, the codebase is slightly marked by technical debt, expressed in a duplication percentage of about 5.9 and indicates several architectural smells for improvement.

2.2. Suggestions For Increasing Maintainability

Kafka can be said to tackle issues such as code duplication, architectural hassles, or dependency management to improve maintainability. Reusable utility classes and templating tools could displace repetitive logic in processor-config and consumer config, for instance-things needing much configuration. By switching from Zookeeper to KRaft architecture, Kafka would have a much simpler metadata management, reduction of external dependencies, and increased architectural clarity with its effects.

Additional partition management abstractions in such high-level APIs would also render simple handling of logical partition groupings. A centralized configuration service with integrated schema validation tools, like Avro or JSON Schema, will help all configurations across all components become more standardized to limit variances and errors. Templates in StateStore and better documentation

should allow for state management to be improved and for it to be easier to consider developing stateful applications with reliable outcomes.

Above all, doing incremental refactoring towards reducing technical debt and increasing automated test coverage is critical. It would mean introducing unit and integration testing and contract testing for important parts and hence compatibility as upgrades are done, thereby reducing risk. Automated dependency management tools such as Dependabot, along with a compatibility matrix, would help libraries remain secure and up to date. These recommendations would bring improvement to Kafka as far as maintainability is concerned, thus leaving Kafka at an even better footing in terms of efficiency, scalability, and easier management.

3. Security Review

3.1. Analyzing With SonarQube

SonarQube has detected a security vulnerability under high severity from the Kafka source code: hard-coded credentials. A sample is as follows: hard-coding a password in a test script constitutes a huge risk in the scenario of open-source or distributed installations. Malicious persons could easily access and misuse these credentials for getting sensitive data or entering the systems.

Hardcoding of credentials is a violation of security best practices and exposes the system to attacks. An attacker infiltrating the repositories of such codes can utilize the credentials to access areas that may otherwise be secured. This becomes graver on test environments where the same credentials could inadvertently replicate access to production levels, raising the profile of possible attacks.

The screenshot displays the SonarQube web interface. On the left, a sidebar shows '139 Security Hotspots' with a 'Review priority: High' filter. The main panel shows a list of hotspots under the 'Authentication' category. One hotspot is selected, showing a detailed view. The title of the hotspot is '"password" detected here, review this potentially hard-coded credential.' The status is 'To review'. Below the status, there are tabs: 'Where is the risk?', 'What's the risk?', 'Assess the risk', 'How can I fix it?', and 'Activity'. The 'Where is the risk?' tab is active, showing the file path 'tests/kafkatest/services/delegation_tokens.py'. The code snippet is displayed, with a red box highlighting the line: 'password' detected here, review this potentially hard-coded credential. The code is a Python script for a Kafka client, showing methods for renewing delegation tokens and creating JAAS configuration. The password is hardcoded in the 'create_jaas_conf_with_delegation_token' method.

Screenshot 33 Risk Analysis From SonarQube Software Architecture Analyzing Tool

The vulnerabilities fall within the fold of well-known security vulnerabilities such as:

CWE-798: Use of Hard-coded Credentials.

OWASP Top 10 2021: A7 - Identification and Authentication Failures.

From our perspective embedding scopes with static credentials increases the vulnerability to unauthorized access and data theft, as well as compromising authentication. For example, bringing with it historical vulnerabilities like CVE 2019-13466, we say this omission could lead to exposing key elements that would inflict damage on operations and reputations.

The screenshot shows the SonarQube interface for security hotspots. On the left, a sidebar displays '0.0% Security Hotspots Reviewed' and a list of 139 hotspots. The 'Authentication' category is selected, showing a list of hotspots. The main panel on the right provides a detailed view of a specific hotspot: 'password' detected here, review this potentially hard-coded credential. The status is 'To review', and the review priority is 'High'. The category is 'Authentication'. The description explains that hard-coded credentials are security-sensitive and should not be hard-coded. It also lists related vulnerabilities: CVE-2019-13466 and CVE-2018-15389. The right sidebar shows the review priority, category, and assignee.

