# Building a Robust High‑Throughput Kafka Cluster (Bare-Metal Handbook)

In this handbook, we explore how to design, configure, and maintain a **robust Apache Kafka cluster** that can handle high-throughput streaming in a distributed system. We assume a **bare-metal deployment** (full control of hardware) and a target load of roughly **1,000 messages per second** with ~**100 KB** per message (about 100 MB/s). Achieving this reliably requires careful planning in cluster architecture, hardware provisioning, Kafka configuration tuning, and comprehensive monitoring. We will cover best practices in each area, providing practical recommendations and trade-offs backed by references.

## 1. Cluster Architecture and Sizing

**Cluster Size and Fault Tolerance:** For production, deploy **at least three Kafka broker nodes** to form a cluster[[1]](https://openmetal.io/resources/blog/building-high-throughput-data-ingestion-pipelines-with-kafka-on-openmetal/#:~:text=Begin%20by%20deploying%20your%20on,cores%2C%20RAM%2C%20and%20%2021). This allows using a **replication factor of 3** for topics, meaning each piece of data is stored on three brokers. A replication factor of 3 is considered an industry standard balance between fault tolerance and overhead[[2]](https://github.com/AutoMQ/automq/wiki/Kafka-Replication:-Concept-&-Best-Practices#:~:text=The%20replication%20factor%20is%20a,maintained%20across%20the%20Kafka%20cluster). With three replicas, the cluster can tolerate up to two broker failures without losing data availability[[3]](https://github.com/AutoMQ/automq/wiki/Kafka-Replication:-Concept-&-Best-Practices#:~:text=,8). If using Kafka versions < 3.x, also run a ZooKeeper ensemble (3 or 5 nodes) for metadata; in newer Kafka releases, you have the option of the built-in KRaft mode to eliminate external ZooKeeper (ensure you run an odd number of controller nodes for quorum).

**Broker Roles:** Each topic partition in Kafka has one **leader broker** handling reads/writes and the rest as **followers** replicating the data[[4]](https://github.com/AutoMQ/automq/wiki/Kafka-Replication:-Concept-&-Best-Practices#:~:text=Leaders%20and%20Followers%20Architecture). Clients produce to and consume from leaders; followers stay in sync (in the **ISR**, in-sync replicas) and take over if a leader fails[[5]](https://github.com/AutoMQ/automq/wiki/Kafka-Replication:-Concept-&-Best-Practices#:~:text=The%20In,with%20the%20leader). Configure **min.insync.replicas=2** (when replication factor is 3) so that at least two brokers (leader + one) must acknowledge a write when using acks=all for strong durability[[3]](https://github.com/AutoMQ/automq/wiki/Kafka-Replication:-Concept-&-Best-Practices#:~:text=,8). This guards against data loss by requiring a quorum of replicas for each commit.

**Topic Partitions and Parallelism:** Partitions are Kafka’s unit of parallelism. More partitions allow greater throughput by distributing load, but too many partitions add overhead. Aim to **size partitions to your throughput and consumer parallelism needs**. A common guideline is **1–2 partitions per CPU core per broker as a starting point**[[6]](https://openmetal.io/resources/blog/building-high-throughput-data-ingestion-pipelines-with-kafka-on-openmetal/#:~:text=Now%2C%20from%20one%20of%20the,partitions%20per%20core%20per%20broker). For example, a cluster of 3 brokers with 4 cores each might start with ~12 partitions for a heavy topic (3 brokers × 4 cores = 12). This allows consumers to scale out (up to 12 consumer threads can read in parallel) and producers to spread load. Monitor how partition count affects utilization: too few partitions may bottleneck on a single broker or consumer, while too many increase metadata overhead and context-switching[[7]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=4)[[8]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=,Kafka%20relies%20on%20the). There is no one-size-fits-all number – adjust based on throughput tests and keep partitions evenly distributed across brokers.

**Scaling Strategies:** Kafka clusters can scale **vertically** (bigger brokers) or **horizontally** (more brokers). Adding brokers increases aggregate disk and network throughput linearly (assuming partitions are redistributed)[[9]](https://openmetal.io/resources/blog/building-high-throughput-data-ingestion-pipelines-with-kafka-on-openmetal/#:~:text=Scaling%20Your%20Pipeline%20for%20Future,Growth). For sustained 100 MB/s ingest, you might spread the load across brokers – e.g. 3 brokers each handling ~33 MB/s – to avoid overloading a single node. It’s often better to have **multiple medium-size brokers** than one giant node, for finer failure domains and easier rolling maintenance[[10]](https://aws.amazon.com/blogs/big-data/best-practices-for-right-sizing-your-apache-kafka-clusters-to-optimize-performance-and-cost/#:~:text=most%20important%20aspects%20are%20as,follows)[[11]](https://aws.amazon.com/blogs/big-data/best-practices-for-right-sizing-your-apache-kafka-clusters-to-optimize-performance-and-cost/#:~:text=,sense%20to%20either%20increase%20the). However, too many small brokers can prolong maintenance (rolling restarts) and raise coordination overhead[[10]](https://aws.amazon.com/blogs/big-data/best-practices-for-right-sizing-your-apache-kafka-clusters-to-optimize-performance-and-cost/#:~:text=most%20important%20aspects%20are%20as,follows). Strike a balance based on hardware and operational capacity. Start with the minimum 3 and scale out if monitoring shows high sustained utilization (and keep the cluster at **~70–80% of theoretical max throughput** to allow headroom for bursts and failover)[[12]](https://aws.amazon.com/blogs/big-data/best-practices-for-right-sizing-your-apache-kafka-clusters-to-optimize-performance-and-cost/#:~:text=For%20production%20clusters%2C%20it%E2%80%99s%20a,cluster%20should%20target%20800%20MB%2Fsec)[[13]](https://aws.amazon.com/blogs/big-data/best-practices-for-right-sizing-your-apache-kafka-clusters-to-optimize-performance-and-cost/#:~:text=To%20find%20the%20right%20configuration,capacity%2C%20for%20instance%2C%20during%20maintenance).

**Network Topology:** If your deployment spans racks or data centers, use rack-aware broker configuration so that replicas are placed on different racks (to survive rack-level outages). For cross-datacenter redundancy or disaster recovery, consider **MirrorMaker2** or Confluent Replicator to mirror topics to a backup cluster (this introduces additional complexity and is typically needed only for multi-DC robust systems).

## 2. Hardware and OS Considerations (Bare Metal)

Leveraging bare metal means you can optimize the hardware and OS for Kafka’s I/O-intensive workload. High throughput streaming (100 MB/s+) will stress **disk, network, and OS resources**, so ensure these components are up to the task:

**Storage – Fast Disks and Filesystem:** Use **fast disks (SSD or NVMe)** for Kafka data logs, as disk throughput is often the limiting factor in Kafka performance[[14]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=,with%20Kafka%27s%20replication%20handling%20redundancy)[[15]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=,Direct%20disk%20access%20is%20preferred). Modern NVMe SSDs can sustain tens of thousands of IOPS and high sequential throughput, which is ideal for Kafka’s append-heavy access pattern (and for compaction if used)[[16][16]](https://github.com/AutoMQ/automq/wiki/Kafka-Performance-Tuning:-Tips-&-Best-Practices#:~:text=Reddit%20discussions%20highlight%20the%20performance,13). In contrast, HDDs may become a bottleneck at 100 MB/s unless using multiple spindles in parallel (and add latency on random access). If using HDDs, consider a **JBOD setup with multiple drives** (Kafka can spread partitions across disks listed in log.dirs) to aggregate throughput[[7]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=4). RAID is *not strictly necessary* for durability (Kafka already replicates data across brokers), but RAID-10 can help if you need a single volume’s throughput or easier single-mount management[[17]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=are%20not%20extremely%20high,Direct%20disk%20access%20is%20preferred). Many Kafka deployments prefer direct-attached disks without RAID/LVM to reduce overhead[[18]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=,Direct%20disk%20access%20is%20preferred)[[19]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=redundancy,Direct%20disk%20access%20is%20preferred). Use a robust filesystem like **XFS or ext4** (XFS is known to perform well with Kafka)[[20]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=1).

**Memory (RAM) and Page Cache:** Kafka heavily relies on the OS page cache to buffer active data in memory for fast access[[21]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=2). After setting the Java heap for the broker (discussed below), **leave the rest of the memory to the OS**. For example, on a machine with 64 GB RAM, you might give the Kafka broker 8 GB heap and allow ~56 GB for the page cache. This way, recent messages and index files stay in memory, reducing disk reads. Monitor cache hit ratios if possible – if the workload is largely sequential writes and reads from tail, the page cache will naturally be effective. Also, **disable or minimize swap** (vm.swappiness=1 or 0) to avoid the OS swapping out Kafka’s memory under load[[22]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=5)[[23]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=Reduce%20,heap%2C%20which%20can%20cripple%20performance). Swapping can pause the JVM and drastically hurt throughput.

