Kafka Fundamentals Cheat Sheet - Master Level

1.1 Core Concepts

Topics, Partitions, Offsets

Definition Topics are logical categories that represent unbounded, immutable event streams with configurable durability and ordering semantics. Partitions are the fundamental unit of parallelism and data distribution, implementing a segmented commit log with monotonically increasing offset identifiers. Offsets serve as unique sequence numbers within each partition, enabling precise positioning for replay, exactly-once processing, and consumer coordination protocols.

Key Highlights Each topic can scale to thousands of partitions across multiple brokers, with partition-level ordering guarantees but no cross-partition ordering. Offsets are 64-bit long integers starting from 0, with gaps possible in compacted topics after log cleaning. Partition assignment uses consistent hashing or custom partitioners, and partition count changes trigger consumer group rebalancing across all subscribers.

Responsibility / Role Topics define retention policies (time-based, size-based, or compacted), replication factors, and cleanup policies that govern data lifecycle management. Partitions distribute data across brokers for horizontal scaling and fault tolerance, while maintaining leader-follower replication protocols. Offsets enable consumer groups to implement various delivery semantics (at-most-once, at-least-once, exactly-once) and support complex stream processing patterns.

Underlying Data Structures / Mechanism Each partition maintains multiple segment files (typically 1GB each) with corresponding index files (.index), timestamp index files (.timeindex), and transaction index files (.txnindex). The log cleaner uses a two-phase compaction algorithm with dirty and clean sections, maintaining key-based deduplication while preserving message ordering. Offset management uses high-water marks and log-end offsets to coordinate replication, with in-sync replicas (ISR) determining partition availability during broker failures.

Advantages Partitioning enables linear scalability with consumer parallelism matching partition count, supporting millions of messages per second across distributed clusters. Immutable append-only logs provide strong durability guarantees and enable time-travel queries for debugging and reprocessing. Log compaction allows indefinite key-based state storage with automatic cleanup, supporting event sourcing and state store patterns efficiently.

Disadvantages / Trade-offs Partition count cannot be decreased once set, and over-partitioning creates metadata overhead and reduces throughput efficiency. Cross-partition transactions require complex coordination protocols with performance penalties. Partition reassignment causes temporary unavailability and increased network I/O, while uneven partition distribution creates broker hotspots and resource imbalance.

Corner Cases Empty partitions consume approximately 1MB memory per partition on each broker, limiting maximum partition density. Compacted topics with frequent key updates can trigger excessive log cleaning overhead, impacting broker performance. Partition leadership changes during network partitions can cause temporary message loss if unclean.leader.election.enable=true.

Limits / Boundaries Maximum message size can theoretically reach 1GB (Integer.MAX_VALUE) but practical limits are 10-100MB due to memory and network constraints. Partition count per broker should not exceed 4,000 in production, with total cluster partition limit around 200,000. Topic names are limited to 249 characters with restrictions on special characters (dots, underscores allowed, others reserved).

Default Values Default partition count is 1, segment size is 1GB (1073741824 bytes), retention is 7 days (168 hours), and replication factor is 1. Default cleanup policy is "delete" with log.cleaner.enable=false, and segment rolling occurs every 7 days or when size limit reached.

Best Practices Size partitions for 25MB-1GB throughput per consumer, typically 10-100 partitions per topic for optimal performance. Use keyed messages judiciously to maintain partition balance while preserving ordering semantics. Monitor partition skew using JMX metrics and implement custom partitioners for workloads with known key distribution patterns.

Producers, Consumers, Consumer Groups

Definition Producers are client applications implementing the Kafka protocol to publish records with configurable delivery semantics, batching strategies, and partitioning logic. Consumers are stateful clients that maintain offset positions and implement various consumption patterns (polling, streaming, batch processing). Consumer groups provide distributed coordination through the group coordination protocol, enabling automatic partition assignment, failure detection, and rebalancing with pluggable assignment strategies.

Key Highlights Producers support idempotent and transactional writes with sequence numbering to prevent duplicates and enable exactly-once semantics. Consumer groups implement cooperative rebalancing protocols (incremental cooperative rebalancing in newer versions) to minimize disruption during member changes. Each partition can only be consumed by one member within a consumer group, but multiple groups can consume the same partition independently.

