Kafka Low Level Architecture

Kafka Low-Level Design

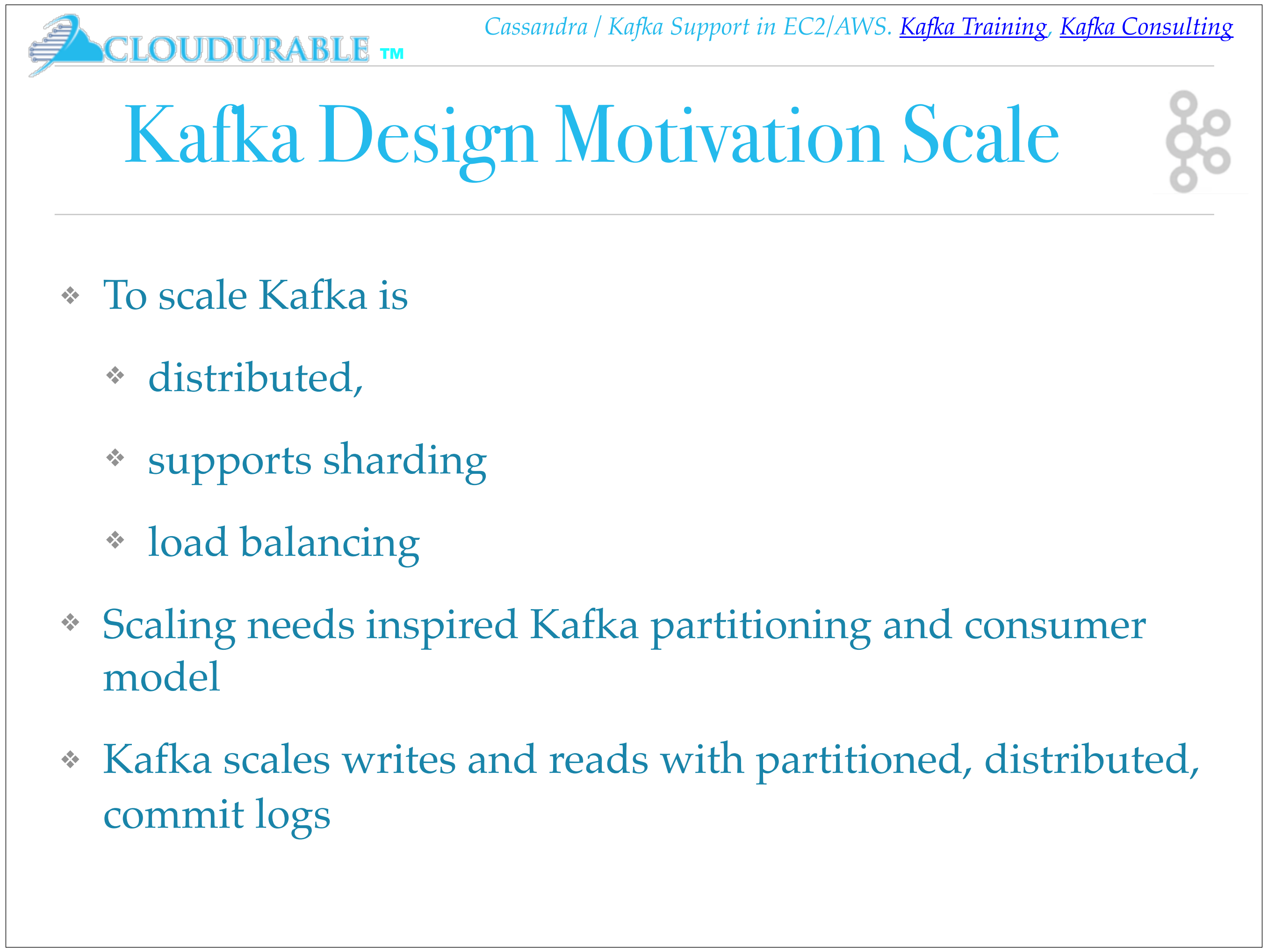
Design discussion of Kafka low level design

Kafka Architecture: Low-Level Design

Kafka Design Motivation Goals

- Kafka built to support real-time analytics
  - Designed to feed analytics system that did real-time processing of streams
  - Unified platform for real-time handling of streaming data feeds
- Goals:
  - high-throughput streaming data platform
  - supports high-volume event streams like log aggregation, user activity, etc.

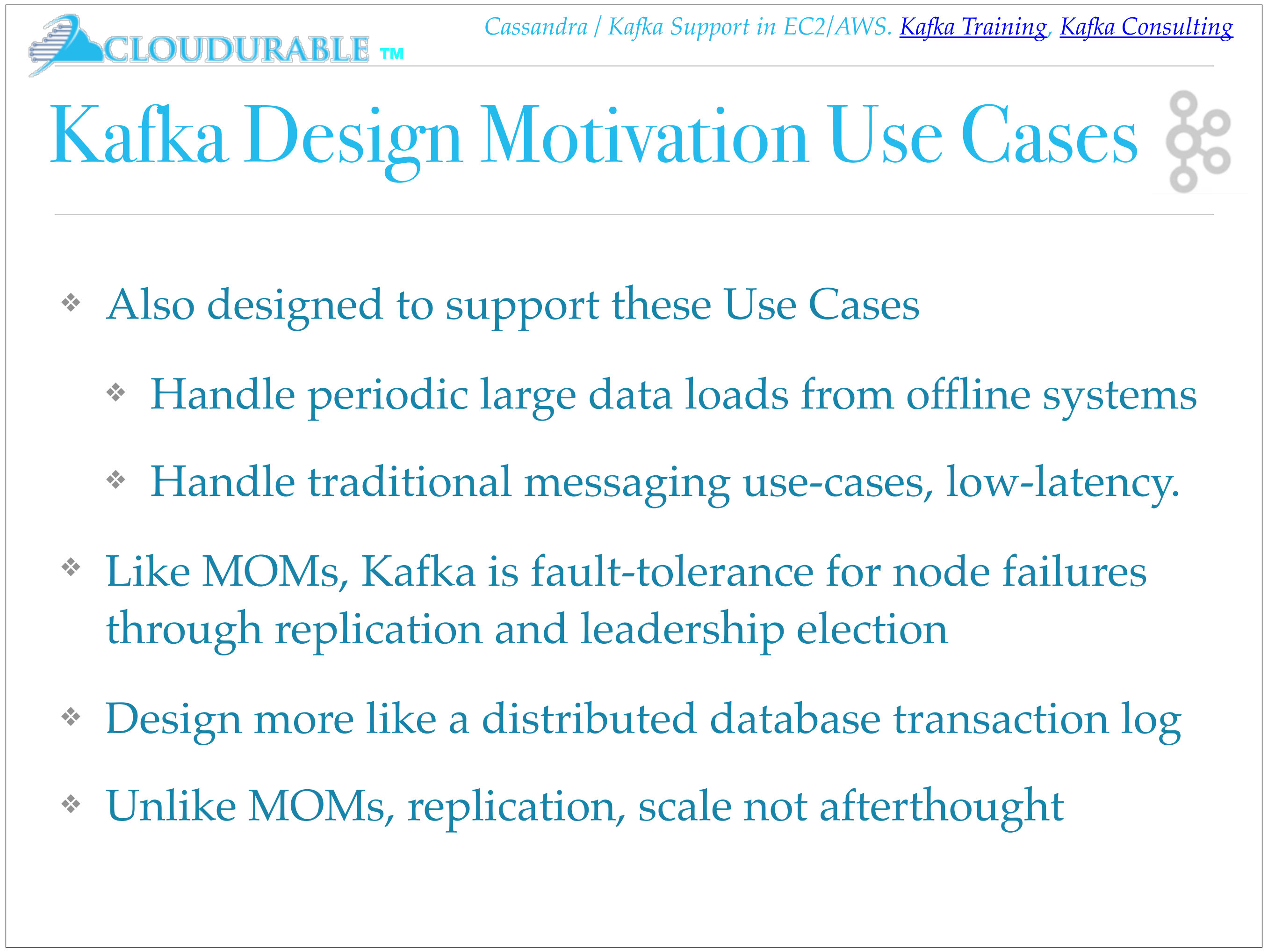
LinkedIn engineering built Kafka to support real-time analytics. Kafka was designed to feed analytics system that did real-time processing of streams. LinkedIn developed Kafka as a unified platform for real-time handling of streaming data feeds. The goal behind Kafka, build a high-throughput streaming data platform that supports high-volume event streams like log aggregation, user activity, etc.



Kafka Design Motivation Scale

- To scale Kafka is
  - distributed,
  - supports sharding
  - load balancing
- Scaling needs inspired Kafka partitioning and consumer model
- Kafka scales writes and reads with partitioned, distributed, commit logs

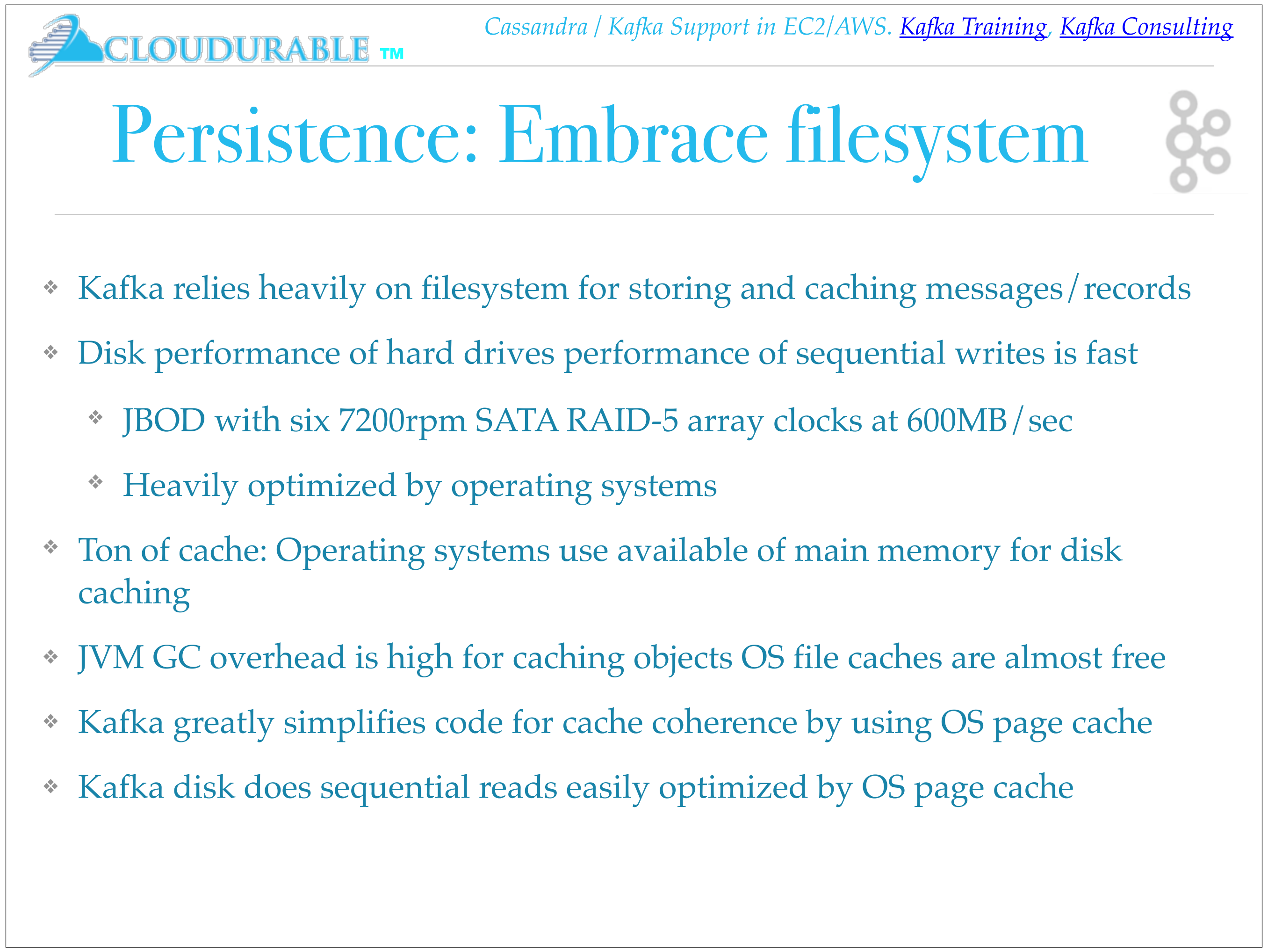
To scale to meet the demands of LinkedIn Kafka is distributed, supports sharding and load balancing. Scaling needs inspired Kafka's partitioning and consumer model. Kafka scales writes and reads with partitioned, distributed, commit logs.



Kafka Design Motivation Use Cases

- Also designed to support these Use Cases
  - Handle periodic large data loads from offline systems
  - Handle traditional messaging use-cases, low-latency.
- Like MOMs, Kafka is fault-tolerance for node failures through replication and leadership election
- Design more like a distributed database transaction log
- Unlike MOMs, replication, scale not afterthought

Kafka was designed to handle periodic large data loads from offline systems as well as traditional messaging use-cases, low-latency. MOM is message oriented middleware think IBM MQSeries, JMS (http://cloudurable.com/blog/kafka-vs-jms/index.html), ActiveMQ, and RabbitMQ. Like many MOMs, Kafka is fault-tolerance for node failures through replication and leadership election. However, the design of Kafka is more like a distributed database transaction log than a traditional messaging system. Unlike many MOMs, Kafka replication was built into the low-level design and is not an afterthought.



Persistence: Embrace filesystem

- Kafka relies heavily on filesystem for storing and caching messages/records
- Disk performance of hard drives performance of sequential writes is fast
  - JBOD with six 7200rpm SATA RAID-5 array clocks at 600MB/sec
  - Heavily optimized by operating systems
- Ton of cache: Operating systems use available of main memory for disk caching
- JVM GC overhead is high for caching objects OS file caches are almost free
- Kafka greatly simplifies code for cache coherence by using OS page cache
- Kafka disk does sequential reads easily optimized by OS page cache

Kafka relies on the filesystem for storing and caching records. The disk performance of hard drives performance of sequential writes is fast (https://mechanical-sympathy.blogspot.com/2011/12/java-sequential-io-performance.html). JBOD is just a bunch of disk drives. JBOD configuration with six 7200rpm SATA RAID-5 array is about 600MB/sec. Like Cassandra tables, Kafka logs are write only structures, meaning, data gets appended to the end of the log. When using HDD, sequential reads and writes are fast, predictable, and heavily optimized by operating systems. Using HDD, sequential disk access can be faster than random memory access and SSD. While JVM GC overhead can be high, Kafka leans on the OS a lot for caching, which is big, fast and rock solid cache.

Also, modern operating systems use all available main memory for disk caching. OS file caches are almost free and don't have the overhead of the OS. Implementing cache coherency is challenging to get right, but Kafka relies on the rock solid OS for cache coherence. Using the OS for cache also reduces the number of buffer copies. Since Kafka disk usage tends to do sequential reads, the OS read-ahead cache is impressive.

Big fast HDDs and long sequential access

- Like Cassandra, LevelDB, RocksDB, and others, Kafka uses long sequential disk access for read and writes
- Kafka uses tombstones instead of deleting records right away
- Modern Disks have somewhat unlimited space and are fast
- Kafka can provide features not usually found in a messaging system like holding on to old messages for a really long time
    - This flexibility allows for interesting application of Kafka

Kafka favors long sequential disk access for reads and writes.
Like Cassandra, LevelDB, RocksDB, and others, Kafka uses a form of log structured storage and compaction instead of an on-disk mutable BTree. Like Cassandra, Kafka uses tombstones instead of deleting records right away.

Since disks these days have somewhat unlimited space and are very fast, Kafka can provide features not usually found in a messaging system like holding on to old messages for a long time. This flexibility allows for interesting application of Kafka.

Kafka Record Retention Redux

- Kafka cluster retains all published records
  - Time based – configurable retention period
  - Size based - configurable based on size
  - Compaction - keeps latest record
- Kafka uses Topic ***Partitions***
- Partitions are broken down into ***Segment*** files

Broker Log Config

Kafka Broker Config for Logs

NAME	DESCRIPTION	DEFAULT
log.dir	Log Directory will topic logs will be stored use this or log.dirs.	/tmp/kafka-logs
log.dirs	The directories where the Topics logs are kept used for JBOD.	
log.flush.interval.messages	Accumulated messages count on a log partition before messages are flushed to disk.	9,223,372,036,854,780,000
log.flush.interval.ms	Maximum time that a topic message is kept in memory before flushed to disk. If not set, uses log.flush.scheduler.interval.ms.	
log.flush.offset.checkpoint.interval.ms	Interval to flush log recovery point.	60,000
log.flush.scheduler.interval.ms	Interval that topic messages are periodically flushed from memory to log.	9,223,372,036,854,780,000