**CPU and Cores:** Kafka is generally more I/O-bound than CPU-bound, but adequate CPU ensures the broker can handle compression, message formatting, and connection overhead. For a throughput-focused cluster, ensure you have enough cores to handle background tasks (garbage collection, disk I/O, replication) without pegging at 100% user CPU. Faster cores (higher clock speed) help reduce per-message overhead and lock contention. Also, set the **CPU frequency governor to “performance”** on Linux to avoid dynamic frequency scaling; consistent full-speed CPU prevents latency hiccups[[24]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=4).

**Network Interface:** A sustained 100 MB/s publish rate will nearly saturate a single 1 Gigabit Ethernet link (which tops out ~125 MB/s at layer 2, less at application level). And with replication (factor 3), the cluster’s internal traffic could be ~300 MB/s (leader to followers) plus consumer traffic. Therefore, equip brokers with **10 GbE (or higher)** network interfaces for high-throughput use cases[[25]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=3). A 10 GbE NIC (which supports up to ~1,250 MB/s) provides plenty of headroom for 100 MB/s payload plus replication and future growth. Ensure network infrastructure (switches, cables) can handle the load; if possible, keep brokers in the same low-latency LAN segment. For added throughput or redundancy, some deploy bond/LACP with multiple NICs, but one 10 Gb link is usually simpler. **Enable jumbo frames (MTU 9000)** if your network supports it, as this can improve throughput by reducing CPU overhead per packet (just ensure all clients and switches use the same MTU)[[26][27]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=,clients%2C%20brokers%2C%20and%20network%20devices).

**OS Kernel Tuning:** On Linux, increase socket buffer limits to fully utilize the NIC’s capacity. For example, raise net.core.rmem\_max and net.core.wmem\_max (maximum TCP receive/send buffer sizes) in **/etc/sysctl.conf**[[25]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=3)[[26]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=,clients%2C%20brokers%2C%20and%20network%20devices). This allows Kafka (and the OS) to use larger TCP windows for high-latency or high-bandwidth scenarios. Similarly, ensure net.ipv4.tcp\_rmem and tcp\_wmem have high enough default and max values (these are triplets defining min/default/max TCP buffer sizes). Tuning these is especially important if using long-haul networks or if you observe frequent network congestion. Also consider increasing the **open file descriptors limit** (ulimit -n) for the Kafka process, because each broker can open many files (log segments) and sockets simultaneously[[28]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=6)[[29]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=Increase%20the%20open%20file%20descriptor,log%20segments%20and%20network%20connections). For example, set a nofile limit of 100k or more. The Linux default (often 1024 or 4096) is far too low for Kafka’s scale. Update /etc/security/limits.conf or systemd unit files to persist this for the kafka user.

**JVM Tuning:** Configure Kafka’s JVM heap size explicitly. A good rule of thumb is **4 GB to 8 GB heap per broker** for most cases[[30]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=%28%60KAFKA_HEAP_OPTS%60%2C%20e.g.%2C%20%60,heavily%20for%20reads%20and%20writes)[[31]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=for%20reads%20and%20writes). Avoid overly large heaps (which can cause long garbage collection pauses and diminish page cache space) unless you have a specific need. Kafka’s default garbage collector (JDK 11+) is G1GC, which is usually suitable. You can set -Xms8g -Xmx8g (for 8 GB) in the Kafka start script or environment. Monitor GC logs – if you see frequent GC or high old-gen usage, you might adjust heap or tune G1GC settings. But generally, Kafka performance benefits more from the OS cache than from huge heaps[[30]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=%28%60KAFKA_HEAP_OPTS%60%2C%20e.g.%2C%20%60,heavily%20for%20reads%20and%20writes).

## 3. Broker Configuration for Throughput and Resilience

After hardware, how you configure the Kafka **broker (server.properties)** has a major impact on throughput. Here are key broker-side tunables and best practices for a robust, high-throughput cluster:

* **Network Threads (num.network.threads)** – This controls the number of threads for handling network I/O (incoming client requests and outgoing responses). The default is 3, which may be low for very busy clusters. A good starting point is to **set network threads equal to the number of CPU cores** on the broker machine[[32]](https://openmetal.io/resources/blog/building-high-throughput-data-ingestion-pipelines-with-kafka-on-openmetal/#:~:text=,on%20your%20bare%20metal%20server)[[33]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=2,threads). For example, on an 8-core machine use num.network.threads=8. This ensures the broker can handle many concurrent connections and network operations in parallel. Monitor the broker metric **NetworkProcessorAvgIdlePercent** – if it drops very low (meaning threads are busy), consider increasing threads[[32]](https://openmetal.io/resources/blog/building-high-throughput-data-ingestion-pipelines-with-kafka-on-openmetal/#:~:text=,on%20your%20bare%20metal%20server).
* **I/O Threads (num.io.threads)** – Threads for disk I/O and request processing. These should also be scaled with hardware. A common setting is **2× number of cores** (especially if you have multiple disks)[[34][34]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=,can%20be%20a%20starting%20point). For example, num.io.threads=16 on an 8-core. This gives more threads to read/write data to disk concurrently. If using several log directories (disks), more I/O threads help handle parallel reads/writes. Again monitor **RequestHandlerAvgIdlePercent** – you want some idle headroom (not 0%) under load[[32]](https://openmetal.io/resources/blog/building-high-throughput-data-ingestion-pipelines-with-kafka-on-openmetal/#:~:text=,on%20your%20bare%20metal%20server).
* **Request Queue (queued.max.requests)** – This is the max number of requests that can queue awaiting processing (default 500). In high-throughput scenarios, you can raise this to **a few hundred or 1000** to avoid throttling producers/consumers if spikes occur[[35]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=6)[[36]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=,processor%20before%20requests%20are%20blocked). For instance, queued.max.requests=1000. Keep an eye on request queue size and handler latency in metrics; a too-large queue could increase latency if it grows significantly, but too small may reject requests under bursty load.
* **Socket Buffers (socket.send.buffer.bytes and socket.receive.buffer.bytes)** – These set the broker’s TCP socket buffer hint for sending/receiving. Defaults are ~100 KB. In a well-tuned OS (with large rmem\_max/wmem\_max as above), you may leave these at -1 (which means use OS default) or set them explicitly to a higher value if needed[[37]](https://openmetal.io/resources/blog/building-high-throughput-data-ingestion-pipelines-with-kafka-on-openmetal/#:~:text=,flight%20data)[[25]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=3). For example, you might use 256000 (256 KB) or more. Larger socket buffers can improve throughput on high-bandwidth or long-latency networks by allowing more data in-flight. However, on a low-latency LAN, the default buffers often suffice. It’s safe to start with OS defaults (which could be 1 MB on modern systems) and adjust if you see network saturation issues[[38]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=6).
* **Replication Threads (num.replica.fetchers)** – This controls how many threads each broker uses to replicate data from partition leaders (for which this broker is a follower). Default is 1. Under high load, especially if brokers host many partitions, a single thread can become a bottleneck for replication. Bumping this to **2–4** threads can speed up replica catch-up and reduce follower lag[[39]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=5)[[40]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=,broker%20hosts%20many%20follower%20partitions). For example, num.replica.fetchers=3 is a balanced choice for a busy cluster. Monitor **follower lag** metrics; if replicas are falling far behind leaders, more fetcher threads or higher replica.fetch.max.bytes may help.
* **Replication Fetch Size (replica.fetch.max.bytes)** – The max bytes per fetch request from followers. Default ~1 MB is low for modern throughput. Consider increasing this (e.g. 4 MB or 8 MB) so followers pull more data per request[[41]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=consistently%20high%2C%20increasing%20this%20can,%28default%3A%2064KB%29%3A%20Network%20receive). Likewise, replica.fetch.response.max.bytes on brokers (max bytes in a fetch response to followers) should be tuned in concert. Setting these to a few multiples of your typical batch size ensures replication is efficient and not making too many small fetches. Just ensure it’s not so high that it strains network/disk (staying under socket buffer limits and memory limits for responses).
* **Log Segments and Retention:** Kafka writes data in segment files per partition. For throughput, a **larger segment size** means fewer files and less frequent disk I/O for rolling segments. A common setting is **log.segment.bytes=512 MB to 1 GB** (instead of default 1 GB, you can choose in that range)[[42]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=starting%20point%20is%20%60num_brokers%20,let%20the%20OS%20manage%20flushing). Larger segments reduce overhead but will increase recovery time (e.g. broker restart scanning a 1 GB segment vs 100 MB segment). Since our focus is throughput and we have large messages (100 KB each), 1 GB segments (~10k messages per segment in worst case) is reasonable. For **retention**, configure according to use-case (time or size based). Retaining a large volume of data won’t directly hurt throughput *until* consumers fall far behind and start reading older (uncached) data, which can increase disk reads. If historical data access is rare, long retention is fine as long as you have disk capacity. Otherwise, consider offloading old data to archival storage or use tiered storage solutions.
* **Durability vs Throughput (Flush Policy):** Kafka by default does **not fsync each message**; it relies on periodic flush or OS cache flush and replication for durability. Avoid setting log.flush.interval.ms or log.flush.interval.messages to low values – that would force frequent fsync and kill throughput[[43]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=segments%20mean%20fewer%20files%2C%20potentially,let%20the%20OS%20manage%20flushing). It’s usually best to keep flush interval high (or infinite, using OS flush). Rely on replication for safety instead of disk-sync on each message. Ensure your replication factor and acks settings (discussed later) are set to avoid data loss in case of crashes rather than trying to flush every message to disk (which isn’t feasible at 100 MB/s rates).
* **Unclean Leader Election:** For robustness, ensure unclean.leader.election.enable=false (which is default in modern Kafka). This prevents out-of-sync brokers from becoming leader and potentially causing data loss. This setting doesn’t directly affect throughput, but it’s important for a “robust” cluster in failure scenarios.
* **Physical Log Placement (log.dirs):** Spread data across multiple disks if available. List each mount in log.dirs (e.g., /mnt/disk1/kafka-logs,/mnt/disk2/kafka-logs) – Kafka will **place partitions evenly across directories** to distribute I/O[[44]](https://openmetal.io/resources/blog/building-high-throughput-data-ingestion-pipelines-with-kafka-on-openmetal/#:~:text=,I%2FO%20from%20the%20operating%20system). This can significantly increase total throughput if each disk handles a portion of partitions (ensuring, say, two 500 MB/s SSDs give an aggregate 1 GB/s capacity). Make sure each listed directory is on a separate physical volume for true performance gain.
* **Broker JVM Settings:** Configure the broker’s Java options for production. We already covered heap sizing (4–8 GB). Use the G1 garbage collector (enabled by default in newer Kafka). It’s also wise to add -XX:MaxGCPauseMillis=20 (as a tuning goal for GC) and ensure GC logging is on for troubleshooting. In containerized environments, use -XX:MaxRAMPercentage instead of a fixed Xmx, if deploying via Docker/Kubernetes, to size heap relative to container memory.