Responsibility / Role Producers handle record serialization, compression, batching, and retry logic while managing connection pools to broker leaders. They implement custom partitioning logic, handle broker metadata discovery, and coordinate transactional boundaries across multiple topic-partitions. Consumer groups manage offset commits to __consumer_offsets topic, coordinate partition assignment through group coordinators, and implement session management with heartbeat protocols for failure detection.

Underlying Data Structures / Mechanism Producers maintain in-memory record batches per partition with configurable batch size and linger time, using compression algorithms (GZIP, Snappy, LZ4, ZSTD) to optimize network utilization. Consumer group coordination uses a state machine with states (Dead, Initializing, Rebalancing, Stable) and implements join/sync protocols for partition assignment. Offset management uses a compacted internal topic (__consumer_offsets) with configurable retention and cleanup policies.

Advantages Producer batching and compression can achieve 10x throughput improvements over individual sends, with async callbacks enabling high-throughput pipelined operations. Consumer groups provide automatic fault tolerance and load balancing without external coordination systems. Transactional producers enable exactly-once processing across multiple partitions and external systems through two-phase commit protocols.

Disadvantages / Trade-offs Producer batching introduces latency trade-offs, with linger.ms creating artificial delays to improve throughput efficiency. Consumer group rebalancing causes stop-the-world pauses affecting all group members, with duration proportional to partition count and assignment complexity. Transactional

processing reduces throughput by 20-30% due to additional coordination overhead and two-phase commit protocols.

Corner Cases Producer retries can cause message reordering unless max.in.flight.requests.per.connection=1, significantly reducing throughput. Consumer groups with frequent membership changes can enter rebalancing loops, effectively stopping all consumption. Manual offset management requires careful coordination to prevent data loss or duplicate processing during consumer failures.

Limits / Boundaries Producer batch size defaults to 16KB but can be configured up to available heap memory, with practical limits around 1-16MB. Consumer groups support up to several hundred members, limited by rebalancing complexity and coordinator capacity. Session timeout ranges from 6 seconds to 30 minutes, balancing failure detection speed with stability during processing delays.

Default Values Producer acknowledgments default to 1 (leader only), batch size is 16384 bytes, linger time is 0ms, and retries are Integer.MAX_VALUE. Consumer session timeout is 45 seconds (increased in recent versions), heartbeat interval is 3 seconds, and max poll interval is 5 minutes.

Best Practices Configure producer acks=all with idempotence for durability, tune batch size and linger time based on latency requirements versus throughput goals. Size consumer groups equal to partition count for maximum parallelism, implement graceful shutdown handlers for clean offset commits. Use transactional producers only when exactly-once semantics are required, as performance overhead is significant.

Brokers and Clusters

Definition Brokers are JVM-based server processes that implement the Kafka protocol, manage partition replicas, and coordinate distributed operations through controller election and metadata synchronization. Clusters represent collections of brokers that collectively provide distributed storage, replication, and fault tolerance through configurable consistency models and partition placement strategies.

Key Highlights Each broker maintains multiple thread pools (network threads, I/O threads, request handler threads) to handle concurrent client connections and replica synchronization. Clusters implement controller election through ZooKeeper or KRaft consensus, with the controller managing partition leadership, replica assignment, and cluster metadata distribution. Brokers participate in in-sync replica sets (ISR) to provide configurable consistency guarantees and availability trade-offs.

Responsibility / Role Brokers handle all client requests (produce, fetch, metadata, admin), manage local disk storage with configurable I/O patterns, and participate in replication protocols with follower lag monitoring. Controllers coordinate cluster-wide operations including partition reassignment, preferred leader election, topic creation/deletion, and configuration changes. Each broker exposes JMX metrics for monitoring throughput, latency, resource utilization, and replication health.

Underlying Data Structures / Mechanism Brokers use memory-mapped files and page cache optimization for efficient disk I/O, with configurable flush policies and file system selection. The controller maintains cluster state in ZooKeeper znodes or KRaft metadata log, using versioned updates and watches/listeners for change propagation. Replica synchronization uses high-water marks and leader epochs to ensure consistency during leadership changes and network partitions.