Broker Log Retention Config

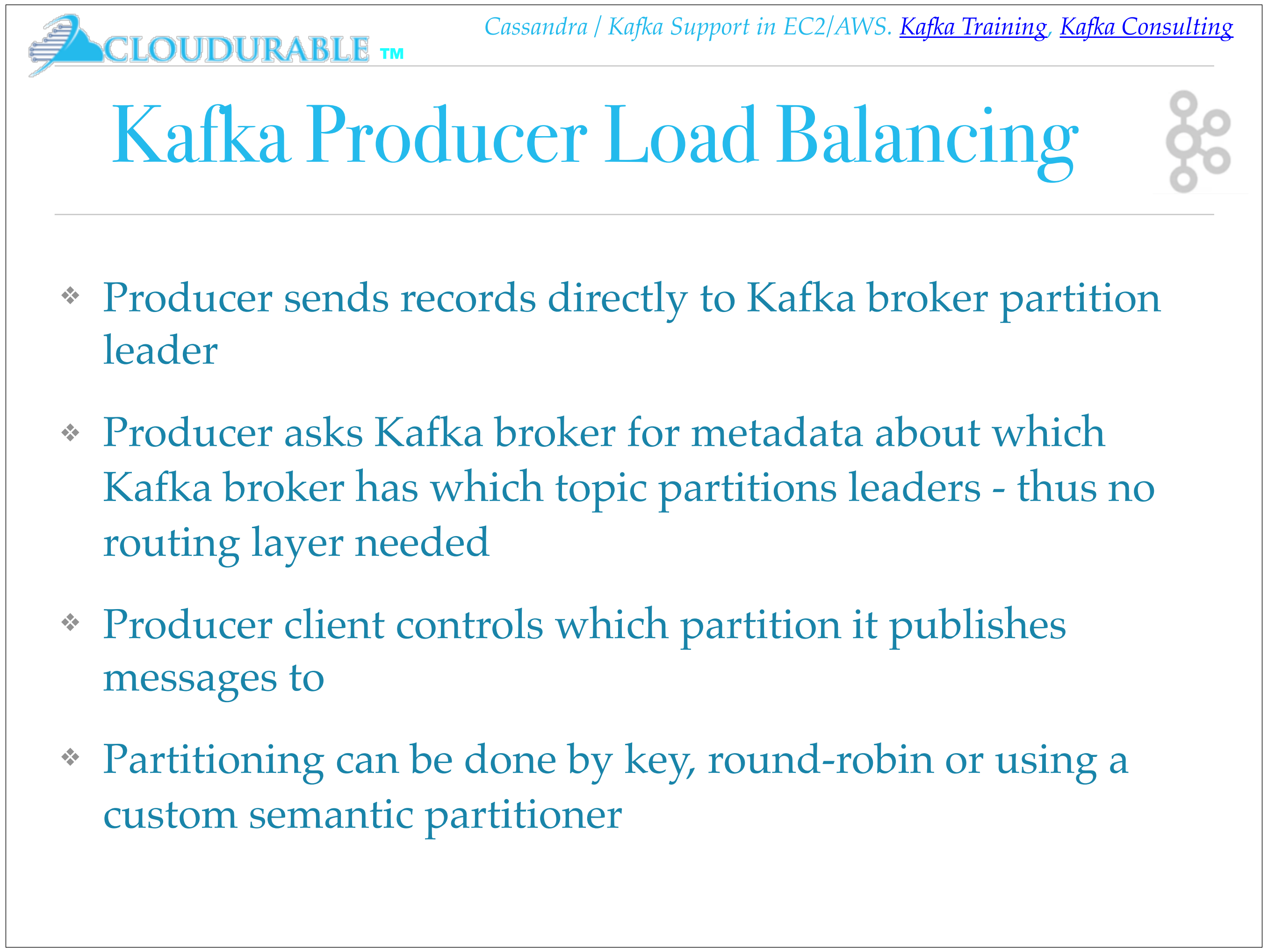
Kafka Broker Config for Logs

log.retention.bytes	Delete log records by size. The maximum size of the log before deleting its older records.	long	-1
log.retention.hours	Delete log records by time hours. Hours to keep a log file before deleting older records (in hours), tertiary to log.retention.ms property.	int	168
log.retention.minutes	Delete log records by time minutes. Minutes to keep a log file before deleting it, secondary to log.retention.ms property. If not set, use log.retention.hours is used.	int	null
log.retention.ms	Delete log records by time milliseconds. Milliseconds to keep a log file before deleting it, If not set, use log.retention.minutes.	long	null

Broker Log Segment File Config

Kafka Broker Config - Log Segments

NAME	DESCRIPTION	TYPE	DEFAULT
log.roll.hours	Time period before rolling a new topic log segment. (secondary to log.roll.ms property)	int	168
log.roll.ms	Time period in milliseconds before rolling a new log segment. If not set, uses log.roll.hours.	long	
log.segment.bytes	The maximum size of a single log segment file.	int	1,073,741,824
log.segment.delete.delay.ms	Time period to wait before deleting a segment file from the filesystem.	long	60,000

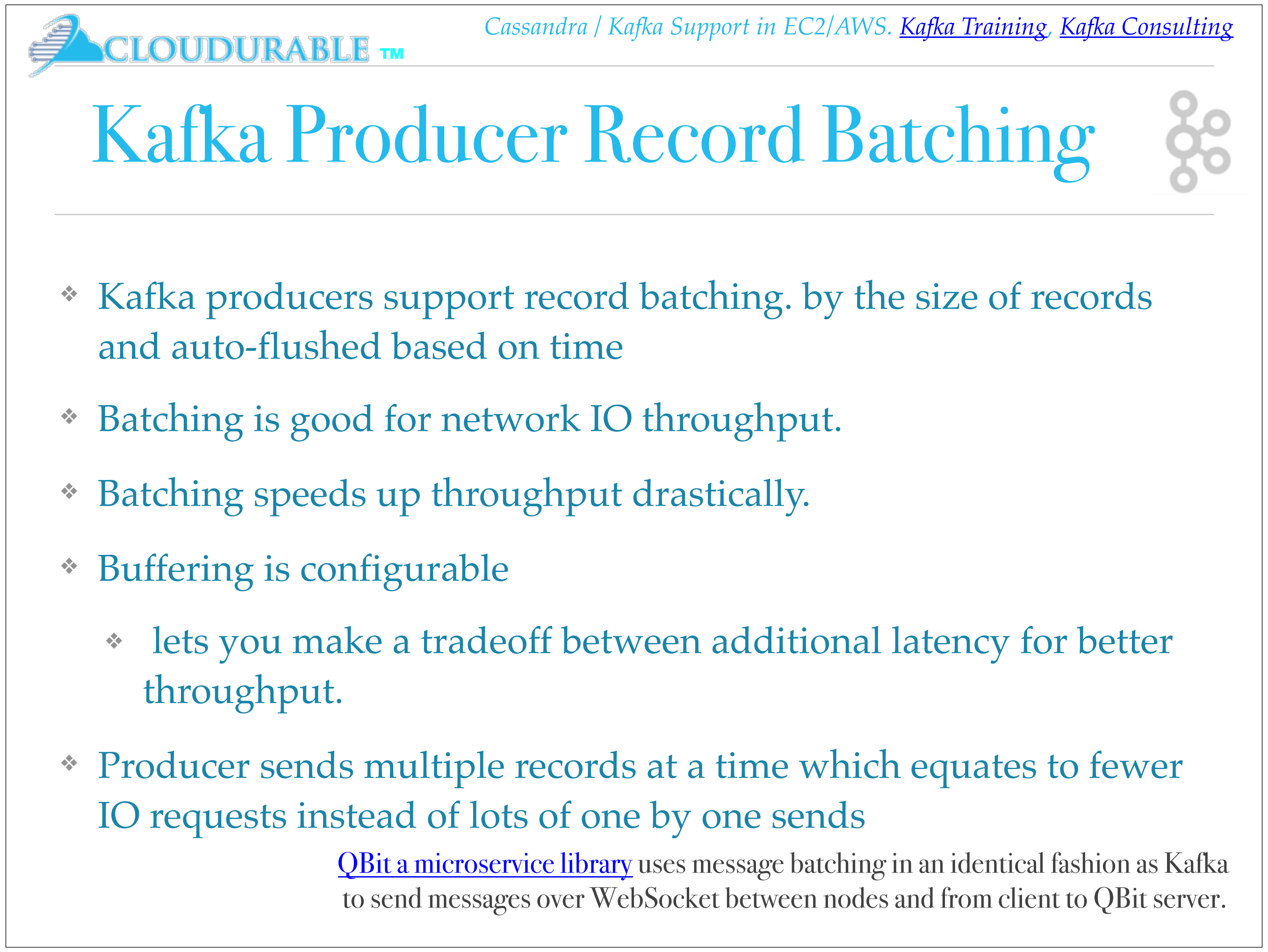


Kafka Producer Load Balancing

- Producer sends records directly to Kafka broker partition leader
- Producer asks Kafka broker for metadata about which Kafka broker has which topic partitions leaders - thus no routing layer needed
- Producer client controls which partition it publishes messages to
- Partitioning can be done by key, round-robin or using a custom semantic partitioner

The producer asks the Kafka broker for metadata about which Kafka broker has which topic partitions leaders thus no routing layer needed. This leadership data allows the producer to send records directly to Kafka broker partition leader.

The Producer client controls which partition it publishes messages to, and can pick a partition based on some application logic. Producers can partition records by key, round-robin or use a custom application-specific partitioner logic.



Kafka Producer Record Batching

- Kafka producers support record batching. by the size of records and auto-flushed based on time
- Batching is good for network IO throughput.
- Batching speeds up throughput drastically.
- Buffering is configurable
  - lets you make a tradeoff between additional latency for better throughput.
- Producer sends multiple records at a time which equates to fewer IO requests instead of lots of one by one sends

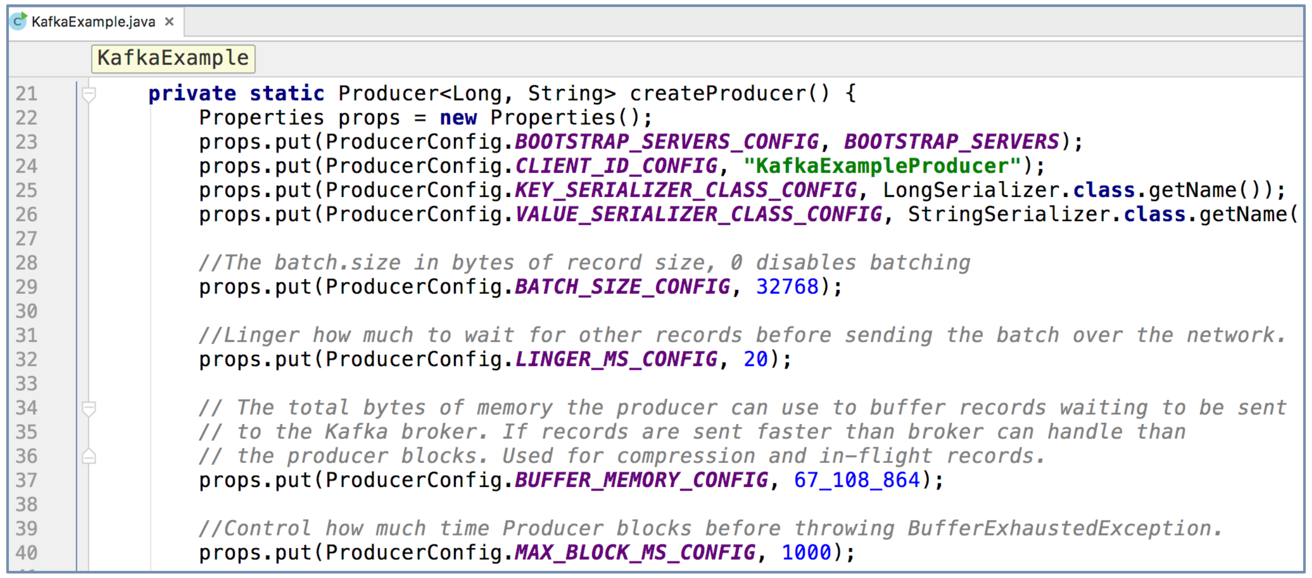
QBit a microservice library uses message batching in an identical fashion as Kafka to send messages over WebSocket between nodes and from client to QBit server.

Kafka producers support record batching. Batching can be configured by the size of records in bytes in batch. Batches can be auto-flushed based on time.

Batching is good for network IO throughput. Batching speeds up throughput drastically.

Buffering is configurable and lets you make a tradeoff between additional latency for better throughput. Or in the case of a heavily used system, it could be both better average throughput and reduces overall latency. Batching allows accumulation of more bytes to send, which equate to few larger I/O operations on Kafka Brokers and increase compression efficiency. For higher throughput, Kafka Producer configuration allows buffering based on time and size. The producer sends multiple records as a batch with fewer network requests than sending each record one by one.

More producer settings for performance



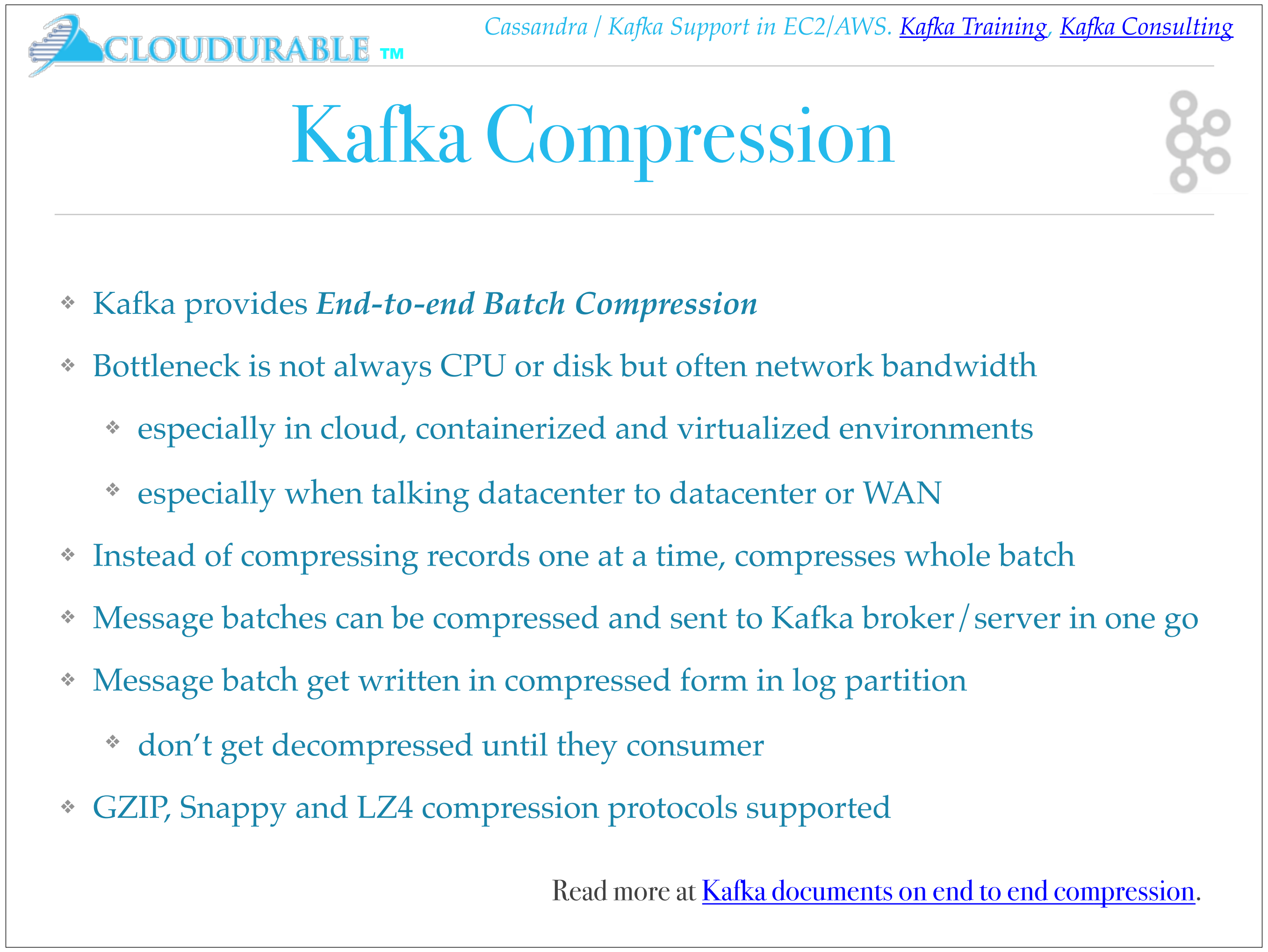
KafkaExample.java ×

KafkaExample

```java
    private static Producer<Long, String> createProducer() {
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
        props.put(ProducerConfig.CLIENT_ID_CONFIG, "KafkaExampleProducer");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, LongSerializer.class.getName());
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName(

        //The batch.size in bytes of record size, 0 disables batching
        props.put(ProducerConfig.BATCH_SIZE_CONFIG, 32768);

        //Linger how much to wait for other records before sending the batch over the network.
        props.put(ProducerConfig.LINGER_MS_CONFIG, 20);

        // The total bytes of memory the producer can use to buffer records waiting to be sent
        // to the Kafka broker. If records are sent faster than broker can handle than
        // the producer blocks. Used for compression and in-flight records.
        props.put(ProducerConfig.BUFFER_MEMORY_CONFIG, 67_108_864);

        //Control how much time Producer blocks before throwing BufferExhaustedException.
        props.put(ProducerConfig.MAX_BLOCK_MS_CONFIG, 1000);
```

For higher throughput, Kafka Producer allows buffering based on time and size.
Multiple records can be sent as a batches with fewer network requests.
Speeds up throughput drastically.

For higher throughput, Kafka Producer allows buffering based on time and size.
Multiple records can be sent as a batches with fewer network requests.
Batching speeds up throughput drastically.



Kafka Compression

- Kafka provides ***End-to-end Batch Compression***
- Bottleneck is not always CPU or disk but often network bandwidth
  - especially in cloud, containerized and virtualized environments
  - especially when talking datacenter to datacenter or WAN
- Instead of compressing records one at a time, compresses whole batch
- Message batches can be compressed and sent to Kafka broker/server in one go
- Message batch get written in compressed form in log partition
  - don't get decompressed until they consumer
- GZIP, Snappy and LZ4 compression protocols supported

Read more at Kafka documents on end to end compression.

In large streaming platforms, the bottleneck is not always CPU or disk but often network bandwidth. There is even more network bandwidth issues in cloud, containerized and virtualized environments as multiple services could be sharing a NiC card. Also, network bandwidth issues can be problematic when talking datacenter to datacenter or WAN. Batching is beneficial for efficient compression and network IO throughput.