## 4. Producer Configuration Tuning (Clients)

Producers have a big impact on throughput. The goal is to send messages in larger batches, compress data, and reduce unnecessary waits – all while balancing latency and durability needs. Below are key producer settings to tune in your producer applications (or globally via client configs) for high-throughput streaming:

* **Batch Size (batch.size)** – This controls the maximum size (in bytes) of a batch of messages before the producer sends it. By default it might be 16 KB; increasing to **128 KB (131,072 bytes)** or more helps throughput[[45]](https://docs.redhat.com/en/documentation/red_hat_streams_for_apache_kafka/2.7/html/kafka_configuration_tuning/con-high-volume-config-properties-str#:~:text=For%20producers%2C%20you%20might%20increase,handle%20large%20numbers%20of%20messages)[[46]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=2,Partitions). A larger batch allows more messages to be sent in one request, **amortizing overhead and improving compression efficiency** (if compression is enabled)[[45]](https://docs.redhat.com/en/documentation/red_hat_streams_for_apache_kafka/2.7/html/kafka_configuration_tuning/con-high-volume-config-properties-str#:~:text=For%20producers%2C%20you%20might%20increase,handle%20large%20numbers%20of%20messages). However, the producer will not exceed this size per partition, so it’s a ceiling rather than a target. Ensure the producer’s memory (buffer.memory) can handle holding these batches. *Trade-off:* Bigger batches add **slight latency** (messages wait to fill the batch) and use more client memory, but greatly boost throughput by reducing network calls.
* **Lingering (linger.ms)** – By default, the producer sends a batch as soon as it’s ready (linger=0). Setting a small linger, e.g. **5–20 ms**, instructs the producer to wait up to that time for more messages to join the batch[[47]](https://docs.redhat.com/en/documentation/red_hat_streams_for_apache_kafka/2.7/html/kafka_configuration_tuning/con-high-volume-config-properties-str#:~:text=Use%20,under%20the%20maximum%20batch%20size). This is very useful in high-throughput or bursty workloads: a **linger of 10 ms** can coalesce a lot of messages that arrive around the same time into one batch. The cost is an added delay (up to that many milliseconds) for messages when the pipeline isn’t full. In practice, a 5–50 ms linger can dramatically increase throughput with minimal impact on end-to-end latency for a steady stream[[47]](https://docs.redhat.com/en/documentation/red_hat_streams_for_apache_kafka/2.7/html/kafka_configuration_tuning/con-high-volume-config-properties-str#:~:text=Use%20,under%20the%20maximum%20batch%20size). For our 1000 msg/s scenario, a linger of say 10 ms is reasonable.
* **Compression (compression.type)** – Enable compression to reduce bandwidth usage and I/O. Kafka supports **gzip, lz4, snappy, zstd**. For high throughput, **LZ4** is often recommended as it’s very fast with decent compression ratio, adding minimal CPU overhead. **Zstd** compresses more (smaller data size, saving network and disk) but uses more CPU, which might be acceptable if CPU is not a bottleneck. Avoid uncompressed if data is large; compression can significantly cut down the 100 KB message size (depending on content) and thus reduce the 100 MB/s throughput requirement. Set producer compression.type=lz4 (or experiment with zstd if CPU allows). The broker and consumers will automatically decompress. *Note:* compressing at the producer typically **outperforms per-topic compression at broker**, because compression is done once at source and persisted as such[[1]](https://openmetal.io/resources/blog/building-high-throughput-data-ingestion-pipelines-with-kafka-on-openmetal/#:~:text=Begin%20by%20deploying%20your%20on,cores%2C%20RAM%2C%20and%20%2021) (you can also set a topic to compress all messages as they’re stored, but that’s redundant if producers already send compressed).
* **Acknowledgment Level (acks)** – This setting balances throughput vs. durability. acks=1 means the leader broker will respond to producer after writing the message to its log (but without waiting for any followers) – this is **fast but can lose data** if the leader crashes right after write before followers have it[[48]](https://stackoverflow.com/questions/70980719/kafka-is-losing-message-when-ack-is-one#:~:text=Kafka%20is%20losing%20message%20when,Kakfa%20Producer%20Message%20Delivery)[[49]](https://forum.confluent.io/t/why-losing-messages-on-read/2759#:~:text=Kafka%20only%20guarantees%20that%20a,ack%20back%20before%20replication%20finished). acks=all means the leader will respond only after all in-sync replicas (min.insync.replicas) have received the message, thus guaranteeing it is fully replicated[[50]](https://www.confluent.io/blog/5-things-every-kafka-developer-should-know/#:~:text=Confluent%20www,to%20using%20the%20acks%3Dall)[[51]](https://forum.confluent.io/t/why-losing-messages-on-read/2759#:~:text=Kafka%20only%20guarantees%20that%20a,ack%20back%20before%20replication%20finished). For a *robust* cluster, it’s recommended to use acks=all together with replication factor 3 and min.insync.replicas 2 (ensuring at least 2 brokers have the data) – this combo gives strong durability. However, acks=all incurs a bit more end-to-end latency (the produce request is slower because it waits for replicas). If your absolute highest throughput with some risk is desired, acks=1 will be slightly faster since it doesn’t wait for followers[[48]](https://stackoverflow.com/questions/70980719/kafka-is-losing-message-when-ack-is-one#:~:text=Kafka%20is%20losing%20message%20when,Kakfa%20Producer%20Message%20Delivery)[[51]](https://forum.confluent.io/t/why-losing-messages-on-read/2759#:~:text=Kafka%20only%20guarantees%20that%20a,ack%20back%20before%20replication%20finished). You must decide: **Throughput-biased** profile might use acks=1 (and risk a tiny window of data loss on broker failure), whereas **durability-biased** profile uses acks=all[[52]](https://www.confluent.io/blog/5-things-every-kafka-developer-should-know/#:~:text=Confluent%20www,to%20using%20the%20acks%3Dall). In either case, set the broker min.insync.replicas accordingly (with a factor 3 cluster, use 2 for acks=all, or can use 1 for acks=1 to allow writes even if one replica is offline).
* **Idempotence and Max In-Flight Requests:** Modern Kafka producers (since 3.0) enable idempotent delivery by default (enable.idempotence=true)[[53]](https://github.com/AutoMQ/automq/wiki/Kafka-Performance-Tuning:-Tips-&-Best-Practices#:~:text=max.inflight.requests.per.connection%20,throughput%20but%20increases%20memory%20usage). This ensures no duplicates are introduced on retries, and it auto-enables acks=all and retries=Integer.MAX by default. One caveat historically was to set max.in.flight.requests.per.connection=1 to keep message order with retries, but since Kafka’s idempotent producer (with sequence numbers) allows up to 5 in-flight requests by default, you **do not need to drop this to 1 unless using older Kafka or strict ordering on failover**[[53]](https://github.com/AutoMQ/automq/wiki/Kafka-Performance-Tuning:-Tips-&-Best-Practices#:~:text=max.inflight.requests.per.connection%20,throughput%20but%20increases%20memory%20usage)[[54]](https://github.com/AutoMQ/automq/wiki/Kafka-Performance-Tuning:-Tips-&-Best-Practices#:~:text=Image). The default 5 in-flight is fine and maintains order in normal operation while improving throughput (more pipelining). In summary, **use the default idempotent producer settings** – it gives you reliability (exactly-once insert semantics when combined with proper processing) with minimal config effort. Only if you see ordering issues or specific constraints should you lower in-flights to 1 (at a significant throughput cost).
* **Retries and Timeouts:** The producer retries count and delivery.timeout.ms determine how it handles transient errors. By default, with idempotence, Kafka client will retry indefinitely (or rather Integer.MAX\_VALUE times) within the delivery timeout (default 2 minutes). You typically **don’t need to tweak retries** – leave it high – but you might adjust delivery.timeout.ms if you want to declare a message failed sooner or later. The defaults are usually fine; focus on monitoring if any errors happen. For high throughput, ensure the request.timeout.ms (broker response wait) isn’t too low. Defaults (30s) are usually sufficient. So, unless you observe timeouts in practice, you can rely on defaults for these. The key is that with idempotence and acks=all, the producer will internally handle retries to ensure delivery to replicas.
* **Producer Memory (buffer.memory)** – If you increased batch sizes significantly, also ensure the producer’s total buffer (memory pool for unsent records) is large enough. The default ~32 MB might be low if you have many partitions or very high throughput. Setting buffer.memory=33554432 (32 MB) to 67108864 (64 MB) or more could be beneficial in a 100 MB/s scenario to buffer bursts. Monitor if producers are blocking due to running out of buffer (exposed via JMX metrics on producers).