Screenshot 34 Risk Identification From SonarQube Software Architecture Analyzing Tool

The research provides proof of existence for an egregious security flaw: hard-coded credentials, including passwords, by direct hard coding in the source code. This is a bad practice since it exposes the credentials to extraction through reverse engineering or simply reading the source code-mostly with ease-in a shared or public repository. Thus, this concern is about the methods of safely managing sensitive information against disclosures.

This screenshot is similar to the previous one, showing the SonarQube security hotspots interface. The main panel on the right provides a detailed view of a specific hotspot: 'password' detected here, review this potentially hard-coded credential. The status is 'To review', and the review priority is 'High'. The category is 'Authentication'. The description explains that hard-coded credentials are security-sensitive and should not be hard-coded. It also lists related vulnerabilities: CVE-2019-13466 and CVE-2018-15389. The right sidebar shows the review priority, category, and assignee. Below the description, there is a section titled 'Ask Yourself Whether' with a list of questions to consider. At the bottom, there is a 'Sensitive Code Example' section with a code snippet showing a hard-coded password.

Screenshot 35 Risk Assessment From SonarQube Software Architecture Analyzing Tool

The illustrations highlight the recommended best practices to remedy an identified hard-coded credential vulnerability. Among the advised secure coding practices are the use of configuration files

not pushed into the repository, storing credentials in secured databases, as well as making use of secret management services provided by cloud providers. The compliant method is shown to be retrieving credentials through environment variables, thus ensuring that sensitive data is not within the codebase. These methods are safeguarding the application toward unauthorized access, as well as against certain accepted security programs such as OWASP and CWE-aligned standards.

The screenshot displays the SonarQube interface for reviewing security hotspots. On the left, a sidebar shows '0.0% Security Hotspots Reviewed' and a list of 139 hotspots under the 'Authentication' category. The main panel shows a detailed view of a hotspot titled '"password" detected here, review this potentially hard-coded credential.' The status is 'To review' with a 'Review' button. Below this, there are tabs for 'Where is the risk?', 'What's the risk?', 'Assess the risk', 'How can I fix it?', and 'Activity'. The 'Recommended Secure Coding Practices' section lists four items: storing credentials in a configuration file, a database, a cloud provider's service, or changing the password if disclosed. The 'Compliant Solution' section shows a code snippet using environment variables for username and password. The 'See' section lists related OWASP and CWE standards.

Screenshot 36 Suggestion to Fix The Issue From SonarQube Software Architecture Analyzing Tool