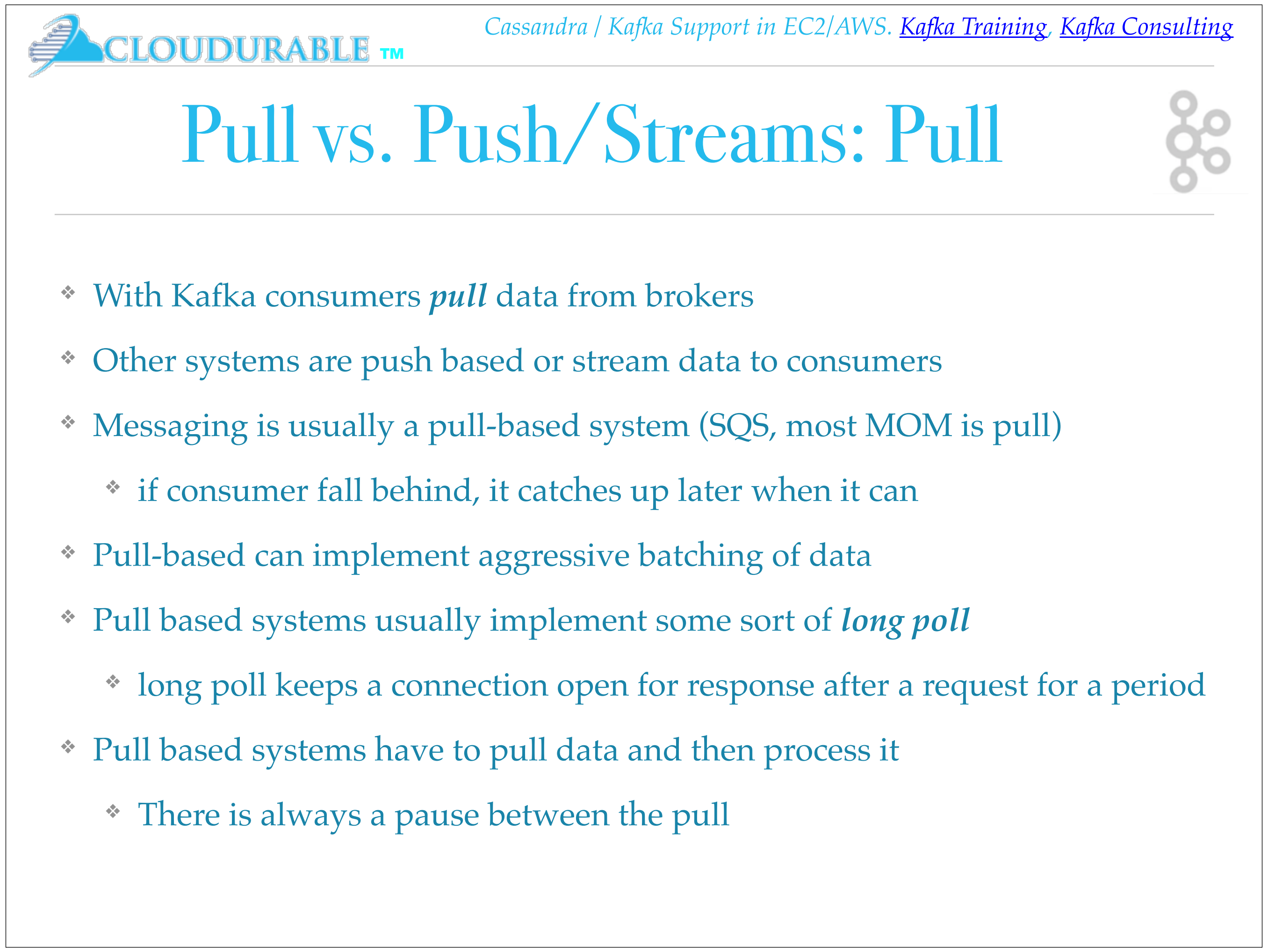
Kafka provides end-to-end batch compression instead of compressing a record at a time, Kafka efficiently compresses a whole batch of records. The same message batch can be compressed and sent to Kafka broker/server in one go and written in compressed form into the log partition. You can even configure the compression so that no decompression happens until the Kafka broker delivers the compressed records to the consumer.

Kafka supports GZIP, Snappy and LZ4 compression protocols.

Kafka Compression Config

Kafka Broker Compression Config

compression.type	Configures compression type for topics. Can be set to codecs ‘gzip', 'snappy', ‘lz4' or ‘uncompressed’. If set to 'producer' then it retains compression codec set by the producer (so it does not have to be uncompressed and then recompressed).	Default: producer



Pull vs. Push/Streams: Pull

- With Kafka consumers ***pull*** data from brokers
- Other systems are push based or stream data to consumers
- Messaging is usually a pull-based system (SQS, most MOM is pull)
  - if consumer fall behind, it catches up later when it can
- Pull-based can implement aggressive batching of data
- Pull based systems usually implement some sort of ***long poll***
  - long poll keeps a connection open for response after a request for a period
- Pull based systems have to pull data and then process it
  - There is always a pause between the pull

With Kafka consumers pull data from brokers. Other systems brokers push data or stream data to consumers. Messaging is usually a pull-based system (SQS, most MOM use pull). With the pull-based system, if a consumer falls behind, it catches up later when it can.

Since Kafka is pull-based, it implements aggressive batching of data. Kafka like many pull based systems implements a long poll (SQS, Kafka both do). A long poll keeps a connection open after a request for a period and waits for a response.

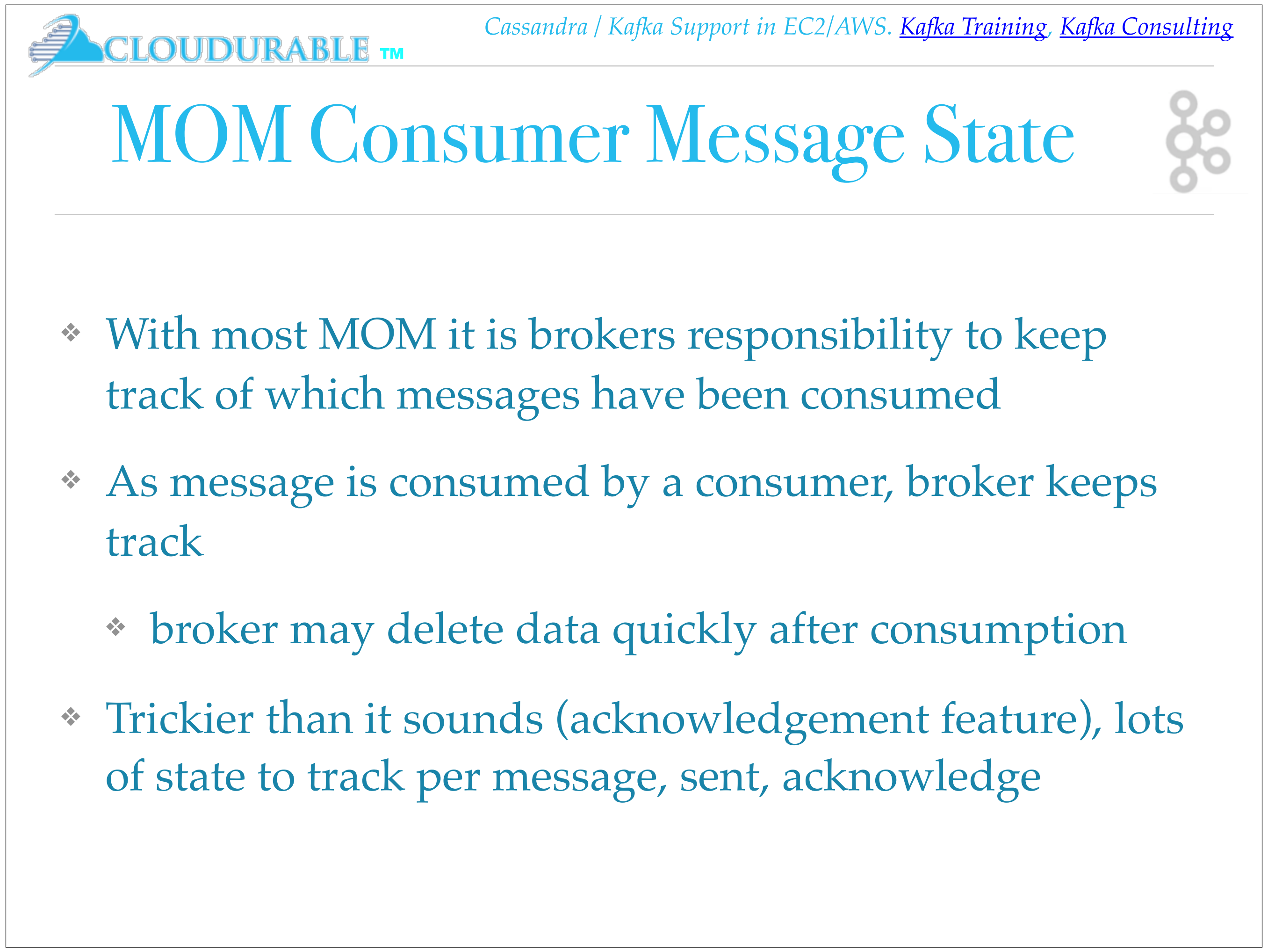
A pull-based system has to pull data and then process it, and there is always a pause between the pull and getting the data.

Pull vs. Push/Streams: Push

- Push based push data to consumers (scribe, flume, reactive streams, RxJava, Akka)
  - push-based have problems dealing with slow or dead consumers
  - push system consumer can get overwhelmed
  - push based systems use back-off protocol (back pressure)
    - consumer can indicate it is overwhelmed, (http://www.reactive-streams.org/)
- Push-based streaming system can
  - send a request immediately or accumulate request and send in batches
- Push-based systems are always pushing data or streaming data
  - Advantage: Consumer can accumulate data while it is processing data already sent
  - Disadvantage: If consumer dies, how does broker know and when does data get resent to another consumer (harder to manage message acks; more complex)

Push based push data to consumers (scribe, flume, reactive streams, RxJava, Akka). Push-based or streaming systems have problems dealing with slow or dead consumers. It is possible for a push system consumer to get overwhelmed when its rate of consumption falls below the rate of production. Some push-based systems use a back-off protocol based on back pressure that allows a consumer to indicate it is overwhelmed see reactive streams (http://www.reactive-streams.org/). This problem of not flooding a consumer and consumer recovery, are tricky when trying to track message acknowledgments.

Push-based or streaming systems can send a request immediately or accumulate requests and send in batches (or a combination based on back pressure). Push-based systems are always pushing data. The consumer can accumulate messages while it is processing data already sent which is an advantage to reduce the latency of message processing. However, if the consumer died when it was behind processing, how does the broker know where the consumer was and when does data get sent again to another consumer. This problem is not an easy problem to solve. Kafka gets around these complexities by using a pull-based system.

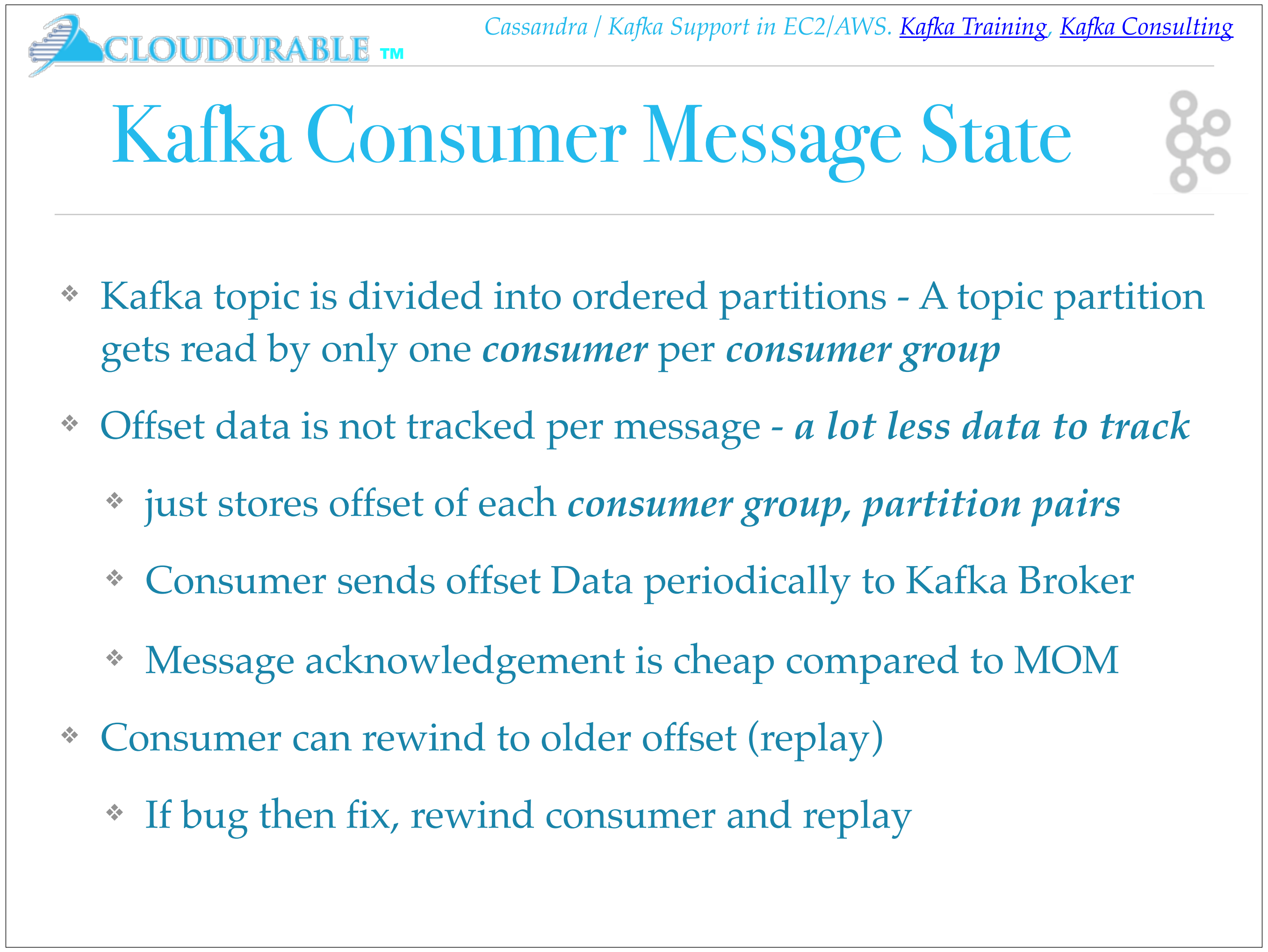


MOM Consumer Message State

- With most MOM it is brokers responsibility to keep track of which messages have been consumed
- As message is consumed by a consumer, broker keeps track
    - broker may delete data quickly after consumption
- Trickier than it sounds (acknowledgement feature), lots of state to track per message, sent, acknowledge

With most MOM it is the broker's responsibility to keep track of which messages gets marked consumed. Message tracking is not an easy task. As consumer consumes messages, the broker keeps track of the state.

The goal in most MOM systems is for the broker to delete data quickly after consumption. Remember most MOMs were written when disks were a lot smaller, less capable, and more expensive. This message tracking is trickier than it sounds (acknowledgment feature), as brokers must maintain lots of states to track per message, sent, acknowledge, and know when to delete or resend the message.



Kafka Consumer Message State

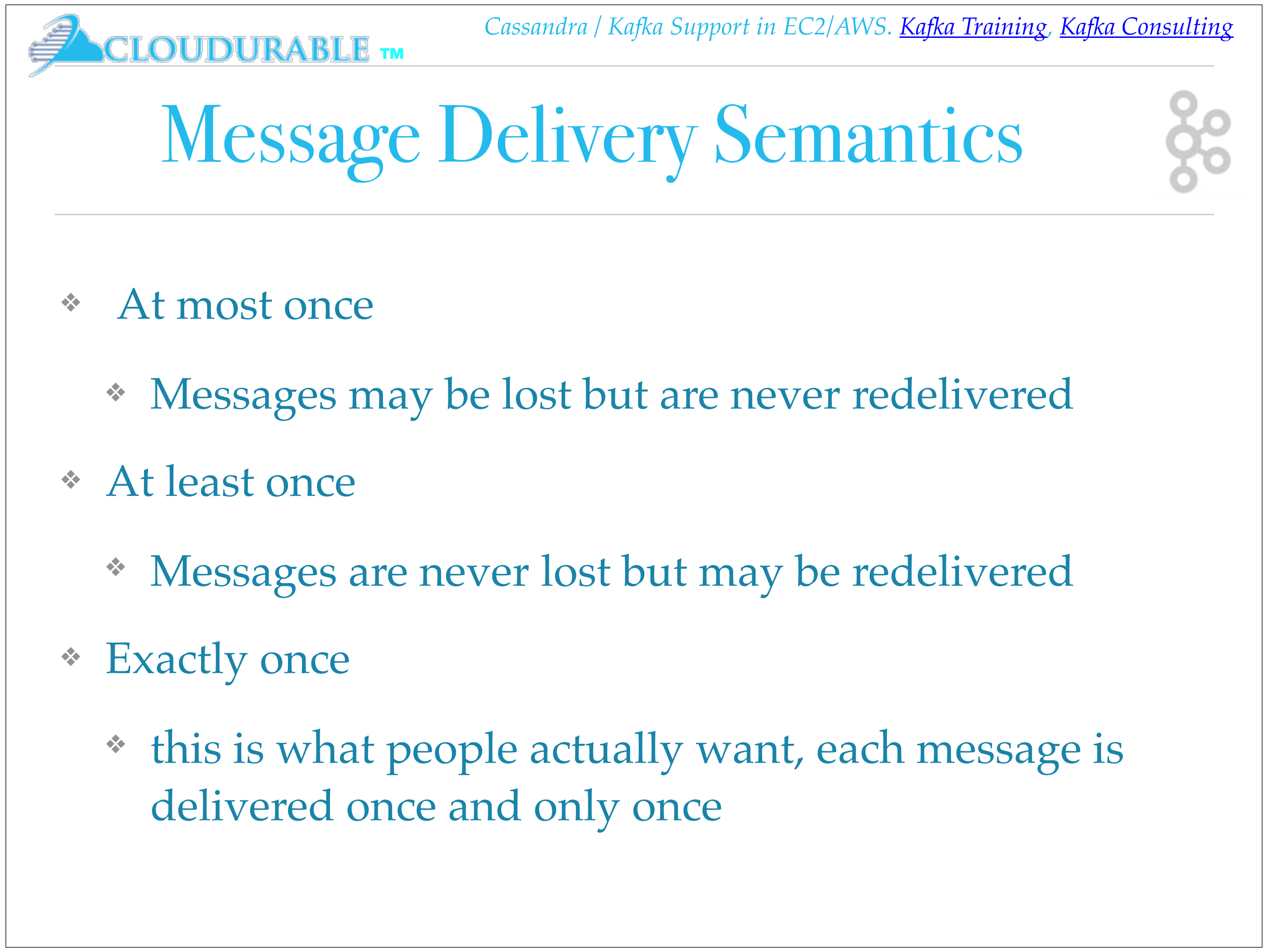
- Kafka topic is divided into ordered partitions - A topic partition gets read by only one ***consumer*** per ***consumer group***
- Offset data is not tracked per message - ***a lot less data to track***
  - just stores offset of each ***consumer group, partition pairs***
  - Consumer sends offset Data periodically to Kafka Broker
  - Message acknowledgement is cheap compared to MOM
- Consumer can rewind to older offset (replay)
  - If bug then fix, rewind consumer and replay

Kafka Consumer state
Remember that Kafka topics get divided into ordered partitions. Each message has an offset in this ordered partition. Each topic partition is consumed by exactly one consumer per consumer group at a time.

This partition layout means, the Broker tracks the offset data not tracked per message like MOM, but only needs the offset of each consumer group, partition offset pair stored. This offset tracking equates to a lot fewer data to track.

The consumer sends location data periodically (consumer group, partition offset pair) to the Kafka broker, and the broker stores this offset data into an offset topic.

