Kafka Consumers Cheat Sheet - Master Level

3.1 Consumer API

KafkaConsumer Basics

Definition KafkaConsumer is a single-threaded, stateful client that maintains partition assignments, offset positions, and broker connections for consuming records from subscribed topics with configurable delivery semantics and performance characteristics. Unlike producers, consumers are NOT thread-safe and require external synchronization or separate consumer instances for concurrent processing, with each consumer maintaining its own connection pool and metadata state.

Key Highlights Each consumer instance maintains subscription state, partition assignment metadata, and current offset positions with automatic broker discovery and metadata refresh capabilities. Consumer instances implement heartbeat protocols with consumer group coordinators to maintain group membership and participate in partition rebalancing during membership changes or failures. Subscription management supports both topic-based subscriptions with automatic partition assignment and manual partition assignment for advanced use cases requiring precise control over partition consumption.

Responsibility / Role Consumers coordinate with consumer group coordinators to maintain group membership, participate in rebalancing protocols, and commit offset positions for progress tracking and failure recovery. They handle record deserialization using pluggable deserializers, manage fetch request optimization with configurable batch sizes and prefetching strategies, and implement session management with heartbeat coordination for failure detection. Critical responsibilities include offset management, partition assignment handling, and graceful shutdown procedures to prevent data loss and minimize rebalancing impact.

Underlying Data Structures / Mechanism Internal architecture maintains SubscriptionState tracking partition assignments and current positions, with Fetcher components managing network requests and record batching from assigned partitions. Consumer coordinator communication uses GroupCoordinator protocol for membership management, rebalancing participation, and offset commit coordination through dedicated broker connections. Metadata management includes topic-partition discovery, broker leadership tracking, and automatic refresh during topology changes with configurable refresh intervals and error handling.

Advantages Automatic partition assignment and rebalancing provide dynamic load distribution and fault tolerance without external coordination systems or manual partition management procedures. Integration with consumer groups enables horizontal scaling of consumption capacity while maintaining ordering guarantees within partition boundaries and exactly-once processing capabilities. Configurable delivery semantics support various consistency models from at-least-once to exactly-once depending on offset commit strategies and application requirements.

Disadvantages / Trade-offs Single-threaded design limits processing parallelism within individual consumer instances, requiring multiple consumer instances or external thread pools for CPU-intensive processing workloads. Consumer group coordination overhead increases with membership changes and can cause temporary consumption pauses during rebalancing operations affecting application latency. Complex failure scenarios including coordinator failures, network partitions, and session timeouts require sophisticated error handling and recovery procedures.

Corner Cases Consumer session timeouts during long-running message processing can trigger unwanted rebalancing and partition reassignment affecting other group members and overall consumption progress. Metadata refresh failures can cause consumers to become temporarily unavailable even when brokers are healthy, requiring careful timeout tuning and retry logic. Consumer close() operations can block indefinitely during network issues, requiring timeout specifications and proper exception handling for graceful shutdown.

Limits / Boundaries Maximum poll interval defaults to 5 minutes (max.poll.interval.ms=300000) limiting message processing time before session timeout and rebalancing triggers. Consumer memory usage scales with assigned partition count and prefetch buffer sizes, typically requiring 32-256MB per consumer depending on configuration and workload characteristics. Session timeout ranges from 6 seconds to 30 minutes balancing failure detection speed with processing time requirements.

Default Values Session timeout is 45 seconds (session.timeout.ms=45000), heartbeat interval is 3 seconds (heartbeat.interval.ms=3000), and max poll interval is 5 minutes (max.poll.interval.ms=300000). Default fetch sizes are 50MB maximum (fetch.max.bytes=52428800) and 1MB default (fetch.min.bytes=1) with 500ms fetch timeout.

Best Practices Use dedicated consumer instances per thread and implement proper shutdown procedures with timeout specifications to prevent resource leaks and rebalancing delays. Monitor consumer lag, processing time, and session timeout metrics to optimize performance and prevent unwanted rebalancing during high-latency processing scenarios. Configure appropriate fetch sizes and poll intervals based on message characteristics and processing requirements, avoiding excessively large batches that can cause memory pressure or session timeouts.

poll() Loop

Definition The poll() loop represents the fundamental consumption pattern where consumers repeatedly invoke poll(timeout) to fetch batches of records from assigned partitions while maintaining heartbeat coordination and session management with consumer group coordinators. This blocking operation serves as the primary mechanism for record retrieval, offset management, rebalancing participation, and consumer liveness indication within the consumer group protocol.