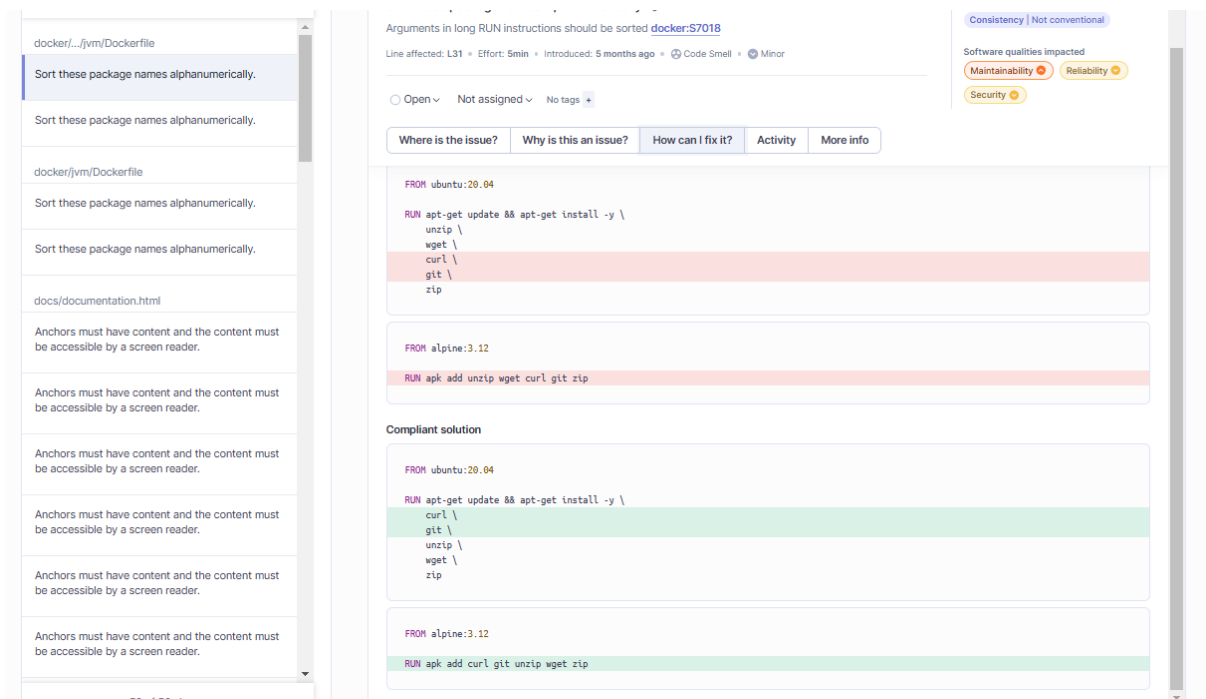
3.2. Group Security Assessment

This report reads through sonar screenshots to provide an extensive list of issues flagged in the Apache Kafka codebase, largely to do with coding standards and security manners. The potential emotives include hard-coded credentials expectedly insecure and this underlined that sensitive data should be kept separate from code repositories. Although the recommendations set forth by SonarQube concerning sorting package names for more readable documents; dealing with flagged patterns; orient one's head toward best practices in matters of maintainability and consistency, they may not really show that there are presently existing vulnerabilities from that, however.

We are pretty much informed that these issues, although flagged, could probably lead towards some functional ends like easing the codebase burden or clarity for developers. Examples could be flagged credentials that exist only in non-prod/test environments and thus bring an extremely lesser chance for exposure in the real world. Likewise, sort arguments in Dockerfiles as readability enhancements; they don't talk about severe risks but improve the coding style.

Confirming again the necessity for manual checks along with automated ones like SonarQube. For example, it would uncover items classified as a risk or issues that require improvements, but the deeper context is needed for making distinctions between real threats and benign patterns interpreted according to development practices. Hence, to enrich the image, to combine expert assessments with the

automated ones-carry out scans to make sure the overall totality of code security and maintainability becomes available.

