Cassandra and Kafka Support on AWS/EC2

Cloudurable
Introduction to Kafka

Support around Cassandra and Kafka running in EC2

Kafka growing

Why Kafka?

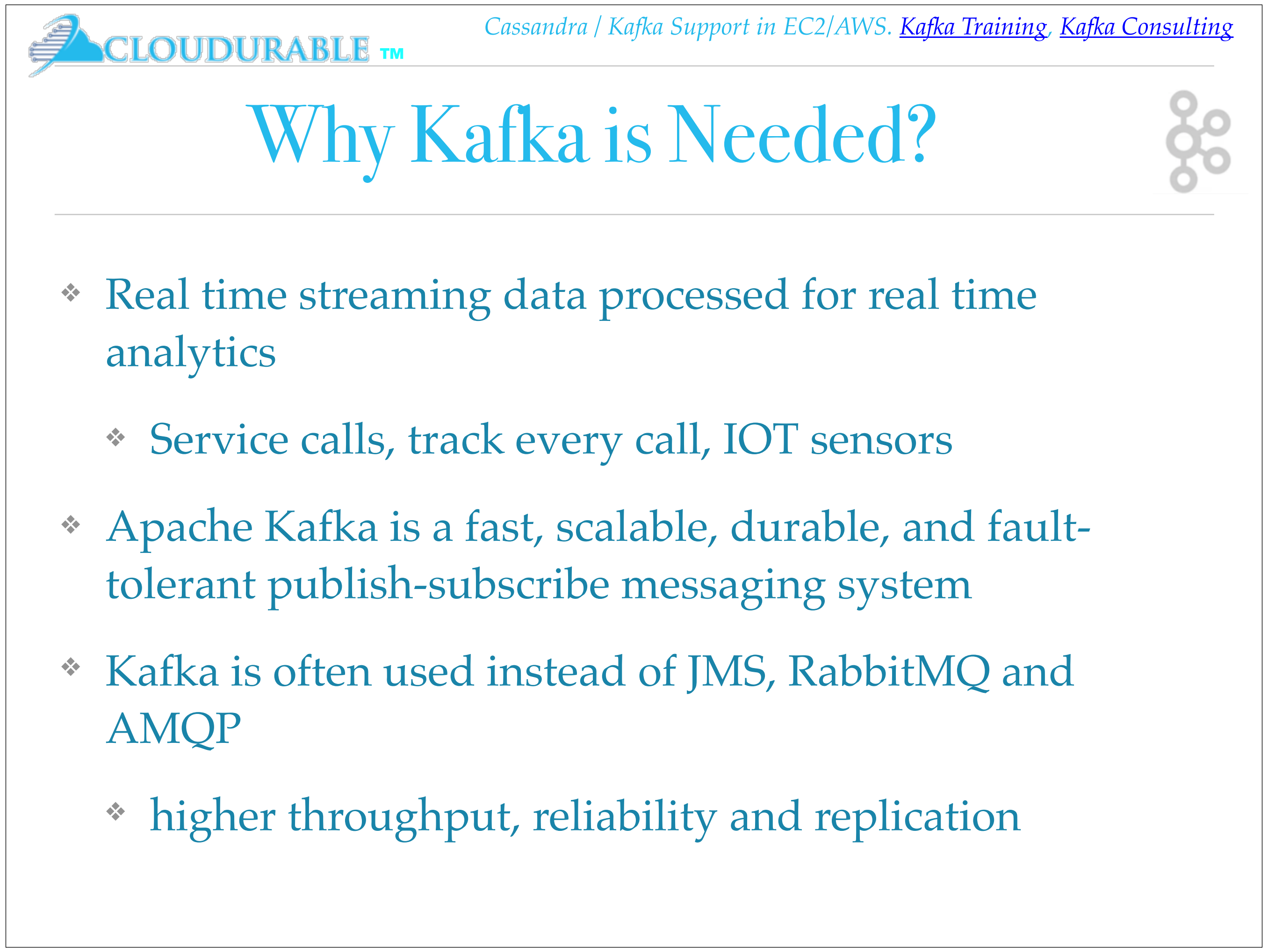
Kafka adoption is on the rise but why

What is Kafka?

Kafka growth exploding

- Kafka growth exploding
- 1/3 of all Fortune 500 companies
- Top ten travel companies, 7 of top ten banks, 8 of top ten insurance companies, 9 of top ten telecom companies
- LinkedIn, Microsoft and Netflix process 4 comma message a day with Kafka (1,000,000,000,000)
- Real-time streams of data, used to collect big data or to do real time analysis (or both)