Key Highlights Each poll() invocation handles multiple responsibilities including heartbeat transmission, metadata refresh, rebalancing participation, and actual record fetching with configurable timeout values controlling blocking behavior. The poll mechanism implements prefetching optimization where subsequent poll() calls may return immediately from internal buffers without network requests if sufficient records are already available. Rebalancing operations occur synchronously within poll() calls, temporarily blocking record consumption while partition reassignment and consumer coordination protocols complete.

Responsibility / Role Poll() operations coordinate heartbeat transmission to prevent session timeouts, participate in consumer group rebalancing protocols, and trigger offset commit operations based on auto-commit configuration or manual commit timing. They handle fetch request batching and optimization, managing network resources efficiently while maintaining consumer responsiveness and session health with consumer group coordinators. Critical responsibilities include exception propagation for irrecoverable errors, graceful handling of temporary failures, and ensuring consistent offset advancement during normal processing flows.

Underlying Data Structures / Mechanism Internal poll() implementation uses Fetcher component with prefetch buffers maintaining ConsumerRecord batches per partition, with configurable buffer sizes and fetch optimization strategies. Heartbeat coordination uses separate thread pools with HeartbeatTask scheduling to maintain session health independent of poll() frequency and processing time characteristics. Rebalancing integration uses ConsumerCoordinator protocols executed synchronously within poll() operations, coordinating with other group members through JoinGroup and SyncGroup request cycles.

Advantages Single-threaded poll() design simplifies consumer programming model while automatically handling complex coordination protocols including heartbeat management, rebalancing participation, and offset management without external threading concerns. Prefetching optimization enables high-throughput consumption by maintaining record buffers and reducing network round-trips, while timeout configuration provides flexible blocking behavior for various application patterns. Integrated rebalancing within poll() ensures atomic partition assignment changes and prevents race conditions between consumption and group membership updates.

Disadvantages / Trade-offs Blocking poll() behavior can cause application responsiveness issues during rebalancing operations or network delays, with rebalancing pauses potentially lasting several seconds depending on group size and assignment complexity. Long-running message processing between poll() calls can trigger session timeouts and unwanted rebalancing, requiring careful balance between processing efficiency and heartbeat maintenance. Synchronous rebalancing within poll() creates stop-the-world scenarios affecting all group members, with duration proportional to partition count and assignment strategy complexity.

Corner Cases Empty poll() results during partition assignment changes can indicate rebalancing in progress, requiring application logic to handle temporary consumption interruption and potential duplicate processing after partition reassignment. Network timeouts during poll() can cause IllegalStateException if session expires, requiring consumer restart and potential offset position loss depending on commit timing. Poll timeout of zero provides non-blocking behavior but may increase CPU usage and reduce batching efficiency during low-throughput scenarios.

Limits / Boundaries Poll timeout ranges from 0 (non-blocking) to Long.MAX_VALUE (indefinite blocking) with typical values between 100ms-30 seconds depending on application responsiveness requirements. Maximum records per poll() defaults to 500 (max.poll.records=500) limiting batch size for processing time management and memory usage control. Fetch sizes per poll() are bounded by consumer memory allocation and network timeout configurations, typically 1-50MB depending on message characteristics.

Default Values Default poll timeout varies by application pattern with no enforced default, max poll records is 500 (max.poll.records=500), and fetch configuration includes 50MB maximum (fetch.max.bytes=52428800). Rebalancing timeout within poll() operations defaults to session timeout value (45 seconds) for coordination completion.

Best Practices Implement poll() loops with appropriate timeout values balancing responsiveness with CPU efficiency, typically 100-1000ms for interactive applications and longer timeouts for batch processing scenarios. Handle empty poll() results gracefully and implement proper exception handling for IllegalStateException and other irrecoverable errors requiring consumer restart. Monitor poll() latency and processing time metrics to optimize batch sizes and prevent session timeout scenarios during high-latency processing operations.