Advantages Multi-broker clusters provide horizontal scaling of both storage capacity and network throughput beyond single-machine limitations. Automatic partition leadership redistribution and replica

replacement enable seamless broker maintenance and failure recovery. Page cache utilization and zero-copy transfers optimize disk I/O and network performance for sustained high-throughput workloads.

Disadvantages / Trade-offs Adding brokers requires manual partition reassignment to achieve load balancing, with significant network overhead during migration. Cluster coordination overhead increases with broker count, affecting metadata operation latency and controller scalability. GC pressure from large heap sizes (>8GB) can cause stop-the-world pauses affecting request processing and replication protocols.

Corner Cases Unclean leader election (unclean.leader.election.enable=true) can cause data loss when ISR is empty but may improve availability during cascading failures. Network partitions can cause controller isolation and temporary cluster unavailability, requiring careful tuning of session timeouts and retry policies. Disk failures on single-disk brokers can cause permanent data loss for under-replicated partitions.

Limits / Boundaries Recommended maximum partition count per broker is 4,000, with total cluster capacity around 200,000 partitions before coordination bottlenecks emerge. JVM heap sizing typically ranges from 4-8GB with recommendation not exceeding 12GB for optimal GC performance. Network connection limits require tuning operating system file descriptor limits to 100,000+ for high-concurrency workloads.

Default Values Default replication factor is 1 (no fault tolerance), broker heap is 1GB, and network threads are 3 with 8 I/O threads. Socket buffer sizes default to 100KB for send/receive operations, and controller socket timeout is 30 seconds.

Best Practices Deploy clusters with odd broker counts (3, 5, 7) for optimal quorum-based decisions and maintain separate disk volumes for logs and OS. Monitor controller failover frequency, partition distribution balance, and under-replicated partition counts as key health indicators. Implement rack awareness for replica placement to survive entire rack failures in data center environments.

Records (Key, Value, Headers, Timestamp)

Definition Records are immutable data structures representing individual events with optional key for partitioning, required value payload, extensible headers for metadata, and timestamps supporting temporal processing patterns. These atomic units support various serialization formats, compression codecs, and schema evolution strategies while maintaining backwards compatibility through versioned record formats.

Key Highlights Keys determine partition placement using configurable hash functions or custom partitioners, enabling co-location of related events for stateful processing. Values support pluggable serialization including Avro, Protobuf, JSON, and custom formats with schema registry integration for evolution management. Headers provide extensible key-value metadata without affecting partition assignment, supporting routing, tracing, and processing hints.

Responsibility / Role Keys enable log compaction by serving as deduplication identifiers and ensure related events maintain ordering within partitions. Values carry business data with serialization handling data type conversion, compression, and schema evolution across producer/consumer versions. Headers store processing metadata, correlation IDs, content types, and routing information without impacting core partitioning logic.

Underlying Data Structures / Mechanism Records use binary wire format with magic bytes for version identification, CRC32C checksums for corruption detection, and variable-length encoding for efficient space utilization. Timestamp handling supports both CreateTime (producer timestamp) and LogAppendTime (broker

timestamp) with configurable policies per topic. Compression operates at batch level with support for GZIP, Snappy, LZ4, and ZSTD algorithms with different compression/speed trade-offs.

Advantages Flexible serialization enables polyglot environments with different data formats per application while maintaining protocol compatibility. Header extensibility supports evolving processing requirements and integration patterns without breaking existing consumers. Timestamp precision enables time-based windowing, late data handling, and temporal join operations in stream processing frameworks.

Disadvantages / Trade-offs Null keys prevent log compaction benefits and cause round-robin distribution potentially creating partition skew with uneven processing loads. Large header collections increase per-record overhead and network utilization, reducing effective throughput for small message workloads. Timestamp accuracy depends on producer clock synchronization and network latency, affecting temporal processing precision.