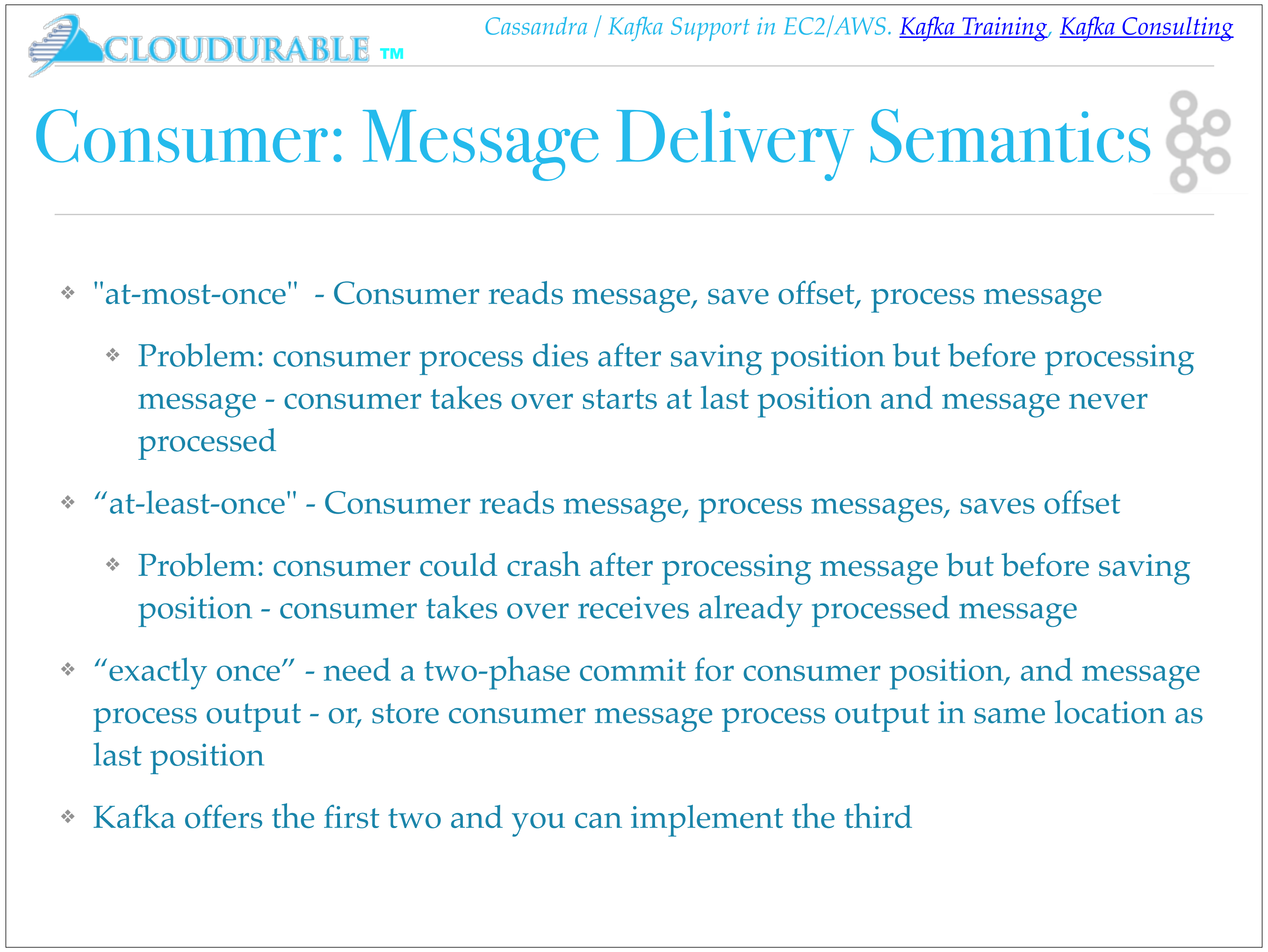
The offset style message acknowledgment is much cheaper compared to MOM. Also, consumers are more flexible and can rewind to an earlier offset (replay). If there was a bug, then fix the bug, rewind consumer and replay the topic. This rewind feature is a killer feature of Kafka as Kafka can hold topic log data for a very long time.



Message Delivery Semantics

- At most once
  - Messages may be lost but are never redelivered
- At least once
  - Messages are never lost but may be redelivered
- Exactly once
  - this is what people actually want, each message is delivered once and only once

There are three message delivery semantics: at most once, at least once and exactly once. At most once is messages may be lost but are never redelivered. At least once is messages are never lost but may be redelivered. Exactly once is each message is delivered once and only once. Exactly once is preferred but more expensive, and requires more bookkeeping for the producer and consumer.



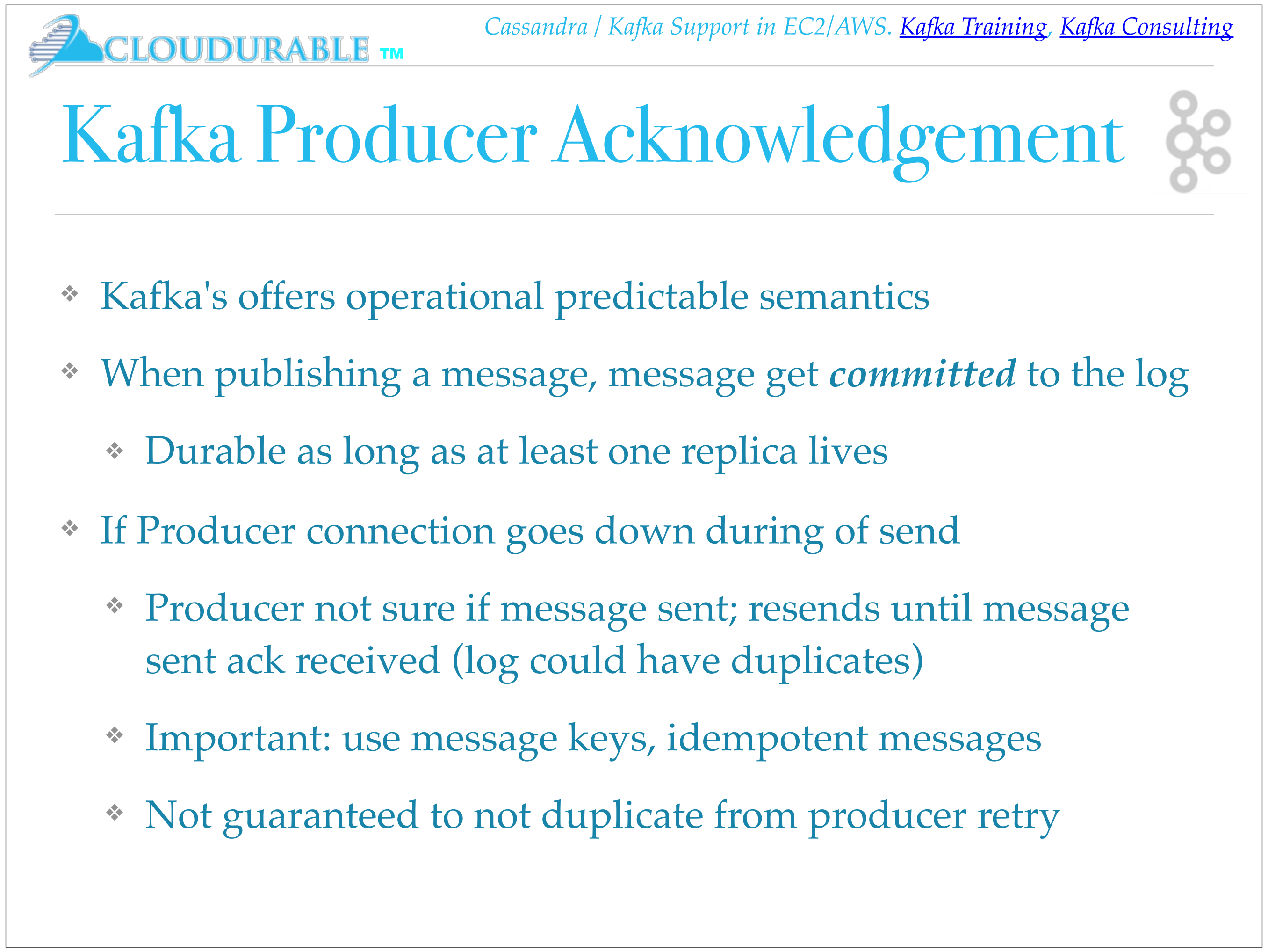
Consumer: Message Delivery Semantics

- "at-most-once" - Consumer reads message, save offset, process message
  - Problem: consumer process dies after saving position but before processing message - consumer takes over starts at last position and message never processed
- “at-least-once" - Consumer reads message, process messages, saves offset
  - Problem: consumer could crash after processing message but before saving position - consumer takes over receives already processed message
- “exactly once” - need a two-phase commit for consumer position, and message process output - or, store consumer message process output in same location as last position
- Kafka offers the first two and you can implement the third

Recall that all replicas have exactly the same log partitions with the same offsets and the consumer groups maintain its position in the log per topic partition. To implement "at-most-once" consumer reads a message, then saves its offset in the partition by sending it to the broker, and finally process the message. The issue with "at-most-once" is a consumer could die after saving its position but before processing the message. Then the consumer that takes over or gets restarted would leave off at the last position and message in question is never processed.

To implement "at-least-once" the consumer reads a message, process messages, and finally saves offset to the broker. The issue with "at-least-once" is a consumer could crash after processing a message but before saving last offset position. Then if the consumer is restarted or another consumer takes over, the consumer could receive the message that was already processed. The "at-least-once" is the most common set up for messaging, and it is your responsibility to make the messages idempotent, which means getting the same message twice will not cause a problem (two debits).

To implement "exactly once" on the consumer side, the consumer would need a two-phase commit between storage for the consumer position, and storage of the consumer's message process output. Or, the consumer could store the message process output in the same location as the last offset.
Kafka offers the first two, and you can implement the third.



Kafka Producer Acknowledgement

- Kafka's offers operational predictable semantics
- When publishing a message, message get ***committed*** to the log
  - Durable as long as at least one replica lives
- If Producer connection goes down during of send
  - Producer not sure if message sent; resends until message sent ack received (log could have duplicates)
  - Important: use message keys, idempotent messages
  - Not guaranteed to not duplicate from producer retry

Kafka Producer Durability and Acknowledgement offers operational predictability semantics for durability. When publishing a message, a message gets "committed" to the log which means all ISRs accepted the message. This commit strategy works out well for durability as long as at least one replica lives.

The producer connection could go down in middle of send, and producer may not be sure if a message it sent went through, and then the producer resends the message. This resend-logic is why it is important to use message keys and use idempotent messages (duplicates ok).
Kafka did not make guarantees of messages not getting duplicated from producer retrying until recently (June 2017). The producer can resend a message until it receives confirmation, i.e., acknowledgment received. The producer resending the message without knowing if the other message it sent made it or not, negates "exactly once" and "at-most-once" message delivery semantics.

Producer Durability Levels

- Producer can specify durability level
- Producer can wait on a message being committed. Waiting for commit ensures all replicas have a copy of the message
- Producer can send with no acknowledgments (0)
- Producer can send with acknowledgment from partition leader (1)
- Producer can send and wait on acknowledgments from all replicas (-1) (default)
- As of June 2017: producer can ensure a message or group of messages was sent "exactly once"

The producer can specify durability level. The producer can wait on a message being committed. Waiting for commit ensures all replicas have a copy of the message.

The producer can send with no acknowledgments (0). The producer can send with just get one acknowledgment from the partition leader (1). The producer can send and wait on acknowledgments from all replicas (-1), which is the default.

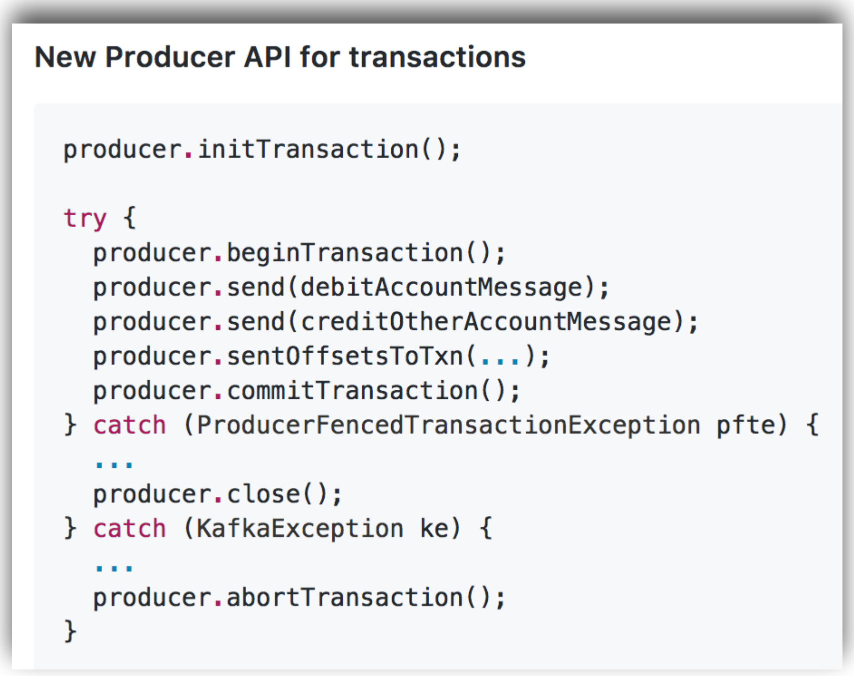
As of June 2017: the producer can ensure a message or group of messages was sent "exactly once".

Improved Producer (coming soon)

- New feature:
    - exactly once delivery from producer, atomic write across partitions, (coming soon),
- producer sends sequence id, broker keeps track if producer already sent this sequence
- if producer tries to send it again, it gets ack for duplicate, but nothing is save to log
- NO API changed

Kafka has improved Producer durability as of the June 2017 release. Kafka now supports "exactly once" delivery from producer (https://www.slideshare.net/apurva2/introducing-exactly-once-semantics-to-apache-kafka), performance improvements and atomic write across partitions.
They achieve this by the producer sending a sequence id, the broker keeps track if producer already sent this sequence, if producer tries to send it again, it gets an ack for duplicate message, but nothing is saved to log. This improvement requires no API change.

Coming Soon: Kafka Atomic Log Writes



New Producer API for transactions

```
producer.initTransaction();

try {
  producer.beginTransaction();
  producer.send(debitAccountMessage);
  producer.send(creditOtherAccountMessage);
  producer.sentOffsetsToTxn(...);
  producer.commitTransaction();
} catch (ProducerFencedTransactionException pfte) {
  ...
  producer.close();
} catch (KafkaException ke) {
  ...
  producer.abortTransaction();
}
```

Consumer only see committed logs

Marker written to log to signify what has been successful transacted

Transaction coordinator and transaction log maintain state

New producer API for transactions

Kafka added producer atomic log writes to the June 2017 release. This improvement means Kafka producers having atomic write across partitions. The atomic writes mean Kafka consumers can only see committed logs (configurable). Kafka has a coordinator that writes a marker to the topic log to signify what has been successfully transacted. The transaction coordinator and transaction log maintain the state of the atomic writes.

Kafka Replication

- Kafka replicates each topic's partitions across a configurable number of Kafka brokers
- Kafka is replicated by default not a bolt-on feature
- Each topic partition has one leader and zero or more followers
    - leaders and followers are called replicas
    - replication factor = 1 leader + N followers
- Reads and writes always go to leader
- Partition leadership is evenly shared among Kafka brokers
    - logs on followers are in-sync to leader's log - identical copy - sans un-replicated offsets
- Followers pull records in batches records from leader like a regular Kafka consumer

Kafka replicates each topic's partitions across a configurable number of Kafka brokers. Kafka's replication model is by default, not a bolt-on feature like most MOMs as Kafka was meant to work with partitions and multi-nodes from the start. Each topic partition has one leader and zero or more followers.

Leaders and followers are called replicas. A replication factor is the leader node plus all of the followers. Partition leadership is evenly shared among Kafka brokers. Consumers only read from the leader. Producers only write to the leaders.

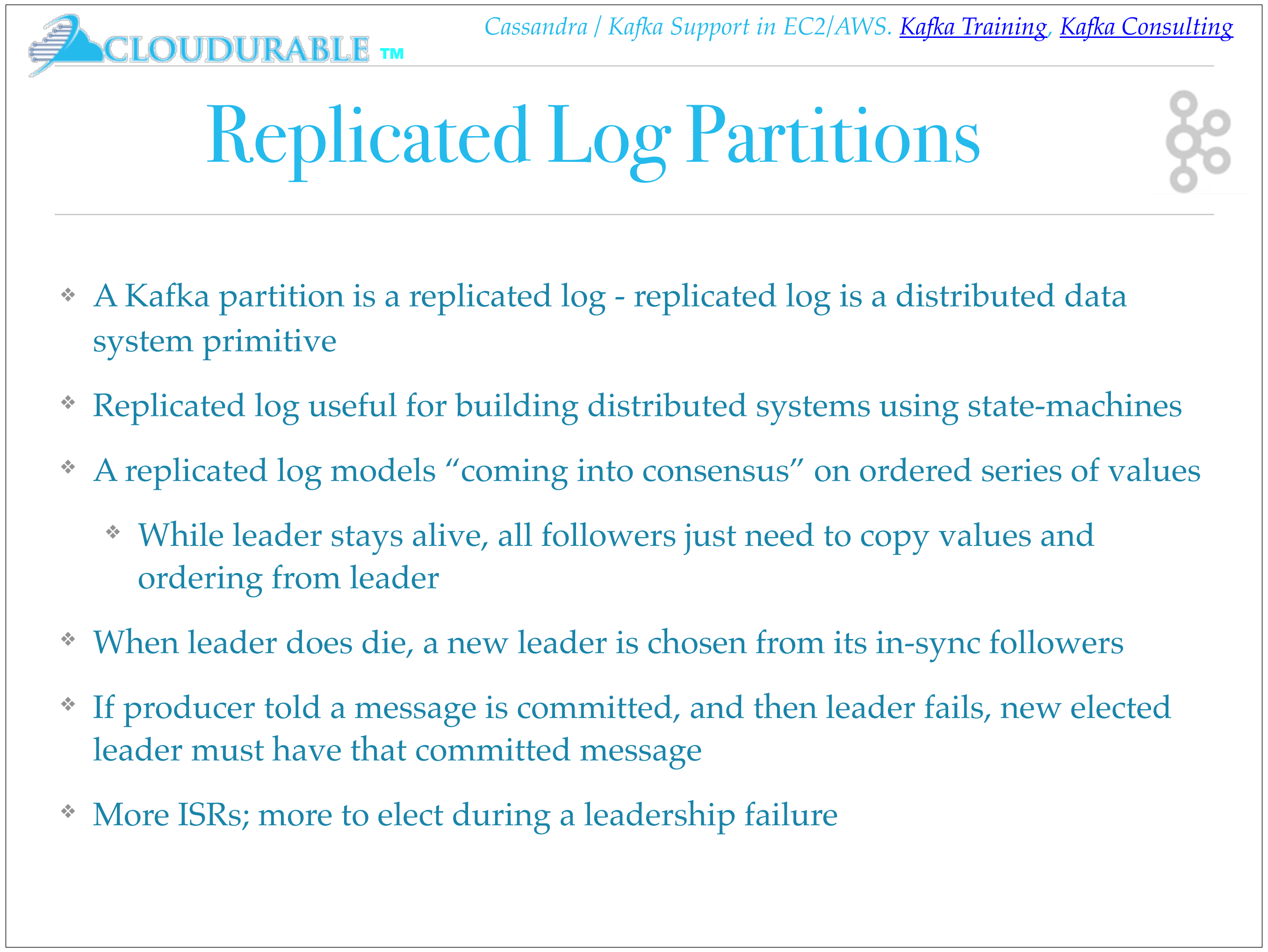
The topic log partitions on followers are in-sync to leader's log, ISRs are an exact copy of the leaders minus the to-be-replicated records that are in-flight. Followers pull records in batches from their leader like a regular Kafka consumer.