Auto vs Manual Commit

Definition Offset commit strategies determine when and how consumer progress is persisted to the __consumer_offsets topic, with automatic commits providing convenience through periodic background commits while manual commits enable precise control over commit timing for advanced delivery semantics. Auto-commit mode uses configurable intervals to commit current positions asynchronously, while manual commit modes provide synchronous and asynchronous APIs for application-controlled commit operations.

Key Highlights Automatic commits occur on configurable intervals (default 5 seconds) during poll() operations, providing at-least-once delivery semantics with potential for duplicate processing during consumer failures or rebalancing scenarios. Manual commits enable exactly-once processing patterns by coordinating offset commits with external transaction systems, database operations, or complex processing workflows requiring atomic operation guarantees. Commit failures in both modes require careful error handling as offset persistence failures can cause processing gaps or duplicate message scenarios during consumer recovery.

Responsibility / Role Auto-commit mode handles offset management transparently with periodic background commits during poll() operations, requiring minimal application logic but providing limited control over commit timing and failure handling. Manual commit strategies enable applications to implement sophisticated delivery semantics including exactly-once processing, transactional coordination with external systems, and custom retry logic for commit failures. Both approaches coordinate with consumer group coordinators for offset persistence and provide mechanisms for handling commit failures and coordinator unavailability.

Underlying Data Structures / Mechanism Auto-commit implementation uses background timers with AutoCommitTask scheduling periodic offset commits based on current consumer positions and configurable commit intervals. Manual commits use OffsetCommitRequest protocols with synchronous or asynchronous variants, supporting both current position commits and explicit offset specification for advanced use cases. Offset commit coordination uses consumer group coordinator communication with retry logic, error classification, and failure propagation through callback mechanisms or exception handling.

Advantages Auto-commit simplifies consumer implementation by eliminating explicit offset management code while providing reasonable at-least-once delivery semantics for most application scenarios. Manual commits enable precise control over delivery semantics, supporting exactly-once processing patterns, external transaction coordination, and sophisticated error handling strategies tailored to application requirements. Both approaches provide progress persistence enabling consumer restart and failure recovery without losing consumption progress or causing excessive duplicate processing.

Disadvantages / Trade-offs Auto-commit can cause message loss during consumer failures if processing completes but commits haven't occurred, and can cause duplicate processing if commits succeed but consumer fails before processing completion. Manual commits require additional application complexity including error handling, retry logic, and careful coordination with message processing workflows to prevent offset/processing inconsistencies. Commit latency increases with manual commits due to synchronous network operations, potentially affecting overall consumption throughput during high-frequency commit scenarios.

Corner Cases Consumer failures between auto-commit intervals can cause duplicate processing of uncommitted messages, with duplicate window size determined by commit interval and processing patterns. Manual commit failures during network partitions or coordinator unavailability can cause consumers to become blocked or require complex error handling to maintain processing progress. Rebalancing during manual commit operations can cause commits to fail with CommitFailedException, requiring careful application logic to handle partition ownership changes.

Limits / Boundaries Auto-commit interval ranges from 1 second to several minutes (default 5 seconds) balancing duplicate processing risk with commit overhead and coordinator load. Manual commit timeout typically matches request timeout (30 seconds default) with configurable values based on network characteristics and coordinator responsiveness patterns. Maximum pending commits per consumer are limited by memory allocation and coordinator capacity, typically hundreds to thousands depending on configuration.

Default Values Auto-commit is enabled by default (enable.auto.commit=true) with 5-second intervals (auto.commit.interval.ms=5000), and manual commit timeout equals request timeout (30 seconds). Retry configuration for commits follows general consumer retry settings with exponential backoff and maximum retry limits.

Best Practices Use auto-commit for simple at-least-once processing scenarios with idempotent message handling, implementing proper duplicate detection and processing logic to handle rebalancing scenarios. Choose manual commits for exactly-once requirements or when coordinating with external transactional systems, implementing comprehensive error handling and retry logic for commit failures. Monitor commit success rates and latency metrics to identify coordinator issues or network problems affecting offset persistence reliability and consumer recovery capabilities.

3.2 Consumer Groups

Rebalancing Strategies (Range, Round Robin, Sticky, Cooperative)

Definition Rebalancing strategies are pluggable algorithms that determine partition assignment distribution among consumer group members during membership changes, with different strategies optimizing for various criteria including load balance, assignment stability, and rebalancing efficiency. These strategies range from simple mathematical distribution (Range, Round Robin) to sophisticated optimization algorithms (Sticky, Cooperative) that minimize reassignment overhead and processing disruption.