Corner Cases Records with identical keys but different timestamps may appear out-of-order due to producer retries, network delays, or broker failover scenarios. Header serialization requires application-level coordination as Kafka treats headers as opaque byte arrays without built-in schema support. Compression effectiveness varies significantly based on message patterns, with structured data achieving 70-90% reduction while binary data may show minimal improvement.

Limits / Boundaries Maximum record size can theoretically reach 1GB (Integer.MAX_VALUE bytes) but practical production limits are 10-100MB due to heap memory and network timeout constraints. Header keys are limited to UTF-8 strings while values must be byte arrays, with total header size contributing to overall record size limits. Individual batch compression cannot exceed available producer heap memory during compression operations.

Default Values Records without explicit keys use null values distributed round-robin across partitions, and timestamps default to CreateTime with producer system clock values. Header collections initialize as empty maps, and default serializers handle primitive types (String, Long, ByteArray) without external dependencies.

Best Practices Design key schemas for even distribution across partitions while maintaining logical grouping for stateful operations, avoiding high-cardinality keys that create hotspots. Use headers sparingly for routing and metadata, implementing consistent serialization strategies across producer applications. Configure compression based on message characteristics and network capacity, with LZ4 offering optimal speed/compression balance for most workloads.

1.2 Setup & Architecture

Kafka Broker Configuration

Definition Broker configuration encompasses over 200 parameters controlling memory allocation, disk I/O patterns, network handling, replication behavior, security policies, and operational characteristics that directly impact cluster performance, reliability, and resource utilization patterns. These configurations range from JVM tuning parameters affecting garbage collection to protocol-level settings governing client interactions and inter-broker coordination.

Key Highlights Configuration parameters are categorized into static (requiring restart) and dynamic (updatable via Admin API) settings, with cluster-wide, broker-level, and topic-level override hierarchies. Critical performance settings include memory allocation ratios, thread pool sizing, batch processing parameters, and

timeout values that must be coordinated across cluster members. Security configurations control authentication mechanisms (SASL, SSL), authorization policies (ACLs), and encryption settings for data in transit and at rest.

Responsibility / Role Memory configurations manage heap allocation, page cache utilization, and off-heap storage for optimal I/O performance with large message workloads. Network settings control connection limits, socket buffer sizes, and request processing parallelism to handle thousands of concurrent client connections. Replication parameters govern ISR management, leader election policies, and data consistency guarantees during broker failures and network partitions.

Underlying Data Structures / Mechanism JVM configurations affect garbage collection algorithms (G1, CMS, Parallel) with heap sizing impacting pause times and throughput characteristics. Log directories support JBOD (Just a Bunch of Disks) configurations with automatic failure handling and partition distribution across available volumes. Dynamic configurations are stored in ZooKeeper znodes or KRaft metadata logs with version tracking and change notification mechanisms.

Advantages Extensive configuration options enable optimization for diverse workload patterns, hardware configurations, and operational requirements without code modifications. Dynamic reconfiguration capabilities allow performance tuning and policy updates without cluster downtime or service interruption. Per-topic configuration overrides provide fine-grained control over retention, replication, and performance characteristics for different data streams.

Disadvantages / Trade-offs Configuration complexity requires deep understanding of interdependencies between parameters, with incorrect settings potentially causing severe performance degradation or instability. Some critical parameters require coordinated updates across all cluster members, complicating rolling updates and operational procedures. Configuration drift across brokers can create subtle inconsistencies and hard-to-diagnose performance issues in production environments.

Corner Cases Memory configurations exceeding available system resources can trigger OOM kills or severe GC thrashing, effectively stopping broker processing. Network timeout settings that are too aggressive can cause false failure detection and unnecessary rebalancing during temporary network congestion. Log flush configurations that are too frequent can overwhelm disk I/O capacity, while infrequent flushing increases data loss risk during sudden failures.

Limits / Boundaries JVM heap recommendations range from 4-8GB with strong advice against exceeding 12GB due to GC performance degradation and pause time increases. File descriptor limits must accommodate open connections plus log segment files, typically requiring 100,000+ on high-throughput production systems. Network connection limits are bounded by operating system socket limits and available memory for connection tracking structures.