Kafka Broker Failover

- Kafka keeps track of which Kafka Brokers are alive (in-sync)
    - To be alive Kafka Broker must maintain a ZooKeeper session (heart beat)
    - Followers must replicate writes from leader and not fall "too far" behind
- Each leader keeps track of set of "in sync replicas" aka ISRs
- If ISR/follower dies, falls behind, leader will removes follower from ISR set - falling behind ***replica.lag.time.max.ms > lag***
- Kafka guarantee: committed message not lost, as long as one live ISR - "committed" when written to all ISRs logs
- Consumer only reads committed messages

Kafka keeps track of which Kafka brokers are alive. To be alive, a Kafka Broker must maintain a ZooKeeper session using ZooKeeper's heartbeat mechanism, and must have all of its followers in-sync with the leaders and not fall too far behind. Both the ZooKeeper session and being in-sync is needed for broker liveness which is referred to as being in-sync. An in-sync replica is called an ISR. Each leader keeps track of a set of "in sync replicas".

If ISR/follower dies, falls behind, then the leader will remove the follower from the set of ISRs. Falling behind is when a replica is not in-sync after `replica.lag.time.max.ms` period. A message is considered "committed" when all ISRs have applied the message to their log. Consumers only see committed messages. Kafka guarantee: committed message will not be lost, as long as there is at least one ISR.



Replicated Log Partitions

- A Kafka partition is a replicated log - replicated log is a distributed data system primitive
- Replicated log useful for building distributed systems using state-machines
- A replicated log models “coming into consensus” on ordered series of values
  - While leader stays alive, all followers just need to copy values and ordering from leader
- When leader does die, a new leader is chosen from its in-sync followers
- If producer told a message is committed, and then leader fails, new elected leader must have that committed message
- More ISRs; more to elect during a leadership failure

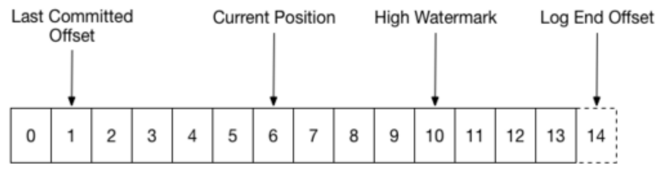
A Kafka partition is a replicated log. A replicated log is a distributed data system primitive. A replicated log is useful for implementing other distributed systems using state machines.
A replicated log models "coming into consensus" on an ordered series of values.

While a leader stays alive, all followers just need to copy values and ordering from their leader. If the leader does die, Kafka chooses a new leader from its followers which are in-sync. If a producer is told a message is committed, and then the leader fails, then the newly elected leader must have that committed message.

The more ISRs you have; the more there are to elect during a leadership failure.

Kafka Consumer Replication Redux

- What can be consumed?
- ***"Log end offset"*** is offset of last record written to log partition and where ***Producers*** write to next
- ***"High watermark"*** is offset of last record successfully replicated to all partitions followers
- ***Consumer*** only reads up to "high watermark". ***Consumer can't read un-replicated data***



"Log end offset" is offset of the last record written to log partition and where Producers writes to next.
"High Watermark" is the offset of the last record that was successfully replicated to all partitions followers.
Consumer only reads up to the "high watermark." Consumers can't read un-replicated data.

Kafka Broker Replication Config

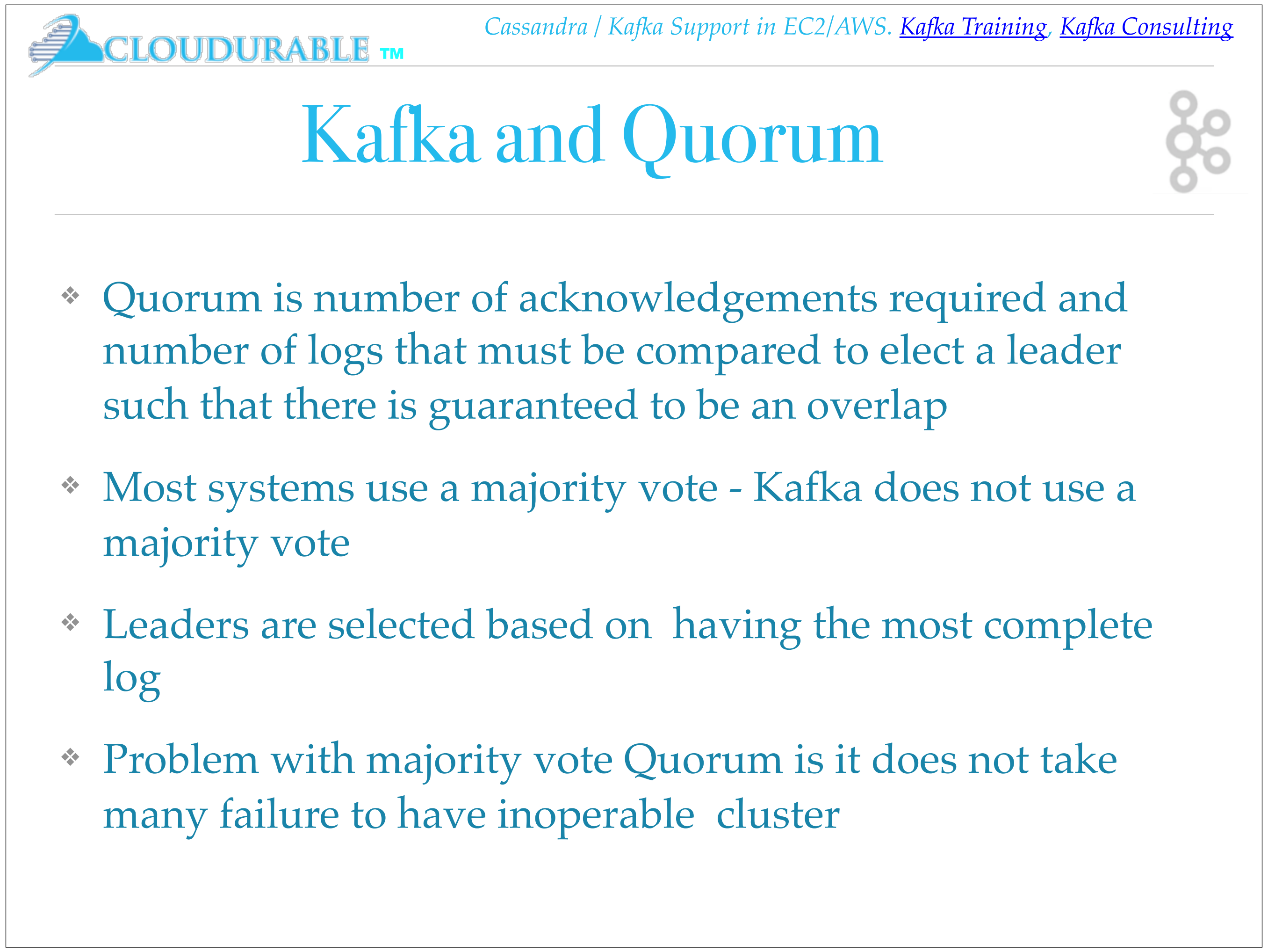
Kafka Broker Config

NAME	DESCRIPTION	TYPE	DEFAULT
auto.leader.rebalance.enable	Enables auto leader balancing.	boolean	TRUE
leader.imbalance.check.interval.seconds	The interval for checking for partition leadership balancing.	long	300
leader.imbalance.per.broker.percentage	Leadership imbalance for each broker. If imbalance is too high then a rebalance is triggered.	int	10
min.insync.replicas	When a producer sets acks to all (or -1), This setting is the minimum replicas count that must acknowledge a write for the write to be considered successful. If not met, then the producer will raise an exception (either NotEnoughReplicas or NotEnoughReplicasAfterAppend).	int	1
num.replica.fetchers	Replica fetcher count. Used to replicate messages from a broker that has a leadership partition. Increase this if followers are falling behind.	int	1

Kafka Replication Broker Config 2

Kafka Broker Config

NAME	DESCRIPTION
replica.high.watermark.checkpoint.interval.ms	The frequency with which the high watermark is saved out to disk used for knowing what consumers can consume. ***Consumer*** only reads up to "high watermark". ***Consumer can't read un-replicated data.***
replica.lag.time.max.ms	Determines which Replicas are in the ISR set and which are not. ISR is is important for acks and quorum.
replica.socket.receive.buffer.bytes	The socket receive buffer for network requests
replica.socket.timeout.ms	The socket timeout for network requests. Its value should be at least replica.fetch.wait.max.ms
unclean.leader.election.enable	What happens if all of the nodes go down? Indicates whether to enable replicas not in the ISR. Replicas that are not in-sync. Set to be elected as leader as a last resort, even though doing so may result in data loss. Availability over Consistency. True is the default.



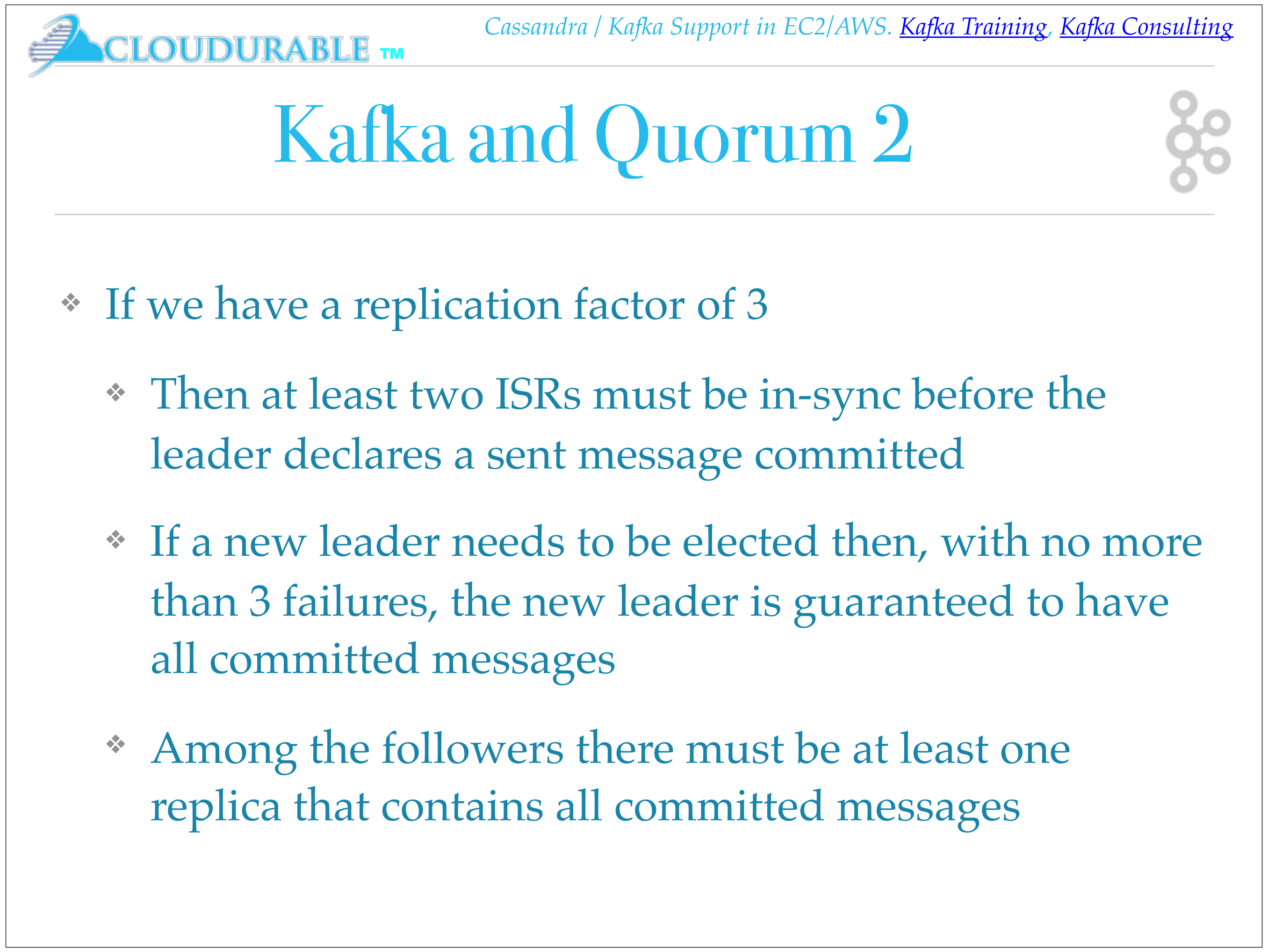
Kafka and Quorum

- Quorum is number of acknowledgements required and number of logs that must be compared to elect a leader such that there is guaranteed to be an overlap
- Most systems use a majority vote - Kafka does not use a majority vote
- Leaders are selected based on having the most complete log
- Problem with majority vote Quorum is it does not take many failure to have inoperable cluster

Quorum is the number of acknowledgments required and the number of logs that must be compared to elect a leader such that there is guaranteed to be an overlap for availability. Most systems use a majority vote, Kafka does not use a simple majority vote to improve availability.

In Kafka, leaders are selected based on having a complete log. Problem with majority vote Quorum is it does not take many failures to have inoperable cluster.

See more at https://kafka.apache.org/documentation/#design_replicatedlog



Kafka and Quorum 2

- If we have a replication factor of 3
    - Then at least two ISRs must be in-sync before the leader declares a sent message committed
    - If a new leader needs to be elected then, with no more than 3 failures, the new leader is guaranteed to have all committed messages
    - Among the followers there must be at least one replica that contains all committed messages

If we have a replication factor of 3, then at least two ISRs must be in-sync before the leader declares a sent message committed.

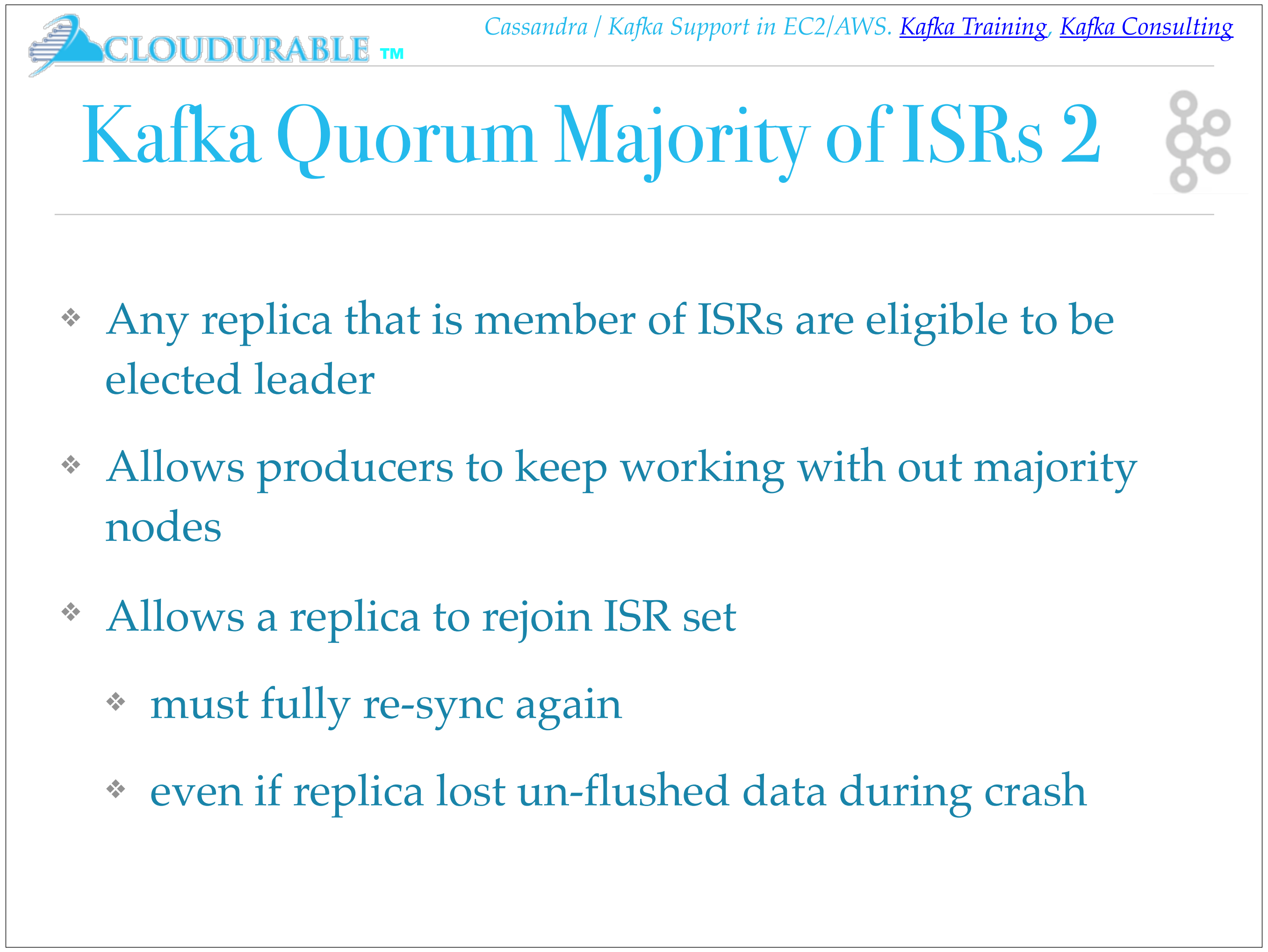
If a new leader needs to be elected then, with no more than 3 failures, the new leader is guaranteed to have all committed messages.