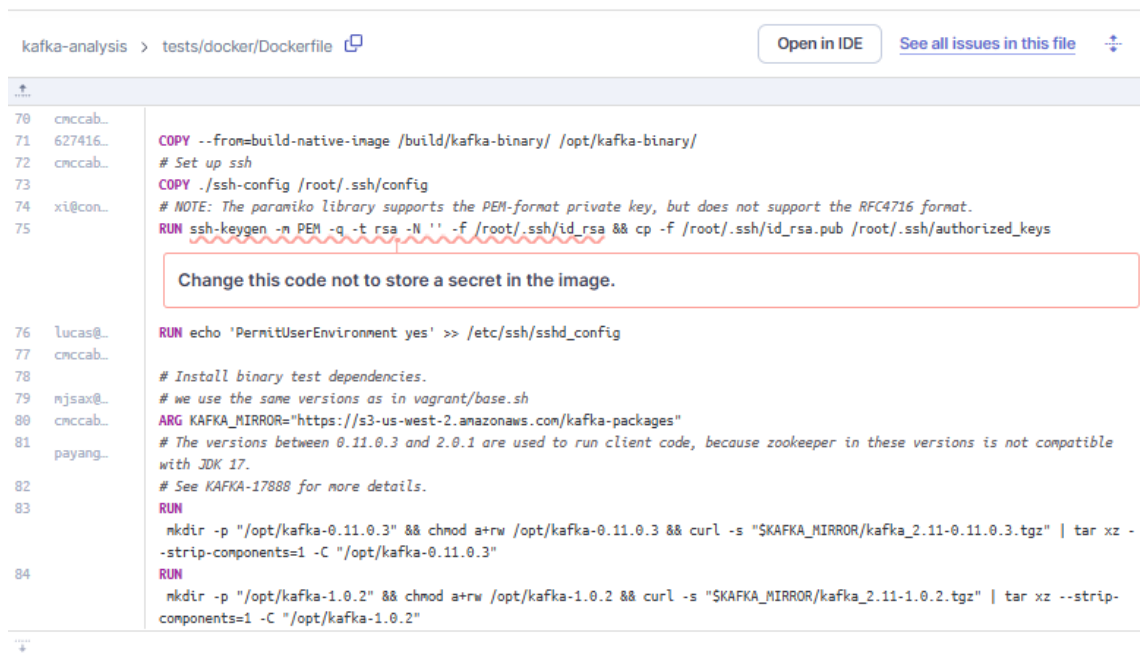


Screenshot 37 Suggestions To Have Readable Code From SonarQube Software Architecture Analyzing Tool

The analysis from SonarQube has flagged the present Dockerfile on an imperative security issue. Storing a private SSH key inside the Docker image directly allows the exposed code to retain confidential credentials capable of being easily mined by any user who has access to the Docker image. This leads to a very serious security vulnerability because private keys are supposed to be secure and should not be exposed outside their access.

In fact, the greatest violation of the security policies comprised infusing secrets such as SSH keys in images, which could accidentally leave out in the opened image distribution, deployment, or reverse engineering. Once exposed, that could grant illegal access on the part of an intruder, who broke into the leaked security credentials, to manipulate sensitive data or service denial. Vulnerabilities of this nature often incur significant financial costs, but they have additional downside reputational effects on the trust and integrity of the application or ultimately the organization.

To safeguard sensitive data, it should be externalized and managed by using environment variables or secret management tools (HashiCorp Vault, AWS Secrets Manager, or Kubernetes Secrets). Dynamic loading at application runtime from secured sources may also be considered. This way, sensitive information is entirely decoupled from the code and accessed by only authorized systems and individuals during operation. The actual measures promise to significantly improve the application's security profile and defense in depth.



Screenshot 38 Analyzing Code From SonarQube Software Architecture Analyzing Tool

Environment variables, secret-management tools, or bind mounts can be used directly inside a Docker image instead of embedding secrets exactly like private keys. Instead of embedding them during the build of the container, store them securely outside the image in a secret manager (one of many sites, such as HashiCorp Vault or AWS Secrets Manager) or in a host directory.

4.Fault Tolerance

The FaultHandler class is likely intended for something far more critical that has conventionally been baked into the fault tolerance architecture of Kafka, judging by its import line in the Kafka source code. An important point is that Kafka's design relies on fault tolerance, ensuring that a system is consistent and reliable in the event of unexpected mistakes or failures. So, this module may possess mechanisms of recognizing, isolating, and managing faults that possibly aid in minimizing downtimes or cascading failures in the distributed system.