Default Values Default heap allocation is 1GB (-Xms1G -Xmx1G), log retention is 7 days (log.retention.hours=168), and replication factor is 1 (default.replication.factor=1). Network configuration defaults include 3 network threads (num.network.threads=3), 8 I/O threads (num.io.threads=8), and 100KB socket buffers (socket.send.buffer.bytes=102400).

Best Practices Allocate 50% of system memory to JVM heap with remaining memory available for page cache optimization, monitor GC logs for pause time analysis and heap utilization patterns. Configure separate disk volumes for log storage and OS operations, implement RAID configurations appropriate for durability

requirements versus performance needs. Establish configuration management practices with version control and validation procedures to prevent configuration drift and ensure consistent cluster behavior.

ZooKeeper vs KRaft (post-3.x)

Definition ZooKeeper is a distributed coordination service providing hierarchical namespace, configuration management, and consensus capabilities that Kafka historically used for metadata storage, controller election, and cluster coordination. KRaft (Kafka Raft) is a self-contained consensus protocol implemented within Kafka brokers that eliminates external ZooKeeper dependency by managing metadata through internal replicated logs and Raft leader election algorithms.

Key Highlights ZooKeeper requires separate 3-5 node clusters with independent deployment, monitoring, and operational procedures, while KRaft integrates metadata management directly into designated Kafka controller brokers. KRaft achieved production readiness in Kafka 3.3 and became the recommended deployment mode, with ZooKeeper support officially deprecated in newer versions. Migration from ZooKeeper to KRaft requires careful planning with potential downtime depending on cluster size and migration strategy.

Responsibility / Role Both systems handle broker membership discovery, topic metadata storage, partition assignment coordination, and access control list (ACL) management across distributed cluster environments. They provide strongly consistent metadata replication, leader election capabilities for controller roles, and configuration change propagation with version tracking and conflict resolution. Critical responsibilities include maintaining partition leadership information, broker liveness detection, and coordinating administrative operations across cluster members.

Underlying Data Structures / Mechanism ZooKeeper implements hierarchical znodes with sequential consistency guarantees, watch mechanisms for change notifications, and majority quorum requirements for write operations. KRaft uses Raft consensus algorithm with leader election, log replication, and metadata stored in internal __cluster_metadata topic with configurable replication factors. Both systems maintain strongly consistent state machines but differ significantly in scalability characteristics and operational complexity.

Advantages ZooKeeper provides battle-tested reliability with extensive operational tooling, monitoring solutions, and deep community knowledge accumulated over decades of production usage. KRaft eliminates external system dependencies, reduces operational complexity, improves metadata scalability to support larger partition counts, and provides faster controller failover times. KRaft also enables faster cluster startup times and simplified deployment architectures.

Disadvantages / Trade-offs ZooKeeper adds significant operational overhead requiring separate cluster management, monitoring, backup procedures, and specialized expertise for troubleshooting and performance tuning. KRaft represents newer technology with evolving tooling ecosystem, limited operational experience, and potential migration risks for existing production environments. ZooKeeper's metadata storage limitations become bottlenecks for clusters with very large partition counts (>100,000).

Corner Cases ZooKeeper network partitions can render entire Kafka clusters unavailable even when broker nodes remain healthy and operational, requiring careful network design and monitoring. KRaft metadata corruption or controller quorum loss can require complex recovery procedures potentially involving metadata reconstruction from broker logs. Split-brain scenarios during network partitions require different recovery procedures depending on coordination system choice.

Limits / Boundaries ZooKeeper clusters typically support maximum 1 million znodes with recommended cluster sizes of 3-5 nodes, and session timeout ranges from 2-40 seconds. KRaft supports significantly larger metadata sets without znode limitations and scales controller operations more efficiently, supporting clusters with hundreds of thousands of partitions. Memory requirements differ significantly, with ZooKeeper requiring 2-8GB heap while KRaft controllers need memory proportional to metadata volume.

Default Values ZooKeeper default configuration includes 3-node clusters, 2GB heap allocation, 2-second tick time, and 20-tick session timeout (40 seconds total). KRaft defaults dedicate specific broker nodes as controllers with separate log directories, default replication factor of 3 for metadata topic, and faster failover timeouts.