**Producer Profile Examples:** In practice, a **throughput-optimized producer** might use batch.size=131072 (128 KB), linger.ms=10, compression.type=lz4, acks=1 (with the understanding of potential risk). A **balanced profile for durability** would use acks=all (with proper ISR settings) and rely on idempotence, keeping other settings similar (batching, compression still apply). Always test with your actual workload – measure produce request rate, batch sizes, and end-to-end latency to fine-tune these values.

## 5. Consumer Configuration Tuning (Clients)

On the consumer side, we want to maximize throughput of fetching data from brokers while avoiding overwhelming the consumer application. Key settings for high-throughput consumers include fetch sizes and parallelism controls:

* **Fetch Min Bytes (fetch.min.bytes)** – By default a Kafka consumer will return data as soon as one message is available (1 byte minimum). For high throughput, increase this so the broker waits to send a larger batch of data. Setting **fetch.min.bytes=1048576 (1 MB)** is a good start[[55]](https://docs.redhat.com/en/documentation/red_hat_streams_for_apache_kafka/2.7/html/kafka_configuration_tuning/con-high-volume-config-properties-str#:~:text=For%20consumers%2C%20you%20might%20increase,property)[[56]](https://docs.redhat.com/en/documentation/red_hat_streams_for_apache_kafka/2.7/html/kafka_configuration_tuning/con-high-volume-config-properties-str#:~:text=For%20producers%2C%20you%20might%20increase,when%20you%20have%20just%20a). This means the broker will try to send at least 1 MB of messages per fetch response (or respond sooner if fetch.max.wait.ms is reached). The effect is **fewer, bigger network packets**, which improves throughput efficiency at the cost of a slight latency if data isn’t coming in fast enough. Pair this with a **fetch.max.wait.ms=100** (e.g. 100 ms) to bound how long the broker will wait[[55]](https://docs.redhat.com/en/documentation/red_hat_streams_for_apache_kafka/2.7/html/kafka_configuration_tuning/con-high-volume-config-properties-str#:~:text=For%20consumers%2C%20you%20might%20increase,property). In steady 100 MB/s flows, the broker will almost always have 1 MB ready quickly, so latency impact is minimal.
* **Max Partition Fetch Bytes (max.partition.fetch.bytes)** – This is the maximum data returned *per partition* in one fetch. If you expect large batches or records, ensure this is comfortably above your largest message or batch size. For 100 KB messages that might be compressed into batches, you might set **8 MB or 16 MB** here[[55]](https://docs.redhat.com/en/documentation/red_hat_streams_for_apache_kafka/2.7/html/kafka_configuration_tuning/con-high-volume-config-properties-str#:~:text=For%20consumers%2C%20you%20might%20increase,property). For example, max.partition.fetch.bytes=16777216 (16 MB) would allow a single partition’s fetch to bring up to 16 MB at once. If you leave this too low (e.g. 1 MB default), a partition with a lot of data might take many small fetches. Increasing it improves throughput but be mindful of consumer memory (it needs to buffer this data). The **fetch.max.bytes** (total bytes for the entire fetch request across all partitions) can also be raised (e.g. to 64 MB or more) to allow the consumer to grab a lot of data in one go[[55]](https://docs.redhat.com/en/documentation/red_hat_streams_for_apache_kafka/2.7/html/kafka_configuration_tuning/con-high-volume-config-properties-str#:~:text=For%20consumers%2C%20you%20might%20increase,property). This should be below the JVM heap available to the consumer; a common setting is 50–100 MB.
* **Max Poll Records (max.poll.records)** – Controls how many records the consumer returns in one poll() call. Default 500 might be conservative for fast processing. If your consumer can handle, say, 1000 or 5000 records in memory and process them in a batch, you can increase **max.poll.records to a higher number (1,000 – 5,000)**[[57]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=Parameter%20Default%20Tuning%20Guidance%20,heartbeat.interval.ms). This reduces the overhead of polling and can increase throughput by processing more data per loop. The trade-off is your processing time per poll might increase – ensure you don’t exceed the consumer’s max.poll.interval.ms (the timeout for how long a consumer can go without polling before being considered dead). If you process 5000 records and it takes some time, you may need to also increase max.poll.interval.ms appropriately. Essentially, tune max.poll.records such that a batch of that size can be processed well within the poll interval.
* **Consumer Parallelism (Consumer Groups):** Throughput scaling on the consumer side often means adding more consumer instances in the group, up to one per partition. If you have 12 partitions and only 2 consumer threads, one thread handles 6 partitions which may be a lot. By running e.g. 6 consumers (threads or processes) in the group, each might handle 2 partitions, increasing total consumption rate (assuming your processing can also parallelize). The maximum parallelism is one consumer per partition[[58]](https://github.com/AutoMQ/automq/wiki/Kafka-Performance-Tuning:-Tips-&-Best-Practices#:~:text=Parallel%20Consumption)[[59]](https://github.com/AutoMQ/automq/wiki/Kafka-Performance-Tuning:-Tips-&-Best-Practices#:~:text=For%20topics%20with%20multiple%20partitions%2C,2). In a bare-metal scenario, you might run multiple consumer threads in one process, or multiple processes across machines, to scale out. Just remember that adding consumers beyond the number of partitions doesn’t help (they’ll sit idle)[[60]](https://stackoverflow.com/questions/41900694/ideal-value-for-kafka-connect-distributed-tasks-max-configuration-setting#:~:text=In%20a%20Kafka%20Connect%20sink%2C,of%20the%20topics%20it%27s%20reading)[[61]](https://stackoverflow.com/questions/41900694/ideal-value-for-kafka-connect-distributed-tasks-max-configuration-setting#:~:text=partitions%20to%20read%20from,of%20the%20topics%20it%27s%20reading). Plan your partition count to accommodate the max consumers you expect to run.
* **Efficient Processing and Flow Control:** No matter how you tune Kafka configs, the consumer application must be efficient in processing messages. High throughput use-cases often require **batch processing** (process a batch of records at once rather than one by one) and possibly asynchronous processing so that the consumer can continue polling while work is being done[[62]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=3). For example, a pattern is to poll() a batch of records and hand them off to a thread pool for processing, then immediately poll again, so the polling thread isn’t idle. Also make sure to commit offsets appropriately. Using **manual commit** (perhaps batched commits) can give more control. For instance, commit offsets after processing 1000 records, instead of auto-commit every 5 seconds, to balance throughput and at-least-once delivery guarantees[[63]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=4)[[64]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=,but%20requires%20careful%20error%20handling). If using enable.auto.commit=true, ensure the processing is quick enough that auto commits don’t lag or time out.
* **Socket Buffer (consumer)** – Similar to producers, consumers also have receive.buffer.bytes. By default it may be 64 KB or 100 KB. In a high bandwidth environment (10 GbE), you might allow a larger OS receive buffer. Often setting receive.buffer.bytes=-1 (use OS default) or a specific size like 1 MB can help if the network has high latency or if you see socket buffer saturation[[38]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=6). However, if producers and brokers are on the same LAN, this is usually not a primary tuning lever unless you’ve tuned the OS.