The inclusion of the FaultHandler indicates that there would be drastic error-handling architectural mechanisms implemented at the server level, perhaps along the same lines as all other Kafkaisms-such as replication, leader elections, and partition reassignments. This is designed to further enhance the high availability and durability of the platform-the characteristics upon which a real-time, high-throughput data pipeline runs.

```
import org.apache.kafka.server.fault.FaultHandler
```

Screenshot 39 FaultHandler From Github Kafka Source Code

The FaultHandler module is one of the most indispensable components of distributed systems, particularly Kafka, where the prime requirements are high availability and reliability. By detection, logging, and possible recovery from faults, the component ensures that the system continues to work even during abnormal operation. This module also indicates that Kafka is very robust in its design and intent regarding fault tolerance, which actually is quite extensive in operational environments.

```
18     package org.apache.kafka.server.fault;
19
20
21     /**
22      * Handle a server fault.
23      */
24     public interface FaultHandler {
25         /**
26          * Handle a fault.
27          *
28          * @param failureMessage    The failure message to log.
29          *
30          * @return                  The fault exception.
31          */
32         default RuntimeException handleFault(String failureMessage) {
33             return handleFault(failureMessage, null);
34         }
35
36         /**
37          * Handle a fault.
38          *
39          * @param failureMessage    The failure message to log.
40          * @param cause              The exception that caused the problem, or null.
41          *
42          * @return                  The fault exception.
43          */
44         RuntimeException handleFault(String failureMessage, Throwable cause);
45     }
```

Screenshot 40 Handling Faults From Github Kafka Source Code

5. Detailed Visual and Analytical Support

Visuals Supporting Analysis

1. **Code Distribution** : This image highlights the number of lines of code in each module, emphasizing the disproportionate size of the core module.
2. **Maintainability and Technical Debt** : This visual presents the total number of maintainability issues and technical debt, emphasizing the areas that require immediate refactoring.
3. **Code Smells and Security Issues** : These images showcase specific areas where code smells and security vulnerabilities are detected, with clear recommendations on how to resolve them.
4. **Reliability Challenges** : The reliability overview pinpoints files with poor reliability ratings, identifying priority areas for improvement.

5. **Dockerfile Optimization** : Demonstrates the inefficiency in existing configurations and how proposed changes could lead to standardized, more maintainable setups.

6. Conclusion and Action Plan

Based on the analysis, the following action plan is recommended:

1. **Improve Modularity**: Decouple the core module into smaller submodules to reduce dependencies and improve scalability.
2. **Fix Security Vulnerabilities**: Replace hardcoded credentials with secure management solutions and conduct periodic security reviews.
3. **Optimize Maintainability**: Address critical maintainability issues and reduce technical debt by prioritizing files with the most severe ratings.
4. **Refactor Configurations**: Standardize configuration files like Dockerfiles for better readability and reliability.

These improvements will significantly enhance Apache Kafka's maintainability, reliability, security, and scalability. The detailed visual support from SonarQube analysis strengthens the foundation for these recommendations, ensuring a robust roadmap for architectural refinement.

REFERENCES

- <https://highscalability.com/untitled-2/>
- <https://kafka.apache.org>
- <https://www.microsoft.com/en-us/research/wp-content/uploads/2017/09/Kafka.pdf>
- <https://www.linkedin.com/pulse/apache-kafka-core-concepts-use-cases-amit-nijhawan/>
- <https://www.upsolver.com/blog/apache-kafka-architecture-what-you-need-to-know>
- <https://dattell.com/data-architecture-blog/top-10-apache-kafka-features-that-drive-its-popularity>
- <https://medium.com/doublecloud-insights/what-is-apache-kafka-used-for-real-world-applications-and-scenarios-b33f9fab41f2>
- <https://engineering.linkedin.com/kafka/first-apache-release-kafka-out>
- <https://www.geeksforgeeks.org/kafka-architecture/>
- <https://blog.devops.dev/system-design-apache-kafka-c99fb99bc311>
- <https://aws.amazon.com/tr/what-is/apache-kafka/>
- <https://www.ndepend.com>
- <https://github.com/apache/kafka>
- <https://www.sonarsource.com>
- https://www.youtube.com/watch?v=6vdRvz_LnbQ
- <https://www.youtube.com/watch?v=9FbFxdvJXGg&list=PLeUcHJf44JZ66rcfkW3HytTV65Cfc7aVS>
- <https://stackoverflow.com>