**Epic:** **Enhanced Kafka Consumer Framework with Multi-Destination Sync and Java 17 Upgrade**

**Objective**

The objective of this epic is to enhance the Kafka consumer framework to support seamless data synchronization from Kafka to Apache Ozone, alongside the existing Kafka-to-Hive (HDFS) syncing capability. Additionally, the epic includes upgrading the framework to Java 17, ensuring compatibility, testing, and releasing the upgraded system. Kafka-to-Oracle CDC (Change Data Capture) module needs to be completed to enable real-time data integration with Oracle databases.

**Acceptance Criteria**

* **Kafka-to-Ozone Sync**:
  + The framework successfully consumes messages from Kafka topics and syncs data to Apache Ozone with configurable mappings.
  + Supports parallel processing of Kafka messages for Ozone without impacting existing Hive sync performance.
  + Ensures data integrity and consistency during sync to Ozone, with error handling for failed writes.
  + Provides monitoring and logging for Ozone sync operations.
* **Java 17 Upgrade**:
  + The entire Kafka consumer framework is upgraded to Java 17, with all dependencies updated to compatible versions.
  + All existing and new features (Ozone sync, Hive sync, Oracle CDC) are thoroughly tested on Java 17.
  + Performance benchmarks confirm no degradation compared to the previous Java version.
  + Successful compilation and deployment of the framework on Java 17 in production-like environments.
* **Kafka-to-Oracle CDC Module**:
  + Sync from Kafka messages to Oracle databases using CDC.
  + Supports configurable mappings between Kafka topics and Oracle tables.
  + Ensures low-latency, reliable data transfer with transactional consistency.
  + Includes retry mechanisms and error handling for Oracle CDC operations.
  + Provides monitoring and logging for CDC operations.
* **Testing and Release**:
  + Comprehensive unit, integration, and end-to-end tests validate all features (Ozone sync, Hive sync, Oracle CDC) on Java 17.
  + Automated tests cover at least 90% of the codebase, including edge cases and failure scenarios.
  + Successful deployment in a staging environment with zero critical defects.
  + Release documentation includes upgrade instructions, rollback plan, and configuration changes.
* **Non-Functional Requirements**:
  + The framework maintains or improves current performance metrics (throughput, latency) for all sync operations.
  + Scalability is validated to handle increased Kafka message volumes (up to 2x current load).
  + Security practices (e.g., authentication, data encryption) are updated to comply with Java 17 standards.

**Design Details**

**Kafka-to-Ozone Sync**

* **Architecture**:
  + Extend the existing Kafka consumer framework to include an Ozone connector module.
  + Use Apache Ozone’s client library (e.g., ozone-client) to write data to Ozone buckets.
  + Implement a pluggable destination interface to support both Hive and Ozone sinks.
  + Use a configuration-driven approach to map Kafka topics to Ozone buckets and keys.
* **Data Flow**:
  + Kafka messages are deserialized using existing Avro/JSON schemas.
  + A new OzoneSink class handles batch writes to Ozone, leveraging Ozone’s filesystem API.
  + Implement parallel processing using Kafka consumer groups to distribute load across Ozone nodes.
* **Error Handling**:
  + Handle transient failures (e.g., network issues) with exponential backoff and retries.
  + Log failed writes to a dead-letter queue (DLQ) in Kafka for manual inspection.
* **Monitoring**:
  + Expose metrics (e.g., sync rate, latency, error rate) via Prometheus/JMX.
  + Integrate with existing logging framework (e.g., SLF4J) for detailed audit trails.

**Kafka-to-Hive Sync**

* **Architecture**:
  + Retain existing Hive sink implementation, ensuring compatibility with new framework changes.
  + Validate schema evolution support for Hive tables during Java 17 migration.
* **Data Flow**:
  + Continue using existing Hadoop/HDFS APIs for Hive writes.
  + Optimize batch processing to minimize impact from Java 17 changes.
* **Error Handling**:
  + Maintain existing DLQ and retry mechanisms for Hive sync failures.

**Java 17 Upgrade**

* **Migration Steps**:
  + Update build tools (e.g., Maven/Gradle) to support Java 17.
  + Replace deprecated APIs (e.g., java.util.Date with java.time) and update libraries (e.g., Kafka clients, Ozone clients).
  + Refactor code to leverage Java 17 features (e.g., records, sealed classes) where beneficial.
* **Testing**:
  + Run static analysis (e.g., SonarQube) to identify Java 17 compatibility issues.
  + Execute performance tests to compare Java 17 vs. previous version.
* **Release**:
  + Create a staged rollout plan with feature toggles for new functionality.
  + Prepare rollback scripts to revert to previous Java version if needed.