Among the followers there must be at least one replica that contains all committed messages.

Kafka Quorum Majority of ISRs

- Kafka maintains a set of ISRs
- Only this set of ISRs are eligible for leadership election
- Write to partition is not committed until all ISRs ack write
- ISRs persisted to ZooKeeper whenever ISR set changes

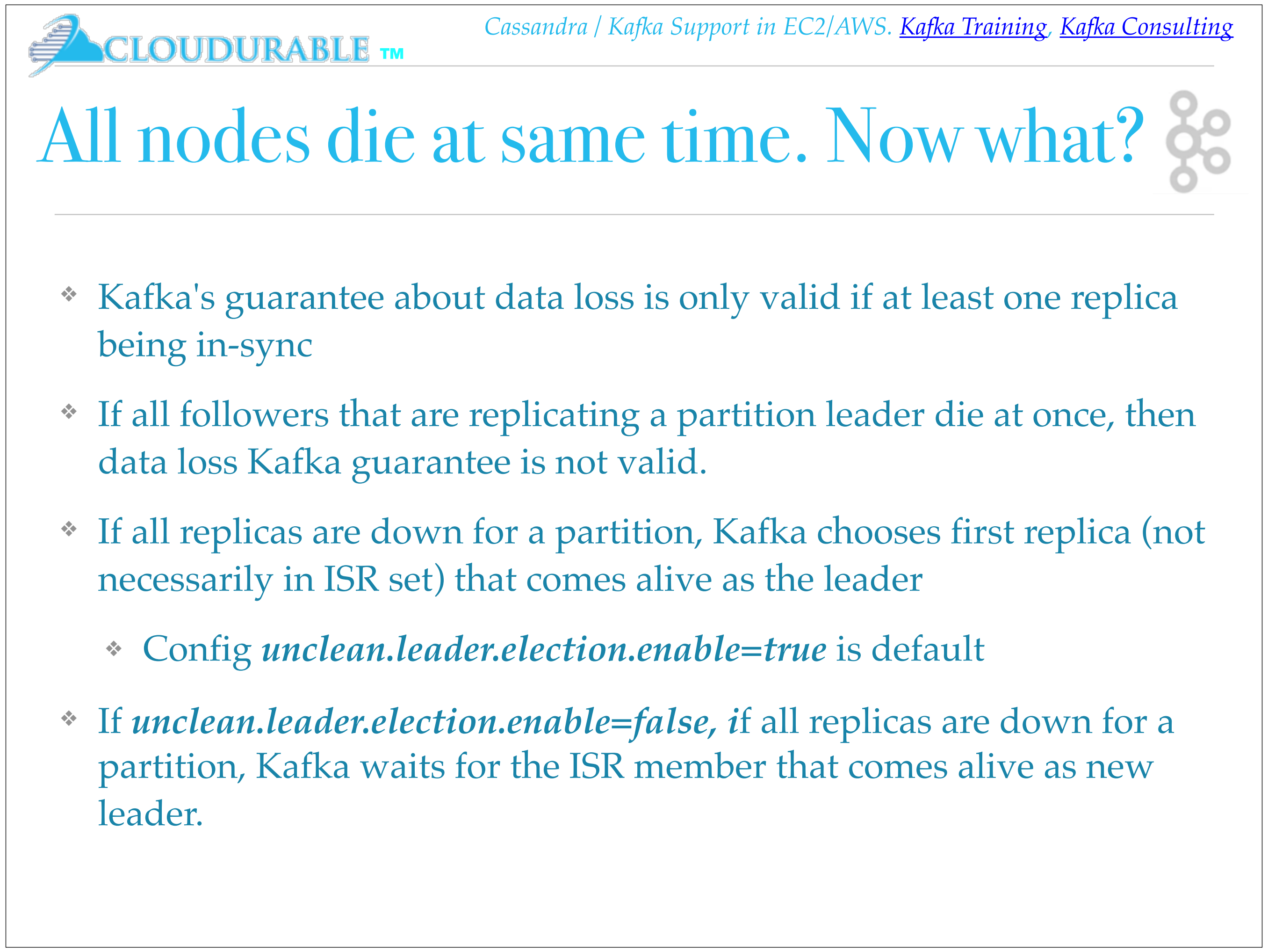
Kafka maintains a set of ISRs per leader. Only members in this set of ISRs are eligible for leadership election. What the producer writes to partition is not committed until all ISRs acknowledge the write. ISRs are persisted to ZooKeeper whenever ISR set changes.



Kafka Quorum Majority of ISRs 2

- Any replica that is member of ISRs are eligible to be elected leader
- Allows producers to keep working with out majority nodes
- Allows a replica to rejoin ISR set
  - must fully re-sync again
  - even if replica lost un-flushed data during crash

Only replicas that are members of ISR set are eligible to be elected leader. This style of ISR quorum allows producers to keep working without the majority of all nodes, but only an ISR majority vote. This style of ISR quorum also allows a replica to rejoin ISR set and have its vote count, but it has to be fully re-synced before joining even if replica lost un-flushed data during its crash.

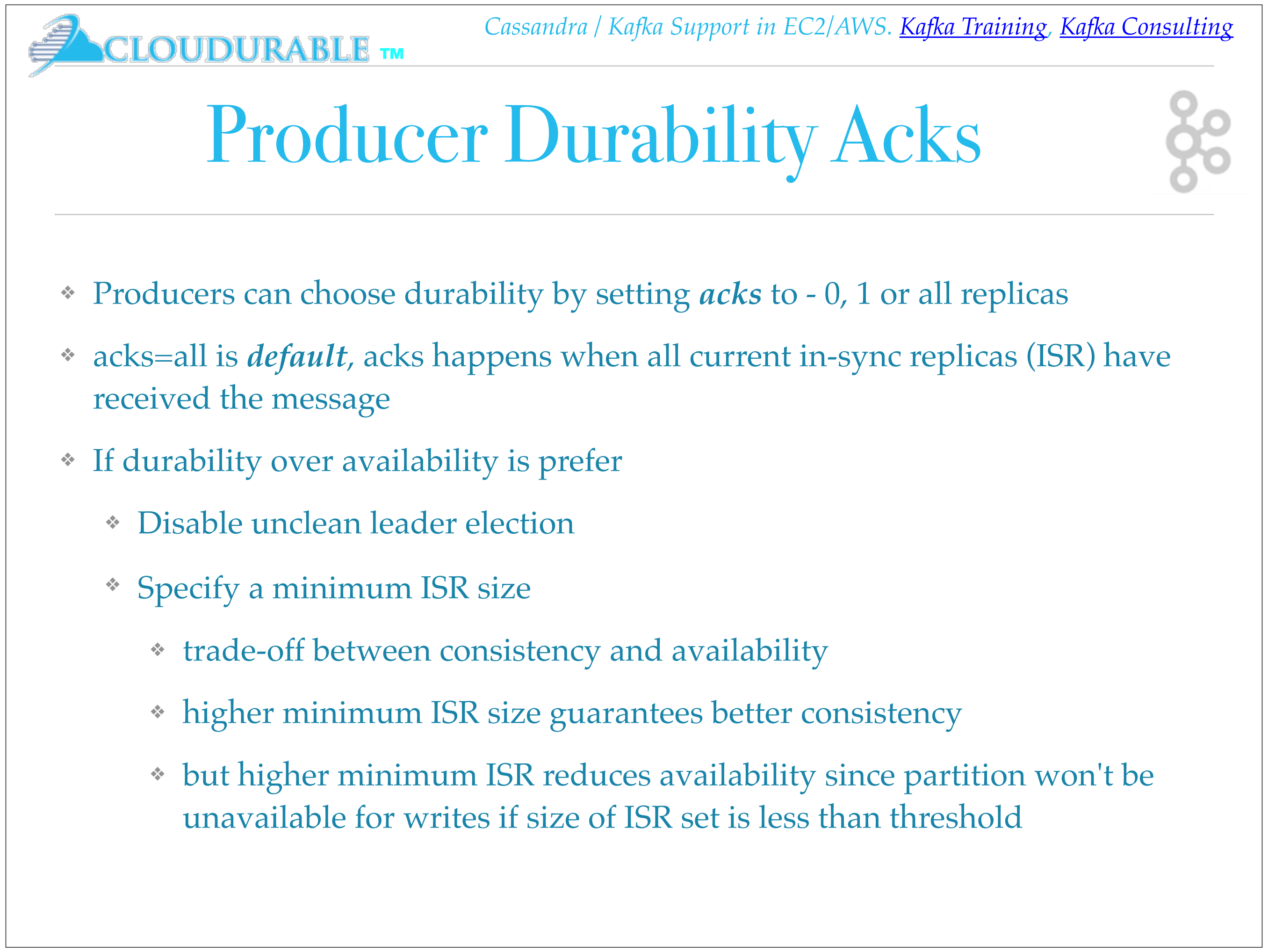


All nodes die at same time. Now what?

- Kafka's guarantee about data loss is only valid if at least one replica being in-sync
- If all followers that are replicating a partition leader die at once, then data loss Kafka guarantee is not valid.
- If all replicas are down for a partition, Kafka chooses first replica (not necessarily in ISR set) that comes alive as the leader
  - Config ***unclean.leader.election.enable=true*** is default
- If ***unclean.leader.election.enable=false, i***f all replicas are down for a partition, Kafka waits for the ISR member that comes alive as new leader.

Kafka's guarantee about data loss is only valid if at least one replica is in-sync. If all followers that are replicating a partition leader die at once, then data loss Kafka guarantee is not valid. If all replicas are down for a partition, Kafka, by default, chooses first replica (not necessarily in ISR set) that comes alive as the leader (config unclean.leader.election.enable=true is default). This choice favors availability to consistency.

If consistency is more important than availability for your use case, then you can set config unclean.leader.election.enable=false then if all replicas are down for a partition, Kafka waits for the first ISR member (not first replica) that comes alive to elect a new leader.

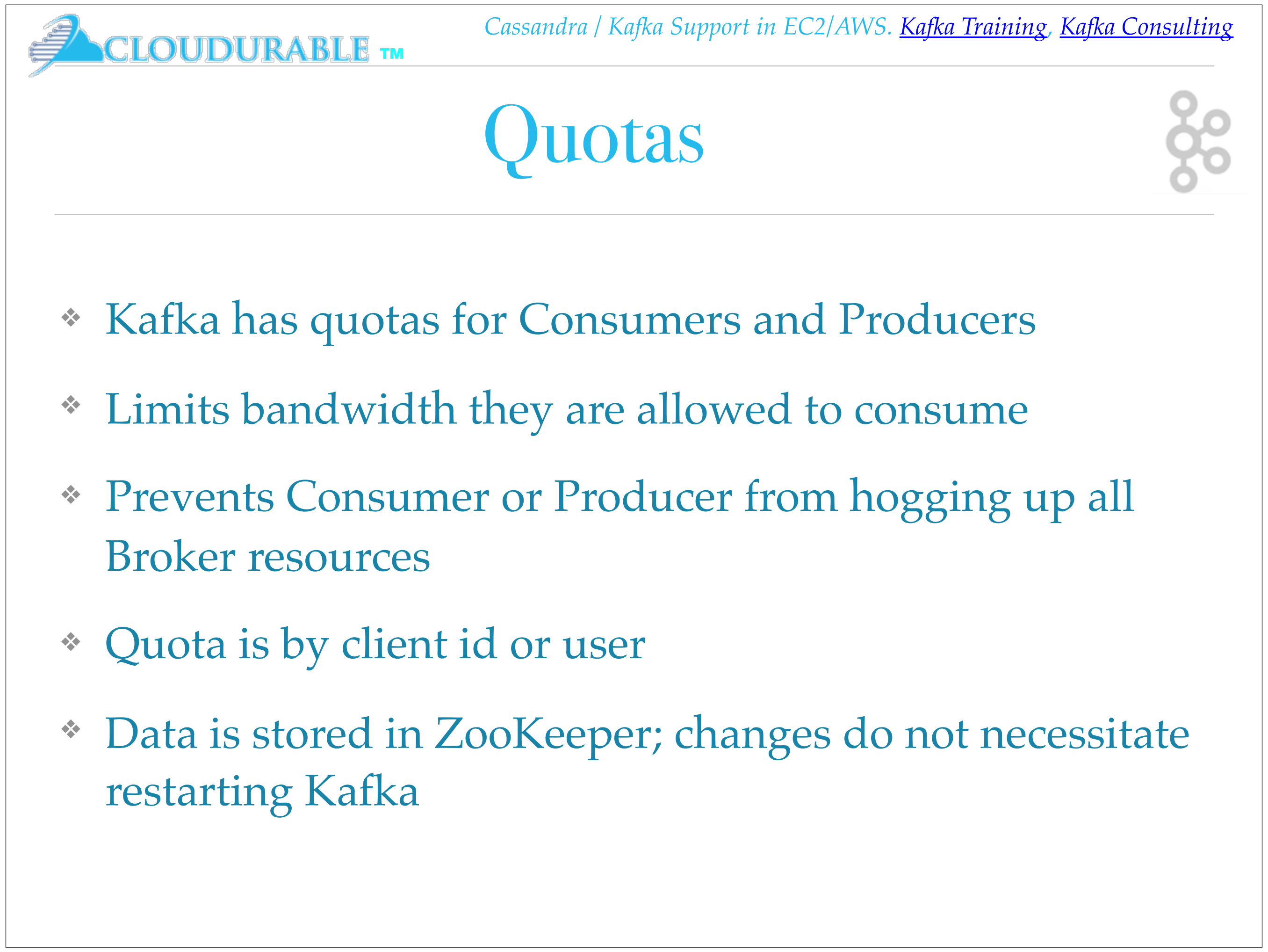


Producer Durability Acks

- Producers can choose durability by setting ***acks*** to - 0, 1 or all replicas
- acks=all is ***default***, acks happens when all current in-sync replicas (ISR) have received the message
- If durability over availability is prefer
  - Disable unclean leader election
  - Specify a minimum ISR size
    - trade-off between consistency and availability
    - higher minimum ISR size guarantees better consistency
    - but higher minimum ISR reduces availability since partition won't be unavailable for writes if size of ISR set is less than threshold

Producers can choose durability by setting acks to - none (0), the leader only (1) or all replicas (-1). The acks=all is the default. With all, the acks happen when all current in-sync replicas (ISRs) have received the message.

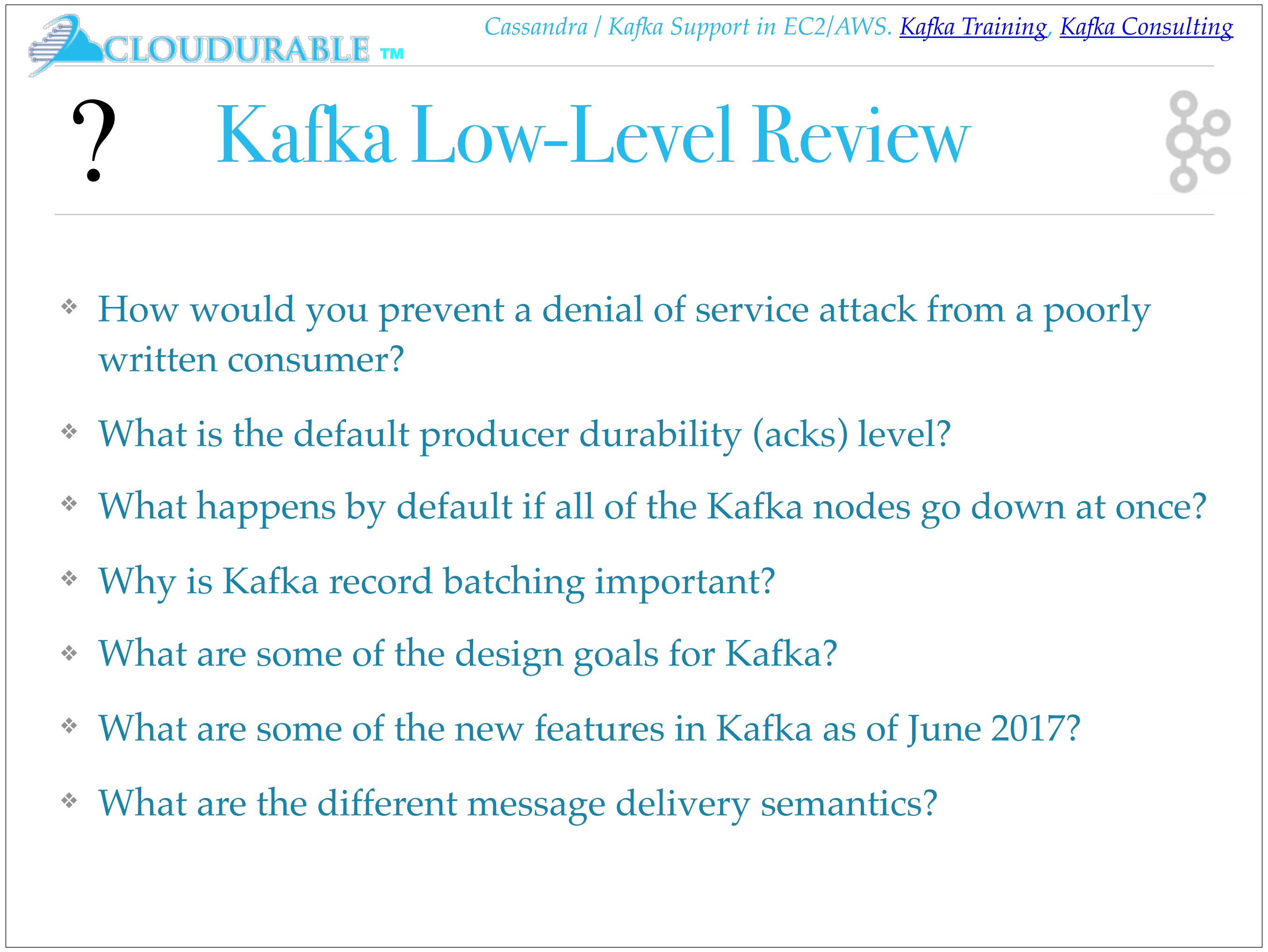
You can make the trade-off between consistency and availability. If durability over availability is preferred, then disable unclean leader election and specify a minimum ISR size. The higher the minimum ISR size, the better the guarantee is for consistency. But the higher minimum ISR, the more you reduces availability since partition won't be unavailable for writes if the size of ISR set is less than the minimum threshold.