Key Highlights Range strategy assigns consecutive partitions per topic to individual consumers providing simple load distribution but potential imbalance with multiple topics, while Round Robin distributes all partitions across consumers regardless of topic boundaries for optimal balance. Sticky assignment attempts to preserve existing assignments during rebalancing to minimize processing disruption and state transfer overhead, while Cooperative rebalancing enables incremental assignment changes without stopping all consumers simultaneously. Strategy selection significantly impacts rebalancing duration, assignment fairness, and processing continuity during consumer group membership changes.

Responsibility / Role Assignment strategies coordinate with consumer group protocols during JoinGroup and SyncGroup phases to distribute partitions optimally based on consumer capacity, partition count, and historical assignment patterns. They handle edge cases including uneven partition/consumer ratios, consumer failure scenarios, and topic partition count changes while maintaining fairness and minimizing reassignment overhead. Critical responsibilities include assignment stability during minor membership changes, load balancing across heterogeneous consumer instances, and integration with consumer session management and heartbeat protocols.

Underlying Data Structures / Mechanism Strategy implementations receive consumer group membership lists and topic-partition metadata during rebalancing operations, using various algorithms to compute optimal assignments with different optimization criteria. Range strategy uses modulo arithmetic for per-topic partition distribution, while Round Robin implements global partition sorting and round-robin assignment across all consumers. Sticky strategies maintain assignment history and use constraint satisfaction algorithms to minimize changes, while Cooperative protocols use incremental assignment updates with multi-phase rebalancing coordination.

Advantages Range strategy provides predictable assignment patterns suitable for stateful processing where partition locality matters, while Round Robin ensures optimal load distribution across consumers with multiple topics. Sticky assignment reduces processing disruption by preserving existing assignments and minimizing state transfer requirements during minor membership changes, improving application performance and reducing resource usage. Cooperative rebalancing enables rolling consumer updates and eliminates stop-the-world rebalancing pauses, significantly reducing application unavailability during membership changes.

Disadvantages / Trade-offs Range assignment can create significant load imbalance with multiple topics having different partition counts, potentially overwhelming subset of consumers while underutilizing others. Round Robin assignment may cause related partitions to be assigned to different consumers, complicating stateful processing and increasing cross-consumer coordination requirements. Sticky and Cooperative strategies add computational complexity during rebalancing operations and may require more sophisticated consumer coordination protocols increasing rebalancing latency.

Corner Cases Partition count changes during runtime can defeat sticky assignment benefits, causing complete reassignment and eliminating assignment stability advantages during topic scaling operations. Consumer capacity heterogeneity can cause suboptimal assignments with equal partition distribution strategies, requiring custom assignment strategies for workload-specific optimization. Assignment strategy changes during consumer group lifecycle require complete rebalancing and loss of assignment history for sticky strategies.

Limits / Boundaries Maximum consumers per group is limited by partition count for optimal parallelism, with excess consumers remaining idle until membership changes or partition additions occur. Assignment computation complexity increases with consumer count and partition count, with sticky strategies potentially requiring seconds for large groups (hundreds of consumers, thousands of partitions). Rebalancing coordination timeout defaults to session timeout (45 seconds) limiting time available for complex assignment calculations and protocol coordination.

Default Values Default assignment strategy is Range (partition.assignment.strategy=org.apache.kafka.clients.consumer.RangeAssignor) with cooperative rebalancing disabled by default in older versions. Rebalancing timeout equals session timeout (45 seconds), and assignment strategies can be configured as ordered lists for fallback behavior.

Best Practices Choose Round Robin or Sticky strategies for balanced load distribution across multiple topics, use Range strategy when partition locality is important for stateful processing or external system coordination. Enable Cooperative rebalancing for production deployments to eliminate rebalancing downtime and improve consumer group availability during membership changes. Monitor assignment distribution and rebalancing frequency to identify optimization opportunities and potential issues with consumer capacity or partition distribution patterns.