**Consumer Example Profile:** A high-throughput consumer might set fetch.min.bytes=1 MB, fetch.max.wait.ms=100 ms, max.partition.fetch.bytes=16 MB, max.poll.records=1000 or more, and run with enough threads to cover partitions. This ensures it pulls data in large chunks and processes efficiently. Always monitor **consumer lag** (how far behind head each consumer is) via Kafka’s metrics or external tools. If you see lag increasing under load, you either need more consumer parallelism or some tuning of the above parameters to fetch/process faster.

## 6. Topic Configuration and Data Management

Beyond broker defaults, certain topic-level settings ensure the cluster remains robust under heavy streaming load:

* **Replication Factor and In-Sync Replicas:** As noted, use **replication.factor=3 for critical topics** in production[[2]](https://github.com/AutoMQ/automq/wiki/Kafka-Replication:-Concept-&-Best-Practices#:~:text=The%20replication%20factor%20is%20a,maintained%20across%20the%20Kafka%20cluster). Set **min.insync.replicas=2** for those topics when using acks=all, so that a produce will fail if fewer than 2 brokers (of 3) are available to receive data[[3]](https://github.com/AutoMQ/automq/wiki/Kafka-Replication:-Concept-&-Best-Practices#:~:text=,8). This prevents a scenario where only one broker (leader) is up and acked the message (which would then be lost if that broker died). The trade-off is that if one broker is down, your producers with acks=all will get exceptions on send (cannot satisfy min ISR). You must decide if availability or consistency is the priority – many choose durability (requiring ISR) and handle the exceptions (maybe buffer messages until the third broker is back or scale the cluster).
* **Message Size Limits:** High-throughput streaming often involves large messages (100 KB each in our scenario). To avoid issues, configure size limits consistently. On brokers, message.max.bytes (the largest message the broker will allow) should be set a bit higher than your max message or batch size – e.g. if you might have 100 KB messages and batches up to 1 MB, you could set message.max.bytes=2 MB to have a cushion[[65]](https://openmetal.io/resources/blog/building-high-throughput-data-ingestion-pipelines-with-kafka-on-openmetal/#:~:text=bin%2Fkafka,data)[[66]](https://openmetal.io/resources/blog/building-high-throughput-data-ingestion-pipelines-with-kafka-on-openmetal/#:~:text=,data). On producers, max.request.size should be <= that broker limit (default ~1 MB, so likely increase producer max.request.size to 2 MB as well). Consumers’ max.partition.fetch.bytes we already adjusted to handle large batches. And for replication, ensure replica.fetch.max.bytes is at least as large as message.max.bytes so followers can fetch the largest messages. Aligning these avoids **“Message Too Large” errors** and ensures large events flow through smoothly[[65]](https://openmetal.io/resources/blog/building-high-throughput-data-ingestion-pipelines-with-kafka-on-openmetal/#:~:text=bin%2Fkafka,data)[[66]](https://openmetal.io/resources/blog/building-high-throughput-data-ingestion-pipelines-with-kafka-on-openmetal/#:~:text=,data).
* **Compression on Topic:** Kafka allows configuring compression.type at the topic level (for data at rest). Typically, if producers send compressed, the broker just stores that. You can set compression.type=producer on the topic to simply use whatever the producer sent[[67]](https://medium.com/javarevisited/5-minutes-engg-kafka-performance-10-tips-for-high-throughput-and-low-latency-2affb64a17a1#:~:text=1,Configure%20Producer%2FConsumer%2FBroker%20Threads%20%C2%B7%206). If you want the broker to recompress (e.g., to enforce a certain codec regardless of producer), you could set compression.type=lz4 or zstd on the topic[[67]](https://medium.com/javarevisited/5-minutes-engg-kafka-performance-10-tips-for-high-throughput-and-low-latency-2affb64a17a1#:~:text=1,Configure%20Producer%2FConsumer%2FBroker%20Threads%20%C2%B7%206). This might be useful if some producers are not compressing and you want to save space. However, recompression uses broker CPU, so if you can enforce producers to compress, do that instead for throughput.
* **Cleanup Policy (Retention vs Compaction):** By default, topics use cleanup.policy=delete with a time or size retention. This is suitable for streaming data where old events expire. Ensure your retention.bytes or retention.ms is set to something that balances business needs and disk usage. E.g., if 100 MB/s = ~8.6 GB per day, keeping 7 days means ~60 GB per topic partition (spread across brokers) – make sure you have disk for that multiplied by replication factor. For a robust cluster, also leave some disk headroom (don’t fill disks >80%). If you have a use-case for **log compaction** (keeping latest version of keys), you can set cleanup.policy=compact (or compact,delete for both). But be aware that compaction is a background process that will consume disk I/O and CPU to rewrite logs. For high throughput topics, compaction can lag if not enough resources. If you must use compaction, consider dedicating some broker threads to it (log.cleaner.threads=7 etc) and perhaps throttle it with log.cleaner.io.max.bytes.per.second to mitigate impact. For pure high-rate event streaming, compaction is usually off.
* **Default Partition Count:** Optionally, you can set a cluster-wide default for partitions (num.partitions in broker config) which is used when topics are auto-created. In a controlled environment, it’s better to explicitly create topics with the desired partition count (and replication) rather than rely on auto-create. This way, you can plan and document the partitioning. Use the kafka-topics.sh --create ... --partitions N --replication-factor R command (or Admin API) as part of your deployment or pipeline provisioning process.

## 7. Integrating Kafka Connect (Ingest/Egress Connectors)

Kafka Connect is a framework to stream data in and out of Kafka (to databases, files, etc.). In a high-throughput system, **Connect nodes and connectors must be configured and scaled appropriately** to not become a bottleneck. Key considerations for Kafka Connect in our scenario:

* **Deploying Kafka Connect Cluster:** Run Connect in **distributed mode** on one or more dedicated worker nodes (could be VMs or bare metal). Ensure these workers have sufficient CPU and memory, as they will handle serialization, external system communication, and use Kafka client under the hood. For 100 MB/s, you likely need multiple Connect workers, depending on connector type and external system throughput.
* **Tasks and Parallelism:** Each connector can be parallelized into multiple **tasks**. The **throughput of a connector is roughly bounded by the number of tasks and the throughput of each task**. For **source connectors** (bringing data into Kafka), tasks produce to Kafka topics – you can scale tasks up to the number of input partitions or parallel streams available from the source. For **sink connectors**, tasks are essentially Kafka consumers – their count is limited by the number of Kafka partitions they read from (a sink with tasks > partitions will have some idle tasks)[[60]](https://stackoverflow.com/questions/41900694/ideal-value-for-kafka-connect-distributed-tasks-max-configuration-setting#:~:text=In%20a%20Kafka%20Connect%20sink%2C,of%20the%20topics%20it%27s%20reading)[[61]](https://stackoverflow.com/questions/41900694/ideal-value-for-kafka-connect-distributed-tasks-max-configuration-setting#:~:text=partitions%20to%20read%20from,of%20the%20topics%20it%27s%20reading). **Aim to set tasks.max = (number of Kafka partitions)** for sink connectors to fully parallelize consumption[[60]](https://stackoverflow.com/questions/41900694/ideal-value-for-kafka-connect-distributed-tasks-max-configuration-setting#:~:text=In%20a%20Kafka%20Connect%20sink%2C,of%20the%20topics%20it%27s%20reading). If a sink handles multiple topics, consider partitions of the busiest topic. For source connectors, refer to the connector’s docs – e.g., a JDBC source might parallelize by table splits; tasks.max is the upper bound, but the connector might only create tasks as meaningful. **In summary, use as many tasks as needed to utilize your Connect cluster’s CPU and the external systems’ capacity, but not more than partitions** (for sinks) or logical streams (for sources).
* **Internal Producer/Consumer Tuning:** Kafka Connect’s source connectors internally use a Kafka producer to write to the cluster, and sink connectors use a consumer to read from the cluster. These internal clients can be tuned just like any other. **Kafka Connect allows passing client configuration via producer.override. and consumer.override. prefix** in the connector config[[47]](https://docs.redhat.com/en/documentation/red_hat_streams_for_apache_kafka/2.7/html/kafka_configuration_tuning/con-high-volume-config-properties-str#:~:text=Use%20,under%20the%20maximum%20batch%20size)[[68]](https://docs.redhat.com/en/documentation/red_hat_streams_for_apache_kafka/2.7/html/kafka_configuration_tuning/con-high-volume-config-properties-str#:~:text=Configure%20the%20,to%20the%20target%20Kafka%20cluster). For example, for a source connector, you might set producer.override.batch.size=131072 and producer.override.linger.ms=50 to batch up outgoing messages from the source before writing[[47]](https://docs.redhat.com/en/documentation/red_hat_streams_for_apache_kafka/2.7/html/kafka_configuration_tuning/con-high-volume-config-properties-str#:~:text=Use%20,under%20the%20maximum%20batch%20size)[[68]](https://docs.redhat.com/en/documentation/red_hat_streams_for_apache_kafka/2.7/html/kafka_configuration_tuning/con-high-volume-config-properties-str#:~:text=Configure%20the%20,to%20the%20target%20Kafka%20cluster). For a sink connector, you could set consumer.override.fetch.min.bytes=1048576 and so on. Essentially, apply the producer and consumer best practices (batching, compression, fetch sizes) to the connectors via these overrides so that the connector isn’t using default settings that could throttle throughput. Many connectors also have their own batch or fetch settings (e.g., JDBC source has batch.max.rows). Tune those as well to move data in chunks rather than one row at a time[[69]](https://www.confluent.io/blog/how-to-increase-throughput-on-kafka-connect-source-connectors/#:~:text=Connectors%20www,not%20expose%20this%20configuration).
* **Ensure Sufficient Throughput to External Systems:** Often the bottleneck is not Kafka at all, but the system Kafka Connect is reading from or writing to (database, REST API, etc.). For a robust pipeline, ensure those systems (and network to them) can handle the needed rate. If a sink (say, HDFS or S3) is slow, you might need multiple connectors or an optimized connector implementation to reach 100 MB/s. Monitor connector-specific metrics (like tasks throughput) in addition to Kafka metrics.
* **Monitoring Connect:** Kafka Connect exposes metrics per connector and task (e.g., records read/written per second, errors). Make sure to include the Connect process in your monitoring stack (JMX exporter can grab Connect metrics too). Also, if running Connect on JVM, same JVM tuning (heap, GC) considerations apply. A Connect worker might need a larger heap if it buffers a lot of data (e.g., the Kafka Connect FilePulse or other connectors that do transformations).

## 8. Monitoring and Observability

Building a robust Kafka cluster is not complete without **comprehensive monitoring**. High throughput systems can exhibit subtle performance issues, and early detection of problems (or saturation) is crucial. We recommend setting up a **Prometheus + Grafana** stack (or an equivalent monitoring solution) to collect and visualize metrics from Kafka brokers, ZooKeeper (if applicable), Kafka Connect, and the host systems.

**Metrics to Monitor:** Key metrics to track include **throughput metrics** (bytes in/out per second), **latencies**, **resource utilization**, and **lag**:

* **Broker Throughput:** Monitor broker BytesInPerSec and BytesOutPerSec for each broker[[70]](https://aws.amazon.com/blogs/big-data/best-practices-for-right-sizing-your-apache-kafka-clusters-to-optimize-performance-and-cost/#:~:text=rebalanced%20within%20the%20cluster%2C%20they,for%20the%20current%20cluster%20size). This tells you how much data each broker is ingesting and serving. It should roughly match your expectations (e.g., ~100 MB/s total in across cluster). If one broker is handling disproportionately more, you might have an imbalance of partitions. Also track **ReplicationBytesIn/OutPerSec**[[70]](https://aws.amazon.com/blogs/big-data/best-practices-for-right-sizing-your-apache-kafka-clusters-to-optimize-performance-and-cost/#:~:text=rebalanced%20within%20the%20cluster%2C%20they,for%20the%20current%20cluster%20size) – the inter-broker replication traffic. This helps verify that replication is keeping up and quantify overhead of replication.
* **Request Rates and Latency:** Kafka provides metrics for request handler rates and queue time. For example, RequestHandlerAvgIdlePercent per broker should ideally stay >30%[[32]](https://openmetal.io/resources/blog/building-high-throughput-data-ingestion-pipelines-with-kafka-on-openmetal/#:~:text=,on%20your%20bare%20metal%20server) – if it drops too low, the broker is near CPU saturation on request handling. The MessageConversionsTime and other request timings can indicate if brokers are struggling. Additionally, measure end-to-end latency if possible (producer send to consumer receive). Tools like Kafka’s end-to-end latency JMX metrics or simply measuring in the application can be used.
* **Consumer Lag:** Perhaps the most important metric for streaming reliability is **consumer lag**. Monitor the lag of each consumer group (how far behind the head of the topic they are). Kafka’s \_\_consumer\_offsets internal topic holds this info, and tools or Kafka Exporter can expose lag metrics. Set alerts for when lag grows beyond a threshold, as it indicates consumers are not keeping up with producers. For example, if lag goes from 0 to millions, something’s wrong downstream.
* **System Resources:** Monitor **CPU usage**, **disk I/O throughput and latency**, **network usage**, and **memory** on each broker machine. This can be done via node exporter (for Prometheus) or other agents. High CPU might mean compression or GC pressure; high disk I/O or latency might indicate nearing the disk limit. Network utilization should be well under the NIC capacity (e.g., on 10GbE, 100 MB/s is ~8% of capacity). If it’s much higher, or if you see packet drops, you might need tuning or more headroom.
* **JMX Metrics:** Kafka’s JMX metrics are rich. Use a JMX exporter to pull them into Prometheus[[71]](https://grafana.com/docs/grafana-cloud/knowledge-graph/enable-prom-metrics-collection/messaging-frameworks/kafka/#:~:text=1,following%20methods). Key broker metrics include KafkaServer:BrokerTopicMetrics for throughput and message rates, KafkaServer:ReplicaManager for replication lag, and KafkaNetwork for request metrics. Also track garbage collection (JVM GC time) and any warnings in broker logs.

**Grafana Dashboards:** Grafana can visualize these metrics in real time. You can use or adapt community dashboards (e.g., the Strimzi Kafka dashboard, which includes broker I/O, CPU, memory, and request stats, or Confluent’s Kafka monitoring dashboard). Important panels would be throughput (bytes in/out), partition count and leader distribution, under-replicated partitions (should be zero normally), consumer lag per group, and system metrics. Don’t forget Kafka Connect – monitor connector task throughputs and any errors (e.g., dead letter queue events if used).

**Alerting:** Set up alerts on critical conditions. For example: broker disk usage > 85%, broker offline, **under-replicated partitions > 0** (if a partition is not fully replicated, maybe a broker is down or slow), consumer lag above X for Y minutes, etc. Alerts ensure you can respond before an issue becomes an outage.

**Logging and Tracing:** In addition to metrics, ensure you have Kafka logs aggregated (brokers log INFO on startup, WARN on potential issues). If using Confluent or another platform, you might have audit logs for client connections, etc. For debugging performance, sometimes enabling request logging (at least sampling) can help trace slow requests. Some users integrate Kafka with distributed tracing (especially for apps using Kafka Streams). This can be complex, but tools like openTelemetry instrumentation for producers/consumers could trace a message’s journey (if needed for deep debugging in a complex microservice environment).

**Capacity Planning:** Use monitoring data to plan capacity. If you notice that daily traffic is increasing and your bytes in is regularly at 50% of network or disk capacity, you know when to add brokers or upgrade hardware. Monitoring trends allows you to scale proactively, keeping the cluster robust under growth.

**Expert Tip:** *Make monitoring a first-class citizen.* As one source puts it, *“Holistic monitoring is key – track Kafka JMX metrics, OS-level metrics, and application performance. Tools like Prometheus and Grafana are invaluable. Without comprehensive monitoring, tuning is just guesswork.”*[[72]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=Expert%20Insight%3A%20Holistic%20Monitoring%20is,Key)[[73]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=Performance%20tuning%20is%20an%20iterative,Without%20comprehensive%20monitoring%2C%20tuning%20is). In other words, gather data, then adjust configuration based on evidence.