**Kafka-to-Oracle CDC Module**

* **Architecture**:
  + Implement a new OracleCDCSink module using Oracle’s JDBC driver and CDC APIs.
  + Use Kafka Connect framework (if applicable) or a custom consumer to process messages.
  + Support transactional updates to Oracle tables with configurable batch sizes.
* **Data Flow**:
  + Deserialize Kafka messages and map to Oracle table schemas.
  + Use Oracle LogMiner or XStream APIs for CDC, depending on Oracle version.
  + Implement a buffer to handle high-throughput Kafka streams.
* **Error Handling**:
  + Handle database connection failures with retries and circuit breakers.
  + Log failed CDC operations to a DLQ with metadata (e.g., message key, offset).
* **Monitoring**:
  + Expose CDC-specific metrics (e.g., transaction rate, commit latency).
  + Integrate with Oracle monitoring tools for database health checks.

**Description**

This epic focuses on extending the Kafka consumer framework to support data synchronization to Apache Ozone while preserving the existing Kafka-to-Hive (HDFS) functionality. The framework will be upgraded to Java 17, ensuring compatibility and leveraging modern Java features for improved maintainability. A new Kafka-to-Oracle CDC module will enable real-time data integration with Oracle databases, supporting use cases requiring low-latency data propagation. The epic includes comprehensive testing, performance optimization, and a robust release strategy to ensure a smooth transition to production. The enhanced framework will provide a scalable, reliable, and extensible solution for multi-destination data synchronization from Kafka.

**Epic: API Gateway Logs Synchronization to SDP View EEH Topic**

**Epic Title**

API Gateway Logs Synchronization to SDP View via EEH Topic

**Objective**

To capture, track, and synchronize user activity logs from the API Gateway to the EEH Tier 1 Kafka topic, subsequently syncing these logs to the SDP cluster using a Kafka consumer framework, and making them available in an Elasticsearch system for real-time alerting and dashboard creation.

**Description**

This epic encompasses the development of a robust logging and synchronization pipeline for user activities passing through the API Gateway. The system will ensure that all relevant user actions are logged, streamed to a Kafka topic (EEH Tier 1), consumed by the SDP cluster, and indexed in Elasticsearch for real-time monitoring, alerting, and visualization. The solution aims to provide seamless tracking, high availability, and scalability to support operational analytics and alerting use cases.

**Acceptance Criteria**

1. **Log Capture**:
   * All user activities passing through the API Gateway are captured in a structured log format (e.g., JSON).
   * Logs include essential metadata such as user ID, timestamp, API endpoint, request method, response status, and latency.
   * Logs are generated with minimal latency and do not impact API Gateway performance.
2. **Kafka Integration**:
   * Logs are published to the EEH Tier 1 Kafka topic in real-time.
   * The Kafka topic is configured with appropriate partitioning and replication for scalability and fault tolerance.
   * Log messages are formatted consistently and include all required metadata.
3. **SDP Cluster Synchronization**:
   * A Kafka consumer framework is implemented to read logs from the EEH Tier 1 Kafka topic.
   * Logs are successfully synced to the SDP cluster with no data loss.
   * The consumer framework handles retries and error scenarios gracefully.
4. **Elasticsearch Integration**:
   * Logs are indexed in Elasticsearch in near real-time.
   * The Elasticsearch index is optimized for search and aggregation queries.
   * Dashboards are created in Elasticsearch (e.g., using Kibana) to visualize key metrics such as API usage, error rates, and latency.
   * Real-time alerts are configured in Elasticsearch for critical events (e.g., high error rates, unusual activity patterns).
5. **Performance and Scalability**:
   * The system handles high throughput of logs without bottlenecks.
   * The pipeline scales horizontally to accommodate increased API traffic.
   * End-to-end latency from log capture to Elasticsearch indexing is within acceptable thresholds (e.g., < 5 seconds).
6. **Monitoring and Reliability**:
   * Monitoring is implemented to track the health of the pipeline (e.g., Kafka lag, consumer errors, Elasticsearch indexing rate).
   * Alerts are set up for pipeline failures or performance degradation.
   * The system ensures no data loss during failures (e.g., Kafka broker downtime, Elasticsearch outages).