Static Membership

Definition Static membership enables consumers to maintain persistent identities across restarts and temporary disconnections using group.instance.id configuration, preventing unnecessary rebalancing operations during planned maintenance, deployments, or brief network interruptions. This feature allows consumer group coordinators to distinguish between temporary consumer unavailability and permanent member departure, optimizing rebalancing behavior for predictable infrastructure patterns and reducing processing disruption.

Key Highlights Static members retain partition assignments during session timeout periods up to configurable maximum duration (session.timeout.ms), enabling quick recovery without rebalancing when consumers reconnect within timeout windows. Consumer instances with identical group.instance.id are treated as the same logical member regardless of network identity, enabling seamless failover during consumer restarts, deployments, or infrastructure maintenance procedures. Static membership works with all rebalancing strategies and provides additional assignment stability benefits when combined with sticky assignment algorithms.

Responsibility / Role Static membership coordination requires consumer group coordinators to maintain member identity mappings and delay rebalancing operations during temporary member unavailability until configured timeout periods expire. Consumers must implement proper session management including graceful shutdown procedures and rapid restart capabilities to maximize static membership benefits and minimize rebalancing windows. Critical responsibilities include group.instance.id uniqueness management, session timeout tuning for infrastructure patterns, and coordination with deployment automation to minimize rebalancing frequency.

Underlying Data Structures / Mechanism Consumer group coordinators maintain member identity registries mapping group.instance.id values to partition assignments and session state, with timeout tracking for delayed rebalancing decisions. Static member session management uses extended timeout windows with heartbeat protocols modified to account for planned disconnection scenarios and rapid reconnection patterns. Assignment preservation during temporary unavailability requires coordinator state persistence and careful timeout management to balance availability with assignment stability.

Advantages Significant reduction in rebalancing frequency during planned maintenance, deployments, and infrastructure changes eliminates processing disruption and improves application availability during operational activities. Preserved partition assignments enable stateful consumers to resume processing immediately without state rebuild, cache warming, or external system reconnection overhead that normally accompanies partition reassignment. Reduced coordinator load and network traffic during membership changes improves overall cluster efficiency and reduces operational overhead for large consumer groups.

Disadvantages / Trade-offs Extended timeout periods can delay legitimate rebalancing when consumers permanently fail, potentially causing processing delays and reduced parallelism until timeout expiration triggers rebalancing. group.instance.id management adds operational complexity requiring unique identifier coordination across consumer instances and careful planning for scaling and deployment scenarios. Static membership prevents dynamic load balancing benefits during heterogeneous consumer performance scenarios where partition reassignment could improve overall processing efficiency.

Corner Cases Duplicate group.instance.id values cause consumer conflicts and assignment issues requiring careful coordination between deployment systems and consumer configuration management. Network partitions during static member timeout windows can cause extended processing delays as coordinators wait for timeout expiration before triggering rebalancing. Consumer restarts with different group.instance.id values lose static membership benefits and trigger immediate rebalancing despite being same logical consumer instance.

Limits / Boundaries Static member timeout periods range from session timeout (45 seconds default) to several minutes depending on infrastructure characteristics and maintenance window requirements. Maximum concurrent static members per group is limited by coordinator memory capacity and partition assignment tracking overhead, typically thousands of members depending on cluster configuration. group.instance.id string length and character restrictions follow Kafka naming conventions with practical limits around 255 characters.

Default Values Static membership is disabled by default (no group.instance.id configured), session timeout for static members typically extends to 5-15 minutes for deployment scenarios. Rebalancing delay equals session timeout value when static membership is enabled, requiring explicit timeout configuration for optimal behavior.

Best Practices Configure static membership for predictable consumer deployments with known restart patterns, using deployment automation to coordinate group.instance.id assignment and avoid conflicts during parallel deployments. Set session timeouts based on typical restart and maintenance window durations, balancing static membership benefits with failure detection requirements for permanent consumer failures. Monitor rebalancing frequency and assignment stability metrics to validate static membership effectiveness and identify opportunities for timeout optimization or deployment procedure improvements.

3.3 Offset Management

__consumer_offsets Topic

Definition The __consumer_offsets topic is an internal, compacted Kafka topic that stores consumer group offset positions, group metadata, and coordinator assignment information as key-value pairs with consumer group coordination serving as the offset storage and retrieval mechanism. This system topic uses partition assignment based on consumer group ID hashing to distribute offset management load across multiple consumer coordinators and provides durable offset storage with configurable retention and cleanup policies.