Quotas

- Kafka has quotas for Consumers and Producers
- Limits bandwidth they are allowed to consume
- Prevents Consumer or Producer from hogging up all Broker resources
- Quota is by client id or user
- Data is stored in ZooKeeper; changes do not necessitate restarting Kafka

Kafka has quotas for consumers and producers to limits bandwidth they are allowed to consume.
These quotas prevent consumers or producers from hogging up all the Kafka broker resources.
The quota is by client id or user. The quota data is stored in ZooKeeper, so changes do not necessitate restarting Kafka brokers.
https://kafka.apache.org/documentation/#design_quotasconfig



? Kafka Low-Level Review

- How would you prevent a denial of service attack from a poorly written consumer?
- What is the default producer durability (acks) level?
- What happens by default if all of the Kafka nodes go down at once?
- Why is Kafka record batching important?
- What are some of the design goals for Kafka?
- What are some of the new features in Kafka as of June 2017?
- What are the different message delivery semantics?

How would you prevent a denial of service attack from a poorly written consumer?
Use Quotas to limit the consumer's bandwidth.

What is the default producer durability (acks) level?
All. Which means all ISRs have to write the message to their log partition.

What happens by default if all of the Kafka nodes go down at once?
Kafka chooses the first replica (not necessarily in ISR set) that comes alive as the leader as `unclean.leader.election.enable=true` is default to support availability.

Why is Kafka record batching important? Optimized IO throughput over the wire as well as to the disk. It also improves compression efficiency by compressing an entire batch.

What are some of the design goals for Kafka? To be a high-throughput, scalable streaming data platform for real-time analytics of high-volume event streams like log aggregation, user activity, etc.

What are some of the new features in Kafka as of June 2017? Producer atomic writes, performance improvements and producer not sending duplicate messages.

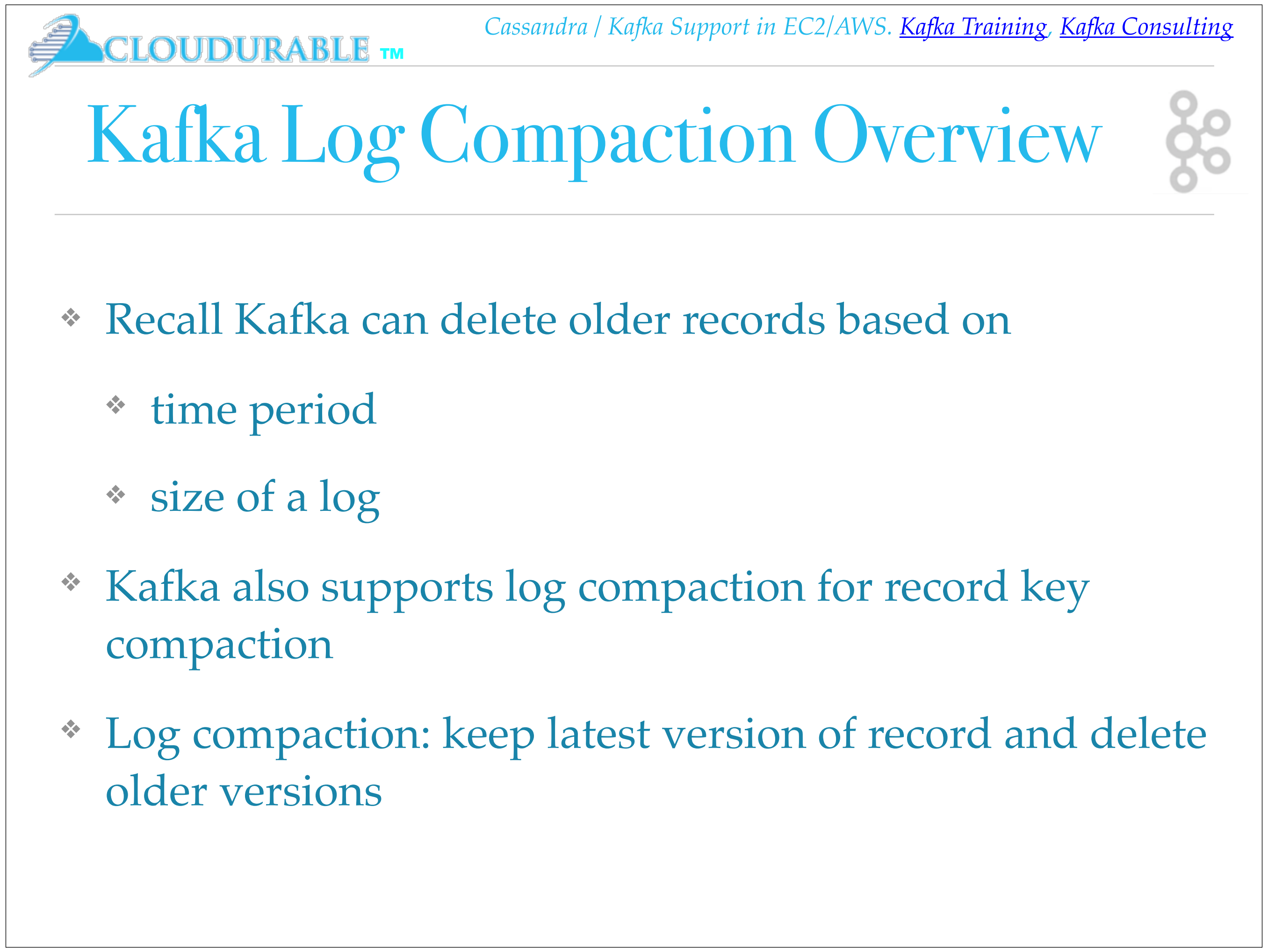
What is the different message delivery semantics? There are three message delivery semantics: at most once, at least once and exactly once.

Kafka Log Compaction Architecture

Kafka Log Compaction

Design discussion of Kafka Log Compaction

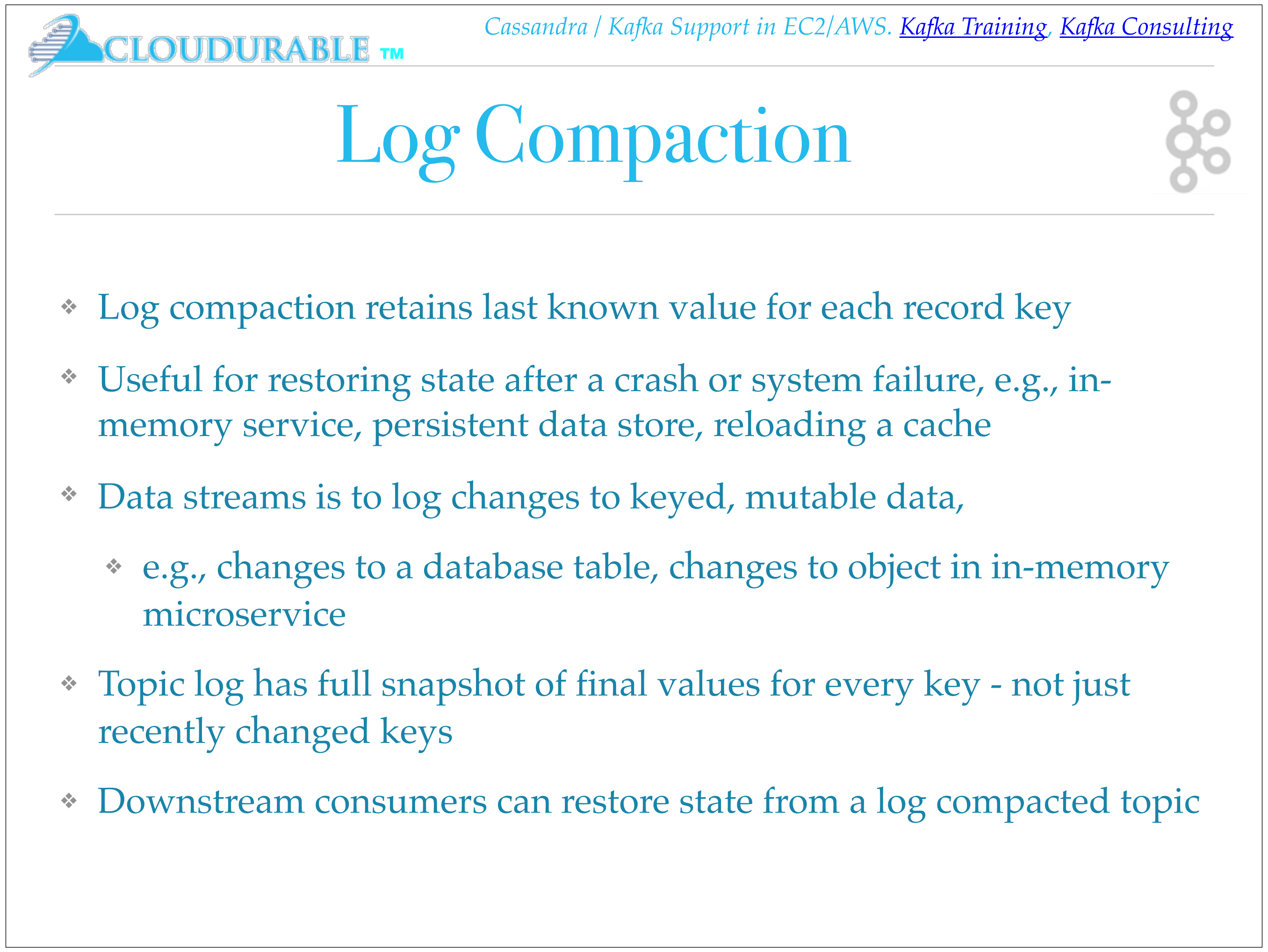
Kafka Architecture: Log Compaction



Kafka Log Compaction Overview

- Recall Kafka can delete older records based on
  - time period
  - size of a log
- Kafka also supports log compaction for record key compaction
- Log compaction: keep latest version of record and delete older versions

Recall that Kafka can delete older records based on time or size of a log. Kafka also supports log compaction for record key compaction. Log compaction means that Kafka will keep the latest version of a record and delete the older versions during a log compaction.



Log Compaction

- Log compaction retains last known value for each record key
- Useful for restoring state after a crash or system failure, e.g., in-memory service, persistent data store, reloading a cache
- Data streams is to log changes to keyed, mutable data,
    - e.g., changes to a database table, changes to object in in-memory microservice
- Topic log has full snapshot of final values for every key - not just recently changed keys
- Downstream consumers can restore state from a log compacted topic

Log compaction retains at least last known value for each record key for a single topic partition. Compacted logs are useful for restoring state after a crash or system failure. They are useful for in-memory services, persistent data stores, reloading a cache, etc. An important use case of data streams is to log changes to keyed, mutable data changes to a database table or changes to object in in-memory microservice.

Log compaction is a granular retention mechanism that retains the last update for each key. A log compacted topic log contains a full snapshot of final record values for every record key not just the recently changed keys. Log compaction allows downstream consumers to restore their state from a log compacted topic.

Log Compaction Structure

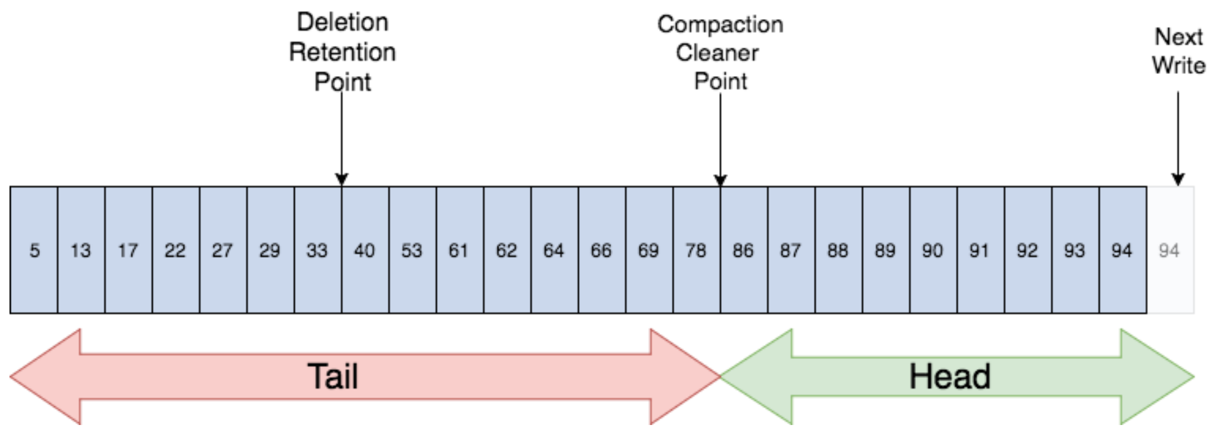
- Log has head and tail
- Head of compacted log identical to a traditional Kafka log
- New records get appended to the head
- Log compaction works at tail of the log
- Tail gets compacted
- Records in tail of log retain their original offset when written after compaction

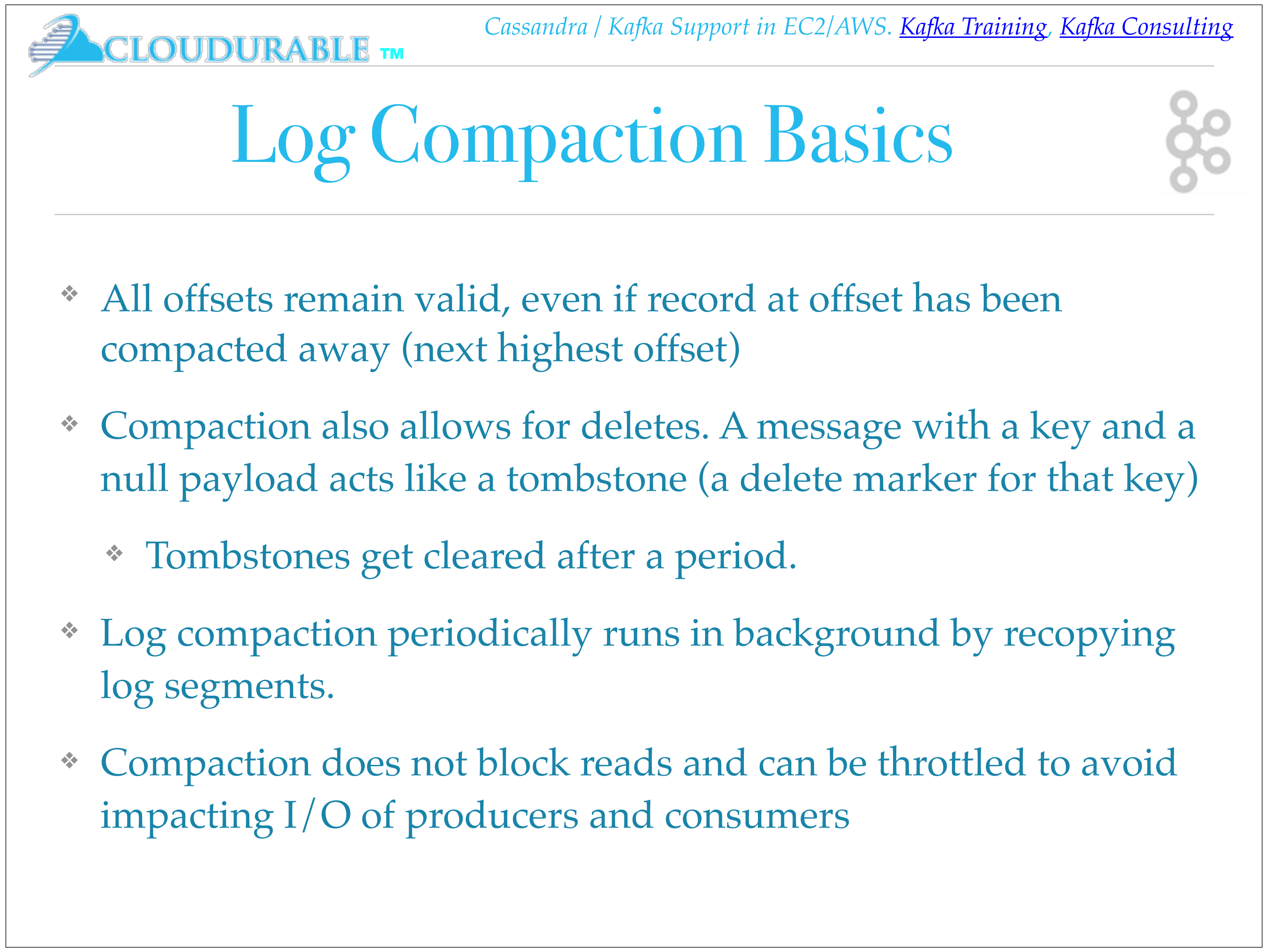
With a compacted log, the log has head and tail. The head of the compacted log is identical to a traditional Kafka log. New records get appended to the end of the head.

All log compaction works at the tail of the log. Only the tail gets compacted. Records in the tail of log retain their original offset when written after being rewritten with compaction cleanup.

https://kafka.apache.org/documentation/#design_compactionbasics

Compaction Tail/Head





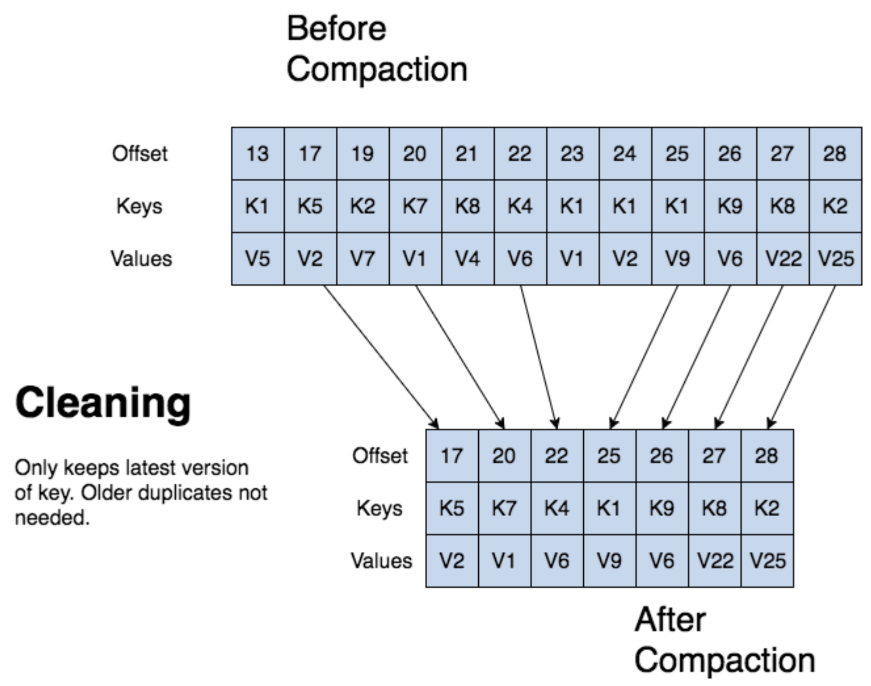
Log Compaction Basics

- All offsets remain valid, even if record at offset has been compacted away (next highest offset)
- Compaction also allows for deletes. A message with a key and a null payload acts like a tombstone (a delete marker for that key)
  - Tombstones get cleared after a period.
- Log compaction periodically runs in background by recopying log segments.
- Compaction does not block reads and can be throttled to avoid impacting I/O of producers and consumers

All compacted log offsets remain valid, even if record at offset has been compacted away as a consumer will get the next highest offset.