## 9. Validation and Testing

Before and during production, validate your Kafka cluster’s performance:

* **Baseline Testing:** Produce and consume at the target rate (1000×100 KB messages/sec) in a staging environment to ensure the cluster meets the throughput with some headroom. Use tools like Kafka’s own performance tester (kafka-producer-perf-test.sh and kafka-consumer-perf-test.sh) or a custom load test. This can reveal if any bottleneck hits early (for example, maybe the disk isn’t as fast as expected, or GC causes hiccups at high heap usage).
* **Failure Scenarios:** Test broker failover. Bring one broker down and see if producers/consumers continue (with acks=all, some will fail until leader election occurs, which is expected). Ensure the system can catch up once the broker returns. This tests the robustness of replication and consumer offset management. Also test what happens if a broker is slow (simulate by throttling or heavy load) – do the others take more load gracefully?
* **Tune Based on Metrics:** If your tests show, say, high p99 latency, check if it’s due to batch settings or GC pauses. Adjust configurations accordingly. Kafka tuning is iterative; the settings given here are starting points, but your optimal values will depend on your specific data patterns (e.g., if all messages compress 10:1, compression is extremely beneficial; if not, CPU might outweigh benefits).
* **Observe in Production:** Once live, keep an eye on those metrics and adjust as needed. Perhaps you’ll find you can increase fetch.min.bytes even more, or you need to raise num.replica.fetchers if replication can’t keep up during peak, etc. Kafka is very configurable – use that to your advantage but change one thing at a time and observe.

## 10. Conclusion

Building a robust Kafka cluster for high-throughput streaming involves a combination of **solid infrastructure**, **well-chosen configurations**, and **vigilant monitoring**. On bare metal, you have the power to optimize hardware (fast disks, 10Gb networking, ample RAM) and OS parameters to give Kafka a strong foundation. A three-node (or larger) cluster with proper replication will tolerate failures without data loss, while tuned producer/consumer settings ensure every bit of throughput is squeezed out efficiently. We presented a range of tuning parameters – from batching and compression to thread pools and socket buffers – along with rationale and trade-offs.

Remember that there’s no single “set these magic values and forget it” – the optimal configuration emerges from **measuring your workload** and **iteratively adjusting**. Use the guidelines here as a starting point: for instance, enable LZ4 compression, start with 128 KB batches and 10 ms linger on producers, 1 MB fetch on consumers, etc., and then profile the system. The goal is to keep the cluster **stable (robust under failure)** and **performant (meeting throughput and latency requirements)**. By following these best practices and continuously monitoring the system’s health[[72]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=Expert%20Insight%3A%20Holistic%20Monitoring%20is,Key), you’ll be well on your way to running Kafka like a pro – handling 100 MB/s and beyond with confidence.

Lastly, don’t forget to plan for the future: if your throughput might double, have a scaling strategy (add partitions and brokers) ready. Kafka can scale to very high rates (many millions of messages/sec) when architected and tuned properly[[12]](https://aws.amazon.com/blogs/big-data/best-practices-for-right-sizing-your-apache-kafka-clusters-to-optimize-performance-and-cost/#:~:text=For%20production%20clusters%2C%20it%E2%80%99s%20a,cluster%20should%20target%20800%20MB%2Fsec)[[13]](https://aws.amazon.com/blogs/big-data/best-practices-for-right-sizing-your-apache-kafka-clusters-to-optimize-performance-and-cost/#:~:text=To%20find%20the%20right%20configuration,capacity%2C%20for%20instance%2C%20during%20maintenance). With the robust bare-metal cluster you’ve built and this handbook as a guide, you have a durable backbone for your distributed streaming system. Happy streaming!

**Sources:** The recommendations above are based on Kafka documentation and field best practices[[1]](https://openmetal.io/resources/blog/building-high-throughput-data-ingestion-pipelines-with-kafka-on-openmetal/#:~:text=Begin%20by%20deploying%20your%20on,cores%2C%20RAM%2C%20and%20%2021)[[2]](https://github.com/AutoMQ/automq/wiki/Kafka-Replication:-Concept-&-Best-Practices#:~:text=The%20replication%20factor%20is%20a,maintained%20across%20the%20Kafka%20cluster)[[25]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=3)[[55]](https://docs.redhat.com/en/documentation/red_hat_streams_for_apache_kafka/2.7/html/kafka_configuration_tuning/con-high-volume-config-properties-str#:~:text=For%20consumers%2C%20you%20might%20increase,property), including guidance from Confluent, Red Hat, New Relic, AWS, and experienced community engineers. All cited references provide additional depth on specific tuning aspects and can be consulted for further details.

[[1]](https://openmetal.io/resources/blog/building-high-throughput-data-ingestion-pipelines-with-kafka-on-openmetal/#:~:text=Begin%20by%20deploying%20your%20on,cores%2C%20RAM%2C%20and%20%2021) [[6]](https://openmetal.io/resources/blog/building-high-throughput-data-ingestion-pipelines-with-kafka-on-openmetal/#:~:text=Now%2C%20from%20one%20of%20the,partitions%20per%20core%20per%20broker) [[9]](https://openmetal.io/resources/blog/building-high-throughput-data-ingestion-pipelines-with-kafka-on-openmetal/#:~:text=Scaling%20Your%20Pipeline%20for%20Future,Growth) [[32]](https://openmetal.io/resources/blog/building-high-throughput-data-ingestion-pipelines-with-kafka-on-openmetal/#:~:text=,on%20your%20bare%20metal%20server) [[37]](https://openmetal.io/resources/blog/building-high-throughput-data-ingestion-pipelines-with-kafka-on-openmetal/#:~:text=,flight%20data) [[44]](https://openmetal.io/resources/blog/building-high-throughput-data-ingestion-pipelines-with-kafka-on-openmetal/#:~:text=,I%2FO%20from%20the%20operating%20system) [[65]](https://openmetal.io/resources/blog/building-high-throughput-data-ingestion-pipelines-with-kafka-on-openmetal/#:~:text=bin%2Fkafka,data) [[66]](https://openmetal.io/resources/blog/building-high-throughput-data-ingestion-pipelines-with-kafka-on-openmetal/#:~:text=,data) Building High-Throughput Data Ingestion Pipelines with Kafka on OpenMetal

<https://openmetal.io/resources/blog/building-high-throughput-data-ingestion-pipelines-with-kafka-on-openmetal/>

[[2]](https://github.com/AutoMQ/automq/wiki/Kafka-Replication:-Concept-&-Best-Practices#:~:text=The%20replication%20factor%20is%20a,maintained%20across%20the%20Kafka%20cluster) [[3]](https://github.com/AutoMQ/automq/wiki/Kafka-Replication:-Concept-&-Best-Practices#:~:text=,8) [[4]](https://github.com/AutoMQ/automq/wiki/Kafka-Replication:-Concept-&-Best-Practices#:~:text=Leaders%20and%20Followers%20Architecture) [[5]](https://github.com/AutoMQ/automq/wiki/Kafka-Replication:-Concept-&-Best-Practices#:~:text=The%20In,with%20the%20leader) Kafka Replication: Concept & Best Practices · AutoMQ/automq Wiki · GitHub

<https://github.com/AutoMQ/automq/wiki/Kafka-Replication:-Concept-&-Best-Practices>

[[7]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=4) [[8]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=,Kafka%20relies%20on%20the) [[14]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=,with%20Kafka%27s%20replication%20handling%20redundancy) [[15]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=,Direct%20disk%20access%20is%20preferred) [[17]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=are%20not%20extremely%20high,Direct%20disk%20access%20is%20preferred) [[18]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=,Direct%20disk%20access%20is%20preferred) [[19]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=redundancy,Direct%20disk%20access%20is%20preferred) [[20]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=1) [[21]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=2) [[22]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=5) [[23]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=Reduce%20,heap%2C%20which%20can%20cripple%20performance) [[24]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=4) [[25]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=3) [[26]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=,clients%2C%20brokers%2C%20and%20network%20devices) [[27]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=,clients%2C%20brokers%2C%20and%20network%20devices) [[28]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=6) [[29]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=Increase%20the%20open%20file%20descriptor,log%20segments%20and%20network%20connections) [[30]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=%28%60KAFKA_HEAP_OPTS%60%2C%20e.g.%2C%20%60,heavily%20for%20reads%20and%20writes) [[31]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=for%20reads%20and%20writes) [[33]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=2,threads) [[34]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=,can%20be%20a%20starting%20point) [[35]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=6) [[36]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=,processor%20before%20requests%20are%20blocked) [[38]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=6) [[39]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=5) [[40]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=,broker%20hosts%20many%20follower%20partitions) [[41]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=consistently%20high%2C%20increasing%20this%20can,%28default%3A%2064KB%29%3A%20Network%20receive) [[42]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=starting%20point%20is%20%60num_brokers%20,let%20the%20OS%20manage%20flushing) [[43]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=segments%20mean%20fewer%20files%2C%20potentially,let%20the%20OS%20manage%20flushing) [[46]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=2,Partitions) [[57]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=Parameter%20Default%20Tuning%20Guidance%20,heartbeat.interval.ms) [[62]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=3) [[63]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=4) [[64]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=,but%20requires%20careful%20error%20handling) [[72]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=Expert%20Insight%3A%20Holistic%20Monitoring%20is,Key) [[73]](https://activewizards.com/blog/advanced-kafka-performance-tuning#:~:text=Performance%20tuning%20is%20an%20iterative,Without%20comprehensive%20monitoring%2C%20tuning%20is) Advanced Kafka Performance Tuning | ActiveWizards: AI & Agent Engineering | Data Platforms