Key Highlights Offset storage uses compacted topic structure with consumer group and partition combinations as keys, enabling automatic cleanup of obsolete offset information while retaining latest positions for active consumer groups. Partition assignment for offset storage uses hash(group.id) % __consumer_offsets.partitions calculation, distributing coordinator responsibilities and offset storage load across cluster brokers for scalability. Topic configuration includes accelerated cleanup policies with 24-hour retention for offset cleanup and specialized compression settings optimized for small key-value offset records.

Responsibility / Role Consumer coordinators manage offset commit and retrieval operations by reading and writing to assigned __consumer_offsets partitions, handling offset validation, duplicate detection, and concurrent access coordination during consumer group operations. The topic serves as authoritative storage for consumer progress tracking, enabling consumer restart and failure recovery scenarios while providing audit trails for consumption patterns and group coordination activities. Critical responsibilities include offset conflict resolution during coordinator failover, retention management for obsolete consumer groups, and integration with consumer group membership protocols.

Underlying Data Structures / Mechanism Internal topic structure uses binary key-value encoding with consumer group ID, topic name, and partition number forming composite keys for offset record identification and retrieval optimization. Compaction algorithms maintain only latest offset values per consumer group-partition combination while preserving group membership metadata and coordinator assignment information for active groups. Offset record format includes timestamp information, metadata versioning, and optional commit information for debugging and audit purposes with backward compatibility across protocol versions.

Advantages Centralized offset storage provides consistent consumer progress tracking across consumer restarts, failures, and rebalancing scenarios without external dependencies or complex coordination protocols. Automatic cleanup through compaction eliminates operational overhead for offset management while maintaining indefinite retention for active consumer groups and automatic cleanup for obsolete groups. Scalable partition-based storage distributes coordinator load and enables horizontal scaling of offset management capacity across cluster growth and increasing consumer group counts.

Disadvantages / Trade-offs Offset topic availability directly affects consumer group functionality, with __consumer_offsets unavailability preventing new consumer group formation, offset commits, and coordinator assignment for affected partitions. Compaction lag during high offset commit rates can cause disk usage growth and potentially affect cluster storage capacity, requiring monitoring and tuning of cleanup policies and compaction frequency. Coordinator assignment changes during broker failures can cause temporary offset commit unavailability until coordinator reassignment completes and offset topic partitions recover.

Corner Cases Offset topic corruption or data loss requires manual recovery procedures and may cause consumer groups to reset to configured auto.offset.reset policies, potentially causing duplicate processing or data loss depending on configuration. Partition reassignment of __consumer_offsets during cluster maintenance can cause temporary consumer group unavailability and coordinator failover affecting all consumer groups assigned to reassigned partitions. Extremely high offset commit rates can overwhelm compaction processes and cause disk space exhaustion requiring throttling or configuration adjustments.

Limits / Boundaries Default configuration creates 50 partitions for __consumer_offsets topic with 3x replication factor, supporting thousands of consumer groups per partition depending on commit frequency and group activity patterns. Offset record size is typically 200-500 bytes including group metadata and coordination information, with retention policies defaulting to 24 hours for offset cleanup and indefinite retention for active offsets. Maximum consumer groups per cluster is primarily limited by __consumer_offsets partition count and coordinator capacity rather than absolute limits.

Default Values Offset topic partition count defaults to 50 (offsets.topic.num.partitions=50), replication factor is 3 (offsets.topic.replication.factor=3), and cleanup policy is compact with 24-hour retention (offsets.retention.minutes=10080). Segment size is 100MB (offsets.topic.segment.bytes=104857600) with cleanup frequency every 15 minutes.

Best Practices Monitor __consumer_offsets topic health including partition distribution, compaction effectiveness, and disk usage patterns as critical cluster health indicators affecting all consumer group functionality. Configure appropriate partition count based on expected consumer group count and commit frequency, typically 1-5 consumer groups per partition for optimal coordinator load distribution. Implement proper backup and disaster recovery procedures for __consumer_offsets topic as consumer group state loss requires complete consumer group reset and potential processing impact.