**Design Details**

**Architecture Overview**

* **API Gateway**: Configured to generate structured logs for all user activities. Logs are enriched with metadata such as user ID, request details, and response status.
* **Kafka Producer**: A lightweight producer application pushes logs from the API Gateway to the EEH Tier 1 Kafka topic.
* **Kafka (EEH Tier 1 Topic)**: A Kafka topic with multiple partitions and replication factor to ensure scalability and fault tolerance.
* **Kafka Consumer Framework**: A consumer application (e.g., built using Kafka Streams or Spring Kafka) reads logs from the EEH topic and syncs them to the SDP cluster.
* **SDP Cluster**: Hosts a secondary Kafka topic or storage layer to store logs for further processing.
* **Elasticsearch**: Indexes logs for real-time search, analytics, and alerting. Kibana is used for dashboard creation.
* **Monitoring and Alerting**: Tools like Prometheus, Grafana, or Elasticsearch's built-in monitoring are used to track pipeline health and trigger alerts.

**Components**

1. **Log Generation**:
   * API Gateway is configured to emit logs in JSON format.
   * Example log structure:
   * {
   * "user\_id": "12345",
   * "timestamp": "2025-06-20T13:08:00Z",
   * "endpoint": "/api/v1/resource",
   * "method": "POST",
   * "status\_code": 200,
   * "latency\_ms": 120,
   * "client\_ip": "192.168.1.1"
   * }
2. **Kafka Producer**:
   * Built using a Kafka client library (e.g., Confluent Kafka Python, Java Kafka Producer).
   * Configured to batch logs for efficient publishing to the EEH Tier 1 topic.
   * Uses asynchronous publishing to minimize impact on API Gateway performance.
3. **Kafka Topic Configuration**:
   * Topic Name: eeh-tier1-api-logs
   * Partitions: 10 (adjust based on throughput)
   * Replication Factor: 3
   * Retention Period: 7 days (configurable)
4. **Kafka Consumer Framework**:
   * Built using Kafka Streams or Spring Kafka.
   * Reads logs from eeh-tier1-api-logs and forwards them to the SDP cluster.
   * Implements error handling (e.g., retry queues, dead letter queues) for failed messages.
5. **Elasticsearch Integration**:
   * Logs are indexed into an Elasticsearch index (e.g., api-logs-2025).
   * Index mappings are optimized for fields like timestamp, user\_id, and status\_code.
   * Kibana dashboards display metrics such as API call volume, error rates, and average latency.
   * Alerts are configured using Elasticsearch's alerting plugin (e.g., for status\_code >= 500).
6. **Monitoring**:
   * Prometheus and Grafana monitor Kafka consumer lag, producer throughput, and Elasticsearch indexing rate.
   * Alerts are triggered for anomalies (e.g., consumer lag > 1000 messages, indexing failures).

**Implementation Steps**

1. Configure API Gateway to emit structured logs.
2. Develop and deploy Kafka producer to publish logs to EEH Tier 1 topic.
3. Set up Kafka topic with appropriate configurations.
4. Implement Kafka consumer framework to sync logs to SDP cluster.
5. Configure Elasticsearch index and mappings for log storage.
6. Develop Kibana dashboards and configure real-time alerts.
7. Set up monitoring and alerting for the pipeline using Prometheus/Grafana.
8. Test the pipeline for scalability, reliability, and performance under high load.
9. Document the pipeline and provide operational runbooks for maintenance.

**Risks and Mitigations**

* **Risk**: High log volume overwhelms Kafka or Elasticsearch.
  + **Mitigation**: Use partitioning in Kafka and sharding in Elasticsearch to scale horizontally.
* **Risk**: Data loss during Kafka or Elasticsearch outages.
  + **Mitigation**: Configure Kafka with high replication and implement retry mechanisms in the consumer.
* **Risk**: Performance impact on API Gateway due to logging.
  + **Mitigation**: Use asynchronous logging and batching to minimize overhead.

**Estimated Effort**

* **Development**: 4-6 weeks (depending on team size and complexity).
* **Testing**: 2-3 weeks for end-to-end testing, including performance and failure scenarios.
* **Deployment**: 1 week for staged rollout and monitoring setup.