Log compaction also allows for deletes. A message with a key and a null payload acts like a tombstone, a delete marker for that key. Tombstones get cleared after a period. Log compaction periodically runs in the background by recopying log segments.
Compaction does not block reads and can be throttled to avoid impacting I/O of producers and consumers.

Log Compaction Cleaning



Before Compaction

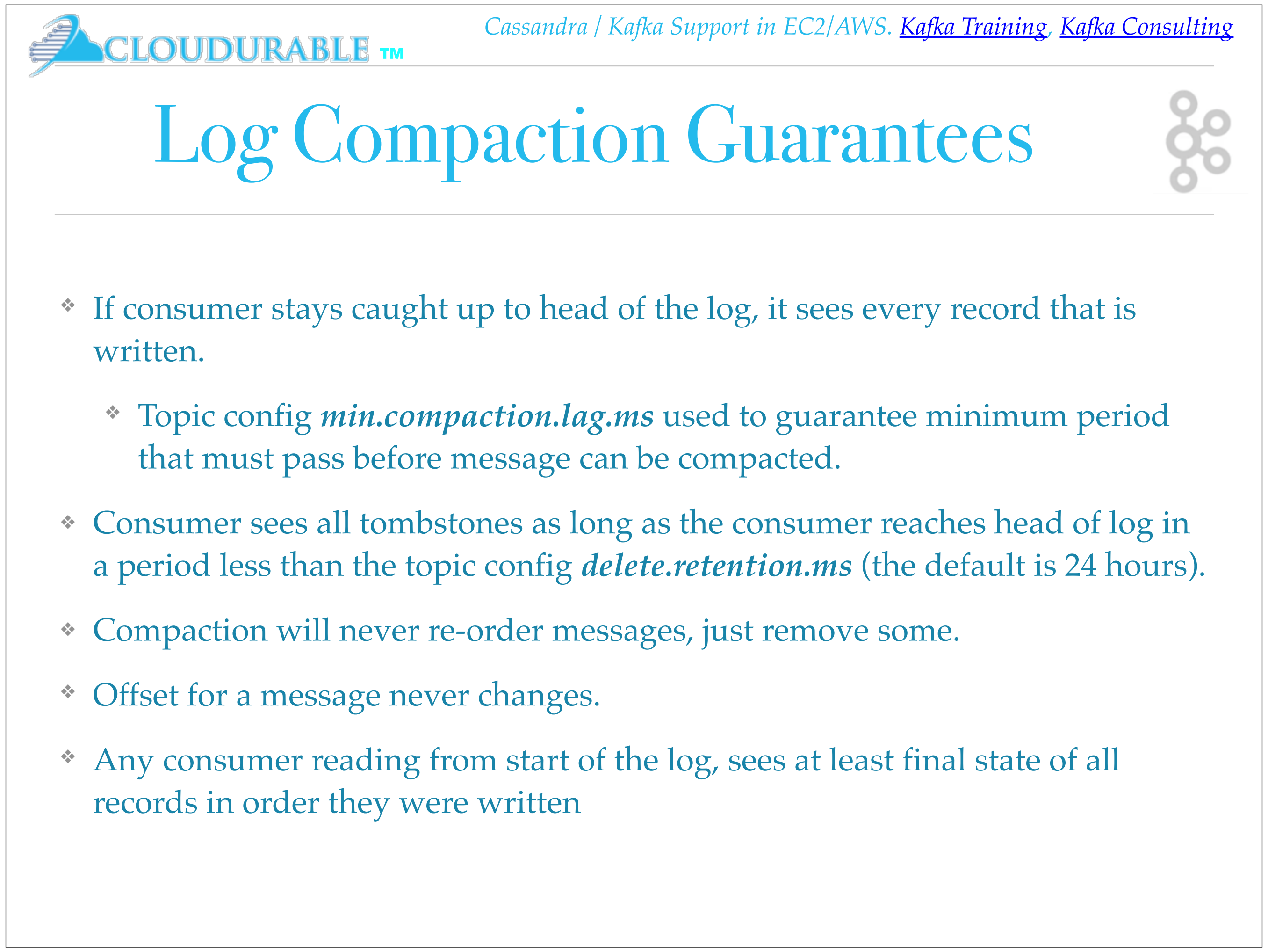
Offset	13	17	19	20	21	22	23	24	25	26	27	28
Keys	K1	K5	K2	K7	K8	K4	K1	K1	K1	K9	K8	K2
Values	V5	V2	V7	V1	V4	V6	V1	V2	V9	V6	V22	V25

Cleaning

Only keeps latest version of key. Older duplicates not needed.

Offset	17	20	22	25	26	27	28
Keys	K5	K7	K4	K1	K9	K8	K2
Values	V2	V1	V6	V9	V6	V22	V25

After Compaction



Log Compaction Guarantees

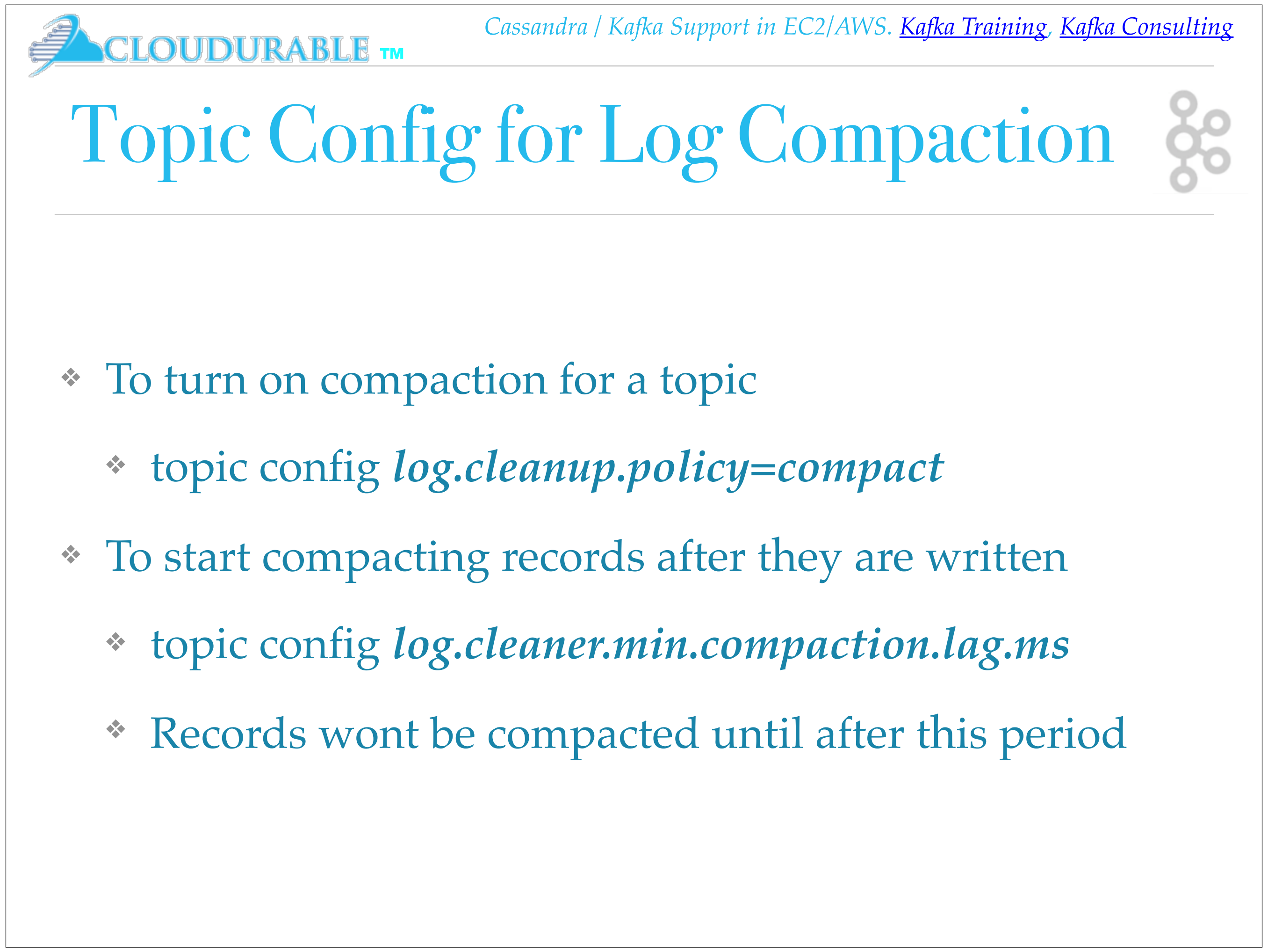
- If consumer stays caught up to head of the log, it sees every record that is written.
    - Topic config ***min.compaction.lag.ms*** used to guarantee minimum period that must pass before message can be compacted.
- Consumer sees all tombstones as long as the consumer reaches head of log in a period less than the topic config ***delete.retention.ms*** (the default is 24 hours).
- Compaction will never re-order messages, just remove some.
- Offset for a message never changes.
- Any consumer reading from start of the log, sees at least final state of all records in order they were written

If consumer stays caught up to head of the log, it sees every record that is written. Topic config min.compaction.lag.ms gets used to guarantee a minimum period that must pass before a message can be compacted. The consumer sees all tombstones as long as the consumer reaches head of a log in a period less than the topic config delete.retention.ms (the default is 24 hours).
Log compaction will never re-order messages, just remove some. Partition offset for a message never changes. Any consumer reading from the start of the log sees at least final state of all records in the order they were written.

Log Cleaner

- Log cleaner does log compaction.
    - Has a pool of background compaction threads that recopy log segments, removing records whose key appears in head of log
- Each compaction thread works as follows:
    - Chooses topic log that has highest ratio: log head to log tail
    - Recopies log from start to end removes records whose keys occur later
- As log partition segments cleaned, they get swapped into log partition
    - Additional disk space required: only one log partition segment
    - not whole partition

Recall that a topic has a log. A topic log is broken up into partitions and partitions are divided into segments which contain records which have keys and values. The Log cleaner does log compaction. The Log cleaner has a pool of background compaction threads. These threads recopy log segment files, removing older records whose key reappears recently in the log. Each compaction thread chooses topic log that has the highest ratio of log head to log tail. Then the compaction thread recopies the log from start to end removing records whose keys occur later in the log. As the log cleaner cleans log partition segments, the segments get swapped into the log partition immediately replacing the older segments. This way compaction does not require double the space of the entire partition as additional disk space required is just one additional log partition segment - divide and conquer.



Topic Config for Log Compaction

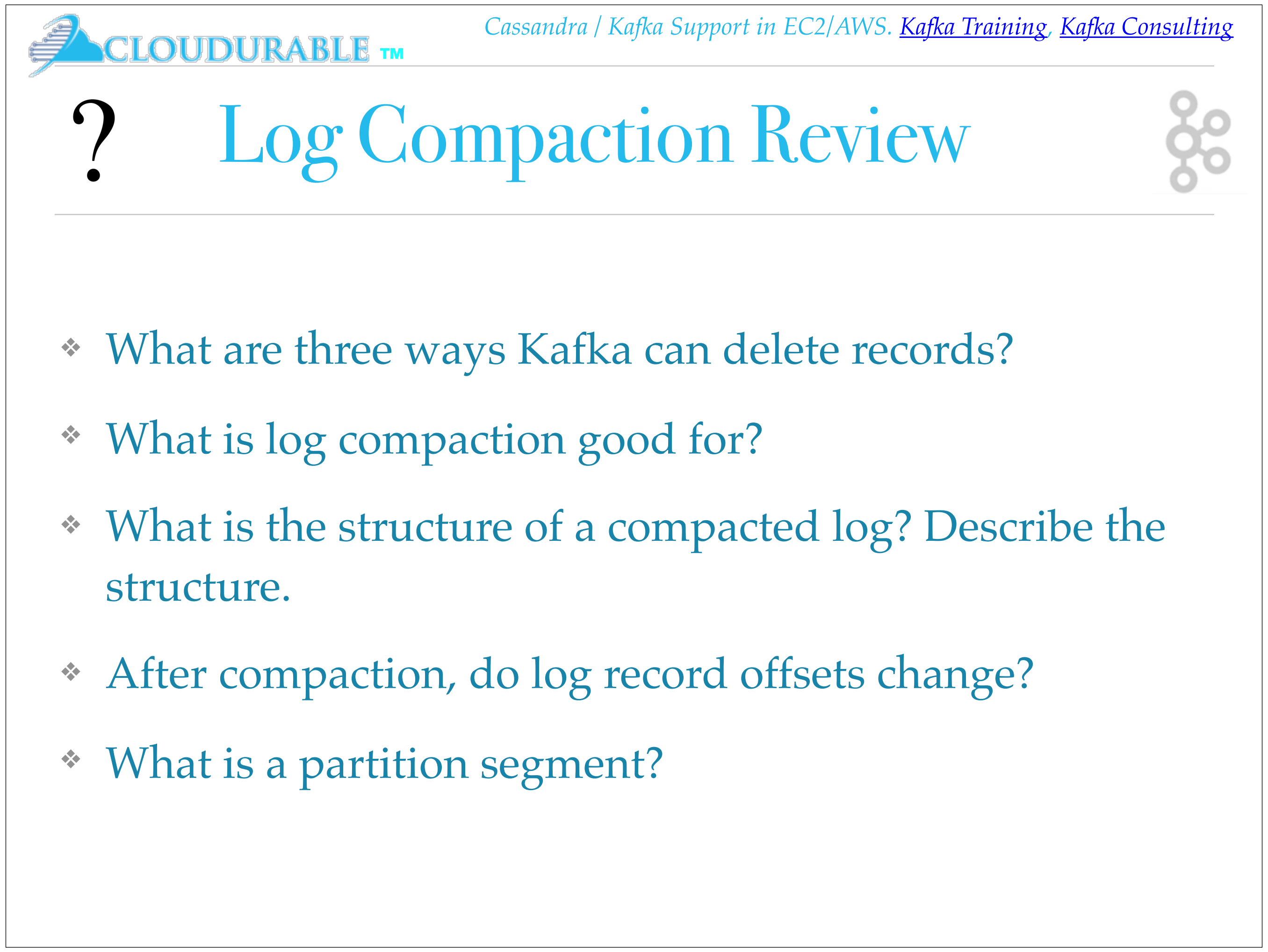
- To turn on compaction for a topic
  - topic config ***log.cleanup.policy=compact***
- To start compacting records after they are written
  - topic config ***log.cleaner.min.compaction.lag.ms***
  - Records wont be compacted until after this period

To turn on compaction for a topic use topic config log.cleanup.policy=compact. To set delay to start compacting records after they are written use topic config log.cleaner.min.compaction.lag.ms. Records won't get compacted until after this period. The setting gives consumers time to get every record.

Broker Config for Log Compaction

Kafka Broker Log Compaction Config

NAME	DESCRIPTION	TYPE	DEFAULT
log.cleaner.backoff.ms	Sleep period when no logs need cleaning	long	15,000
log.cleaner.dedupe.buffer.size	The total memory for log dedupe process for all cleaner threads	long	134,217,728
log.cleaner.delete.retention.ms	How long record delete markers (tombstones) are retained.	long	86,400,000
log.cleaner.enable	Turn on the Log Cleaner. You should turn this on if any topics are using clean.policy=compact.	boolean	TRUE
log.cleaner.io.buffer.size	Total memory used for log cleaner I/O buffers for all cleaner threads	int	524,288
log.cleaner.io.max.bytes.per.second	This is a way to throttle the log cleaner if it is taking up too much time.	double	1.7976931348623157E308
log.cleaner.min.cleanable.ratio	The minimum ratio of dirty head log to total log (head and tail) for a log to get selected for cleaning.	double	0.5
log.cleaner.min.compaction.lag.ms	Minimum time period a new message will remain uncompacted in the log.	long	0
log.cleaner.threads	Threads count used for log cleaning. Increase this if you have a lot of log compaction going on across many topic log partitions.	int	1
log.cleanup.policy	The default cleanup policy for segment files that are beyond their retention window. Valid policies are: "delete" and “compact”. You could use log compaction just for older segment files. instead of deleting them, you could just compact them.	list	[delete]



? Log Compaction Review

- What are three ways Kafka can delete records?
- What is log compaction good for?
- What is the structure of a compacted log? Describe the structure.
- After compaction, do log record offsets change?
- What is a partition segment?

What are three ways Kafka can delete records? Kafka can delete older records based on time or size of a log. Kafka also supports log compaction for record key compaction.

What is log compaction good for? Since Log compaction retains last known value it is a full snapshot of the latest records it is useful for restoring state after a crash or system failure for an in-memory service, a persistent data store, or reloading a cache. It allows downstream consumers to restore their state.

What is the structure of a compacted log? With a compacted log, the log has head and tail. The head of the compacted log is identical to a traditional Kafka log. New records get appended to the end of the head. All log compaction works at the tail of the compacted log.

After compaction, do log record offsets change? No.

What is a partition segment? Recall that a topic has a log. A topic log is broken up into partitions and partitions are divided into segment files which contain records which have keys and values. Segment files allow for divide and conquer when it comes to log compaction. A segment file is part of the partition. As the log cleaner cleans log partition segments, the segments get swapped into the log partition immediately replacing the older segment files. This way compaction does not require double the space of the entire partition as additional disk space required is just one additional log partition segment.

References

- **Learning Apache Kafka**, Second Edition 2nd Edition by Nishant Garg (Author), 2015, ISBN 978-1784393090, Packet Press
- ***Apache Kafka Cookbook***, 1st Edition, Kindle Edition by Saurabh Minni (Author), 2015, ISBN 978-1785882449, Packet Press
- Kafka Streams for Stream processing: A few words about how Kafka works, Serban Balamaci, 2017, ***Blog: Plain Ol' Java***
- Kafka official documentation, 2017
- Why we need Kafka? Quora
- Why is Kafka Popular? Quora
- Why is Kafka so Fast? Stackoverflow
- Kafka growth exploding (Tech Republic)