Reset Policies (Earliest, Latest)

Definition Reset policies define consumer behavior when no committed offsets exist for assigned partitions or when committed offsets are out of range due to data retention policies, broker failures, or partition changes. The auto.offset.reset configuration determines whether consumers begin consumption from partition beginning (earliest), end (latest), or fail with exception (none) when offset positions cannot be determined from __consumer_offsets topic.

Key Highlights Earliest policy positions consumers at partition beginning enabling complete data processing but potentially consuming large volumes of historical data during initial startup or offset reset scenarios. Latest policy positions consumers at current partition end eliminating historical data processing overhead but potentially missing records produced between consumer shutdown and restart periods. None policy throws NoOffsetForPartitionException requiring explicit application handling for offset reset scenarios, enabling custom offset management strategies and preventing unintended data processing behavior.

Responsibility / Role Reset policies coordinate with offset management systems during consumer startup, coordinator failures, and partition reassignment scenarios to establish initial consumption positions when committed offsets are unavailable or invalid. They provide default behavior for consumer group initialization, disaster recovery scenarios, and consumer development/testing environments where offset history may be unavailable or unreliable. Critical responsibilities include preventing unintended data processing during consumer group setup and providing predictable behavior during offset range violations and retention-related offset loss.

Underlying Data Structures / Mechanism Reset policy implementation uses OffsetFetcher coordination with broker APIs to determine partition beginning and end positions when committed offsets are unavailable or out of valid range. Position resolution involves ListOffsets requests to partition leaders for earliest/latest offset determination and validation against retention boundaries and partition availability. Exception handling for reset policy violations integrates with consumer coordinator protocols and provides detailed error information for application-level offset management decisions.

Advantages Earliest reset enables comprehensive data processing and replay capabilities supporting audit scenarios, data migration, and complete event stream processing without data loss during consumer initialization. Latest reset provides immediate processing of new data without historical overhead, optimal for real-time processing scenarios and reducing consumer startup latency in high-throughput environments. None policy enables sophisticated application-level offset management with custom reset logic, external offset storage, and precise control over consumption boundaries during various failure scenarios.

Disadvantages / Trade-offs Earliest reset can cause excessive processing overhead and extended startup times when partitions contain large volumes of historical data, potentially overwhelming downstream systems during consumer initialization. Latest reset may cause data loss if consumers miss records produced during downtime periods, requiring careful coordination with producer patterns and availability requirements. None policy requires complex application logic for offset reset handling and can cause consumer failures during common scenarios like retention-based offset expiration.

Corner Cases Partition retention policies can invalidate committed offsets between consumer sessions, triggering reset policy behavior unexpectedly and potentially causing data loss or duplicate processing depending on policy configuration. Consumer group coordinator changes can temporarily make offset information unavailable, triggering reset policies even when valid offsets exist in __consumer_offsets topic. Reset policy evaluation during rebalancing can cause different consumers to receive different starting positions for same partitions depending on timing and coordinator availability.

Limits / Boundaries Reset policy evaluation occurs during partition assignment and cannot be changed dynamically without consumer restart, requiring careful initial configuration based on application requirements and data processing patterns. Earliest reset processing time depends on partition data volume and consumer processing capacity, potentially requiring hours or days for historical data processing in high-volume scenarios. Latest reset may skip significant amounts of data depending on retention policies and consumer downtime duration, requiring monitoring and alerting for data gap detection.

Default Values Default reset policy is latest (auto.offset.reset=latest), consumer startup timeout for offset resolution is 60 seconds, and reset policy evaluation occurs during each partition assignment operation. Exception handling for invalid reset policies follows general consumer error propagation patterns with detailed error messages.

Best Practices Configure reset policies based on application data processing requirements and tolerance for data loss versus processing overhead, using earliest for audit scenarios and latest for real-time processing requirements. Monitor offset validity and retention policy interactions to prevent unexpected reset policy triggers, implementing alerting for offset range violations and partition retention events. Design applications to handle reset policy scenarios gracefully including data validation, processing time estimation, and downstream system coordination during initial startup or recovery scenarios.

Lag Monitoring

Definition Consumer lag represents the difference between current partition end offsets (log-end-offset) and consumer committed positions, providing critical metrics for consumption performance monitoring, capacity planning, and operational alerting for consumer group health. Lag monitoring involves tracking per-partition lag values, aggregate consumer group lag, and lag trends over time to identify processing bottlenecks, capacity issues, and consumer group operational problems.