Best Practices Plan ZooKeeper to KRaft migrations during scheduled maintenance windows with comprehensive testing in staging environments replicating production configurations and workload patterns. Deploy KRaft controller nodes on dedicated hardware with sufficient memory and fast storage for metadata operations, separate from regular broker workloads. Implement robust monitoring for both coordination systems focusing on leader election frequency, metadata operation latency, and quorum health metrics.

Cluster Metadata & Controller Election

Definition Cluster metadata represents the authoritative source of partition assignments, topic configurations, broker membership, replica placement policies, and access control information that defines complete cluster state and operational behavior. Controller election implements distributed consensus algorithms to select a single broker responsible for metadata management, partition leadership coordination, and administrative operation execution across the entire cluster.

Key Highlights The controller maintains comprehensive cluster topology including partition-to-broker mappings, in-sync replica sets, leader assignments, and configuration overrides with versioning for conflict detection and resolution. Controller election uses ZooKeeper leader election or KRaft consensus protocols to ensure single active controller per cluster, with automatic failover during failures. Metadata changes propagate through dedicated communication channels with acknowledgment mechanisms ensuring eventual consistency across all brokers.

Responsibility / Role Controllers coordinate all partition state changes including leader election during broker failures, replica assignment during cluster expansion, and partition reassignment during maintenance operations. They process administrative requests for topic creation/deletion, configuration updates, and ACL modifications while ensuring cluster-wide consistency and proper validation. Metadata distribution responsibilities include broadcasting updates to all brokers and coordinating rolling updates of cluster configuration.

Underlying Data Structures / Mechanism Metadata storage varies between ZooKeeper (hierarchical znodes with watchers) and KRaft (replicated log segments with Raft consensus) but both maintain versioned state with change tracking capabilities. Controller election uses distributed locking mechanisms with lease renewal protocols to prevent split-brain scenarios during network partitions. Metadata propagation implements reliable broadcast protocols with retry logic and acknowledgment tracking for delivery guarantees.

Advantages Centralized metadata management ensures consistent cluster state and simplifies coordination logic compared to distributed consensus approaches for every operation. Automatic controller failover provides high availability for administrative operations without manual intervention or external coordination

systems. Metadata versioning enables conflict detection, rollback capabilities, and ensures consistent view of cluster state across all participants.

Disadvantages / Trade-offs Controller becomes performance bottleneck for metadata-intensive operations including topic creation, partition reassignment, and large-scale configuration updates affecting cluster scalability. Controller failures temporarily suspend all administrative operations until new leader election completes, potentially impacting automated deployment and scaling processes. Single controller architecture creates potential availability issues if election algorithms fail or network partitions isolate the controller from cluster members.

Corner Cases Metadata corruption scenarios require careful recovery procedures potentially involving metadata reconstruction from individual broker state and transaction logs. Controller isolation during network partitions can create scenarios where administrative operations appear successful but fail to propagate to isolated brokers. Concurrent metadata modifications can trigger race conditions requiring careful ordering and conflict resolution in controller logic.

Limits / Boundaries Controller capacity typically handles 10,000-50,000 partition state changes per second depending on hardware and metadata complexity, with performance degrading as cluster size increases. Metadata size limitations depend on storage backend (ZooKeeper znode limits vs KRaft log capacity) but generally support clusters with millions of partitions. Controller election timeouts range from seconds to minutes balancing availability against consistency requirements.

Default Values ZooKeeper-based controller election timeout is 18 seconds with 6-second session timeout, while KRaft controller elections complete in 2-5 seconds typically. Metadata propagation timeout defaults to 30 seconds for broker acknowledgments, and controller failure detection occurs within 6-10 seconds under normal conditions.

Best Practices Monitor controller failover frequency as key cluster health indicator, with frequent elections suggesting network instability, configuration issues, or resource constraints. Implement comprehensive metadata backup strategies including periodic snapshots and transaction log archival for disaster recovery scenarios. Size controller resources (CPU, memory, network) appropriately for expected administrative workload and metadata volume, considering peak operational periods and growth projections.