<https://activewizards.com/blog/advanced-kafka-performance-tuning>

[[10]](https://aws.amazon.com/blogs/big-data/best-practices-for-right-sizing-your-apache-kafka-clusters-to-optimize-performance-and-cost/#:~:text=most%20important%20aspects%20are%20as,follows) [[11]](https://aws.amazon.com/blogs/big-data/best-practices-for-right-sizing-your-apache-kafka-clusters-to-optimize-performance-and-cost/#:~:text=,sense%20to%20either%20increase%20the) [[12]](https://aws.amazon.com/blogs/big-data/best-practices-for-right-sizing-your-apache-kafka-clusters-to-optimize-performance-and-cost/#:~:text=For%20production%20clusters%2C%20it%E2%80%99s%20a,cluster%20should%20target%20800%20MB%2Fsec) [[13]](https://aws.amazon.com/blogs/big-data/best-practices-for-right-sizing-your-apache-kafka-clusters-to-optimize-performance-and-cost/#:~:text=To%20find%20the%20right%20configuration,capacity%2C%20for%20instance%2C%20during%20maintenance) [[70]](https://aws.amazon.com/blogs/big-data/best-practices-for-right-sizing-your-apache-kafka-clusters-to-optimize-performance-and-cost/#:~:text=rebalanced%20within%20the%20cluster%2C%20they,for%20the%20current%20cluster%20size) Best practices for right-sizing your Apache Kafka clusters to optimize performance and cost | AWS Big Data Blog

<https://aws.amazon.com/blogs/big-data/best-practices-for-right-sizing-your-apache-kafka-clusters-to-optimize-performance-and-cost/>

[[16]](https://github.com/AutoMQ/automq/wiki/Kafka-Performance-Tuning:-Tips-&-Best-Practices#:~:text=Reddit%20discussions%20highlight%20the%20performance,13) [[53]](https://github.com/AutoMQ/automq/wiki/Kafka-Performance-Tuning:-Tips-&-Best-Practices#:~:text=max.inflight.requests.per.connection%20,throughput%20but%20increases%20memory%20usage) [[54]](https://github.com/AutoMQ/automq/wiki/Kafka-Performance-Tuning:-Tips-&-Best-Practices#:~:text=Image) [[58]](https://github.com/AutoMQ/automq/wiki/Kafka-Performance-Tuning:-Tips-&-Best-Practices#:~:text=Parallel%20Consumption) [[59]](https://github.com/AutoMQ/automq/wiki/Kafka-Performance-Tuning:-Tips-&-Best-Practices#:~:text=For%20topics%20with%20multiple%20partitions%2C,2) Kafka Performance Tuning: Tips & Best Practices · AutoMQ/automq Wiki · GitHub

<https://github.com/AutoMQ/automq/wiki/Kafka-Performance-Tuning:-Tips-&-Best-Practices>

[[45]](https://docs.redhat.com/en/documentation/red_hat_streams_for_apache_kafka/2.7/html/kafka_configuration_tuning/con-high-volume-config-properties-str#:~:text=For%20producers%2C%20you%20might%20increase,handle%20large%20numbers%20of%20messages) [[47]](https://docs.redhat.com/en/documentation/red_hat_streams_for_apache_kafka/2.7/html/kafka_configuration_tuning/con-high-volume-config-properties-str#:~:text=Use%20,under%20the%20maximum%20batch%20size) [[55]](https://docs.redhat.com/en/documentation/red_hat_streams_for_apache_kafka/2.7/html/kafka_configuration_tuning/con-high-volume-config-properties-str#:~:text=For%20consumers%2C%20you%20might%20increase,property) [[56]](https://docs.redhat.com/en/documentation/red_hat_streams_for_apache_kafka/2.7/html/kafka_configuration_tuning/con-high-volume-config-properties-str#:~:text=For%20producers%2C%20you%20might%20increase,when%20you%20have%20just%20a) [[68]](https://docs.redhat.com/en/documentation/red_hat_streams_for_apache_kafka/2.7/html/kafka_configuration_tuning/con-high-volume-config-properties-str#:~:text=Configure%20the%20,to%20the%20target%20Kafka%20cluster) Chapter 6. Handling high volumes of messages | Kafka configuration tuning | Streams for Apache Kafka | 2.7 | Red Hat Documentation

<https://docs.redhat.com/en/documentation/red_hat_streams_for_apache_kafka/2.7/html/kafka_configuration_tuning/con-high-volume-config-properties-str>

[[48]](https://stackoverflow.com/questions/70980719/kafka-is-losing-message-when-ack-is-one#:~:text=Kafka%20is%20losing%20message%20when,Kakfa%20Producer%20Message%20Delivery) Kafka is losing message when ack is one - Stack Overflow

<https://stackoverflow.com/questions/70980719/kafka-is-losing-message-when-ack-is-one>

[[49]](https://forum.confluent.io/t/why-losing-messages-on-read/2759#:~:text=Kafka%20only%20guarantees%20that%20a,ack%20back%20before%20replication%20finished) [[51]](https://forum.confluent.io/t/why-losing-messages-on-read/2759#:~:text=Kafka%20only%20guarantees%20that%20a,ack%20back%20before%20replication%20finished) Why losing messages on read? - Confluent Community

<https://forum.confluent.io/t/why-losing-messages-on-read/2759>

[[50]](https://www.confluent.io/blog/5-things-every-kafka-developer-should-know/#:~:text=Confluent%20www,to%20using%20the%20acks%3Dall) [[52]](https://www.confluent.io/blog/5-things-every-kafka-developer-should-know/#:~:text=Confluent%20www,to%20using%20the%20acks%3Dall) Top 5 Things Every Apache Kafka Developer Should Know - Confluent

<https://www.confluent.io/blog/5-things-every-kafka-developer-should-know/>

[[60]](https://stackoverflow.com/questions/41900694/ideal-value-for-kafka-connect-distributed-tasks-max-configuration-setting#:~:text=In%20a%20Kafka%20Connect%20sink%2C,of%20the%20topics%20it%27s%20reading) [[61]](https://stackoverflow.com/questions/41900694/ideal-value-for-kafka-connect-distributed-tasks-max-configuration-setting#:~:text=partitions%20to%20read%20from,of%20the%20topics%20it%27s%20reading) amazon s3 - Ideal value for Kafka Connect Distributed tasks.max configuration setting? - Stack Overflow

<https://stackoverflow.com/questions/41900694/ideal-value-for-kafka-connect-distributed-tasks-max-configuration-setting>

[[67]](https://medium.com/javarevisited/5-minutes-engg-kafka-performance-10-tips-for-high-throughput-and-low-latency-2affb64a17a1#:~:text=1,Configure%20Producer%2FConsumer%2FBroker%20Threads%20%C2%B7%206) Kafka Performance — 10 Tips for High Throughput and Low Latency

<https://medium.com/javarevisited/5-minutes-engg-kafka-performance-10-tips-for-high-throughput-and-low-latency-2affb64a17a1>

[[69]](https://www.confluent.io/blog/how-to-increase-throughput-on-kafka-connect-source-connectors/#:~:text=Connectors%20www,not%20expose%20this%20configuration) Kafka Connect: How to Increase Throughput on Source Connectors

<https://www.confluent.io/blog/how-to-increase-throughput-on-kafka-connect-source-connectors/>

[[71]](https://grafana.com/docs/grafana-cloud/knowledge-graph/enable-prom-metrics-collection/messaging-frameworks/kafka/#:~:text=1,following%20methods) Kafka | Grafana Cloud documentation

<https://grafana.com/docs/grafana-cloud/knowledge-graph/enable-prom-metrics-collection/messaging-frameworks/kafka/>