Key Highlights Lag calculation requires coordination between consumer position information from __consumer_offsets topic and current partition end offsets from broker metadata, with measurements affected by consumer commit frequency and offset commit strategies. Real-time lag monitoring uses JMX metrics, consumer coordinator APIs, and external monitoring systems to track lag trends, detect processing bottlenecks, and trigger operational alerts for consumer group performance issues. Lag interpretation requires understanding of producer patterns, partition assignment distribution, and consumer processing characteristics to distinguish between operational issues and normal processing variations.

Responsibility / Role Consumer lag monitoring systems coordinate with consumer coordinators and broker metadata to collect lag measurements across consumer group members and partition assignments for comprehensive group performance visibility. They provide early warning systems for consumer group capacity issues, processing bottlenecks, and infrastructure problems affecting consumption rates through automated alerting and trend analysis capabilities. Critical responsibilities include lag threshold management, false positive prevention during rebalancing scenarios, and integration with capacity planning systems for consumer group scaling decisions.

Underlying Data Structures / Mechanism Lag calculation uses consumer coordinator APIs to retrieve committed offset positions from __consumer_offsets topic and broker metadata APIs for current partition end offsets with timestamp coordination for accurate lag measurements. Monitoring systems implement caching and batching strategies to reduce overhead of frequent lag calculations while maintaining measurement accuracy and responsiveness for operational alerting requirements. Metric aggregation includes per-partition, per-consumer, and per-group lag calculations with historical trending and statistical analysis capabilities for performance optimization and capacity planning.

Advantages Comprehensive lag visibility enables proactive identification of consumer group performance issues, capacity constraints, and processing bottlenecks before they affect application SLAs or downstream system dependencies. Automated lag monitoring provides operational alerting for consumer failures, processing slowdowns, and capacity planning requirements without manual monitoring overhead or delayed problem detection. Integration with consumer group metadata enables correlation between lag trends and consumer membership changes, rebalancing events, and infrastructure modifications for root cause analysis.

Disadvantages / Trade-offs Frequent lag monitoring can create additional load on consumer coordinators and broker metadata systems, potentially affecting cluster performance during high-frequency monitoring scenarios with large numbers of consumer groups. Lag measurements during rebalancing periods can show misleading values due to partition reassignment and consumer restart scenarios, requiring sophisticated filtering and trend analysis to prevent false alerts. Complex lag interpretation requires understanding of application processing patterns, producer behavior, and infrastructure characteristics that may not be captured in basic lag metrics.

Corner Cases Consumer group rebalancing can cause temporary lag spikes as partitions reassign and consumers restart processing, requiring lag monitoring systems to account for rebalancing events in alerting thresholds and trend analysis. Producer burst patterns can create lag spikes that appear as consumer performance issues but reflect normal workload variations requiring correlation with producer metrics for accurate interpretation. Offset commit failures can cause lag measurements to show stale values not reflecting actual consumer processing progress, requiring coordination with commit success rates for accurate monitoring.

Limits / Boundaries Lag monitoring frequency is limited by broker metadata refresh rates and consumer coordinator response capacity, typically ranging from seconds to minutes depending on cluster size and monitoring system architecture. Maximum useful lag measurement horizon depends on partition retention policies and data volume characteristics, with measurements becoming less meaningful for historical data beyond typical processing windows. Consumer group scale affects monitoring system requirements with large deployments requiring hundreds or thousands of consumer groups requiring efficient lag collection and aggregation strategies.

Default Values Consumer lag monitoring typically uses 30-60 second intervals for routine monitoring with faster intervals (5-15 seconds) for critical consumer groups, and alerting thresholds vary based on application SLAs and processing patterns. JMX metrics provide real-time lag information with consumer-records-lag-max and consumer-records-lag-avg measurements, and consumer coordinator APIs enable external monitoring system integration.

Best Practices Establish lag monitoring baselines based on normal processing patterns and producer behavior to set appropriate alerting thresholds and prevent false positive alerts during normal workload variations. Implement lag trending and historical analysis to identify gradual performance degradation and capacity planning requirements before critical thresholds are reached. Correlate lag monitoring with consumer group membership changes, processing time metrics, and infrastructure events to enable rapid root cause identification and resolution during consumer group performance issues.