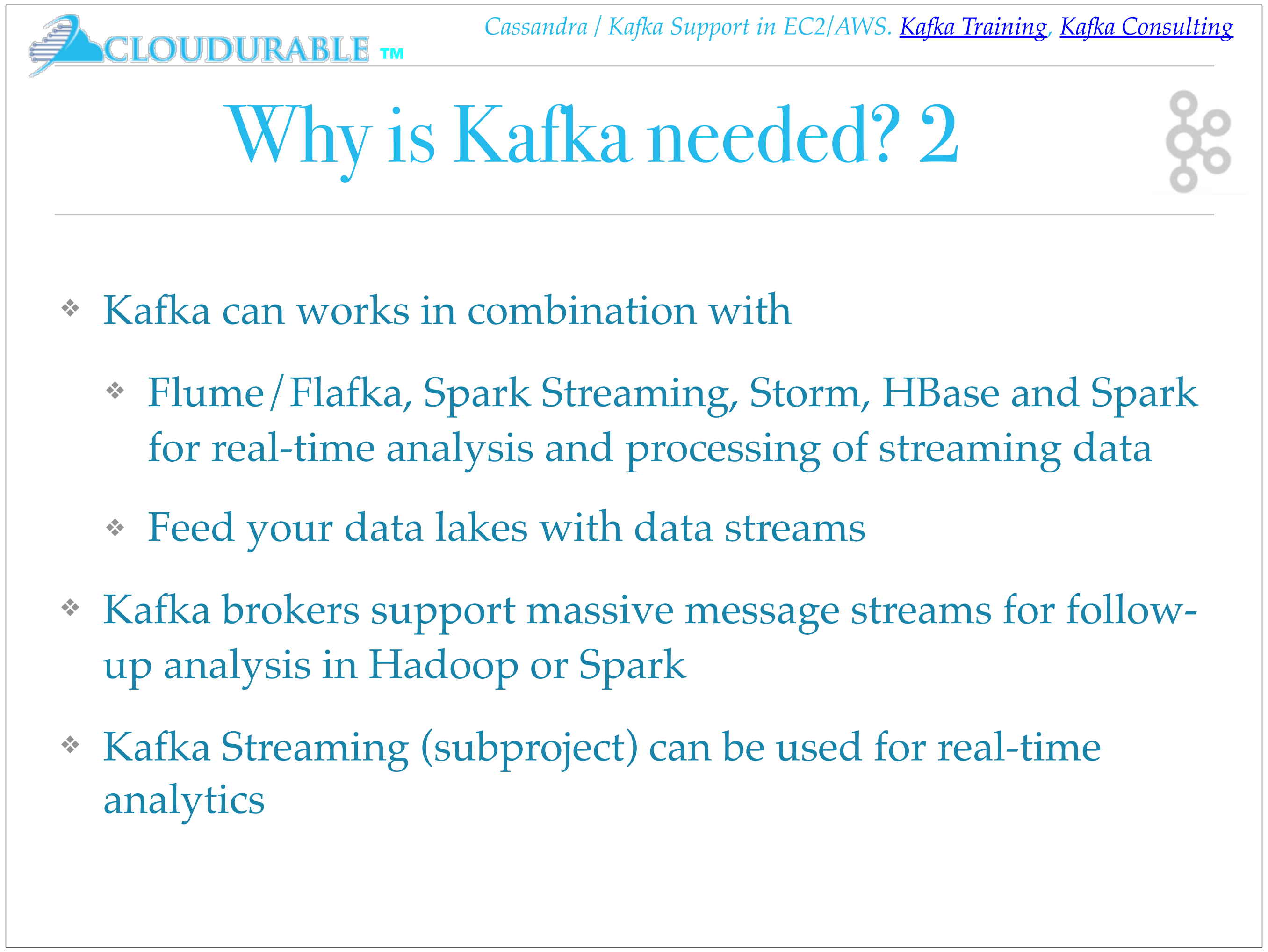
Kafka's growth is exploding, more than 1⁄3 of all Fortune 500 companies use Kafka. These companies includes the top ten travel companies, 7 of top ten banks, 8 of top ten insurance companies, 9 of top ten telecom companies, and much more. LinkedIn, Microsoft and Netflix process four comma messages a day with Kafka (1,000,000,000,000). Kafka is used for real-time streams of data, used to collect big data or to do real time analysis or both). Kafka is used with in-memory microservices to provide durability it can be used to feed events to CEP (complex event streaming systems), and IOT/IFTTT style automation systems.



Why Kafka is Needed?

- Real time streaming data processed for real time analytics
  - Service calls, track every call, IOT sensors
- Apache Kafka is a fast, scalable, durable, and fault-tolerant publish-subscribe messaging system
- Kafka is often used instead of JMS, RabbitMQ and AMQP
  - higher throughput, reliability and replication

Kafka often gets used in the real-time streaming data architectures to provide real-time analytics. Since Kafka is a fast, scalable, durable, and fault-tolerant publish-subscribe messaging system, Kafka is used in use cases where JMS, RabbitMQ, and AMQP may not even be considered due to volume and responsiveness. Kafka has higher throughput, reliability and replication characteristics which make it applicable for things like tracking service calls (tracks every call) or track IOT sensors data where a traditional MOM might not be considered.



Why is Kafka needed? 2

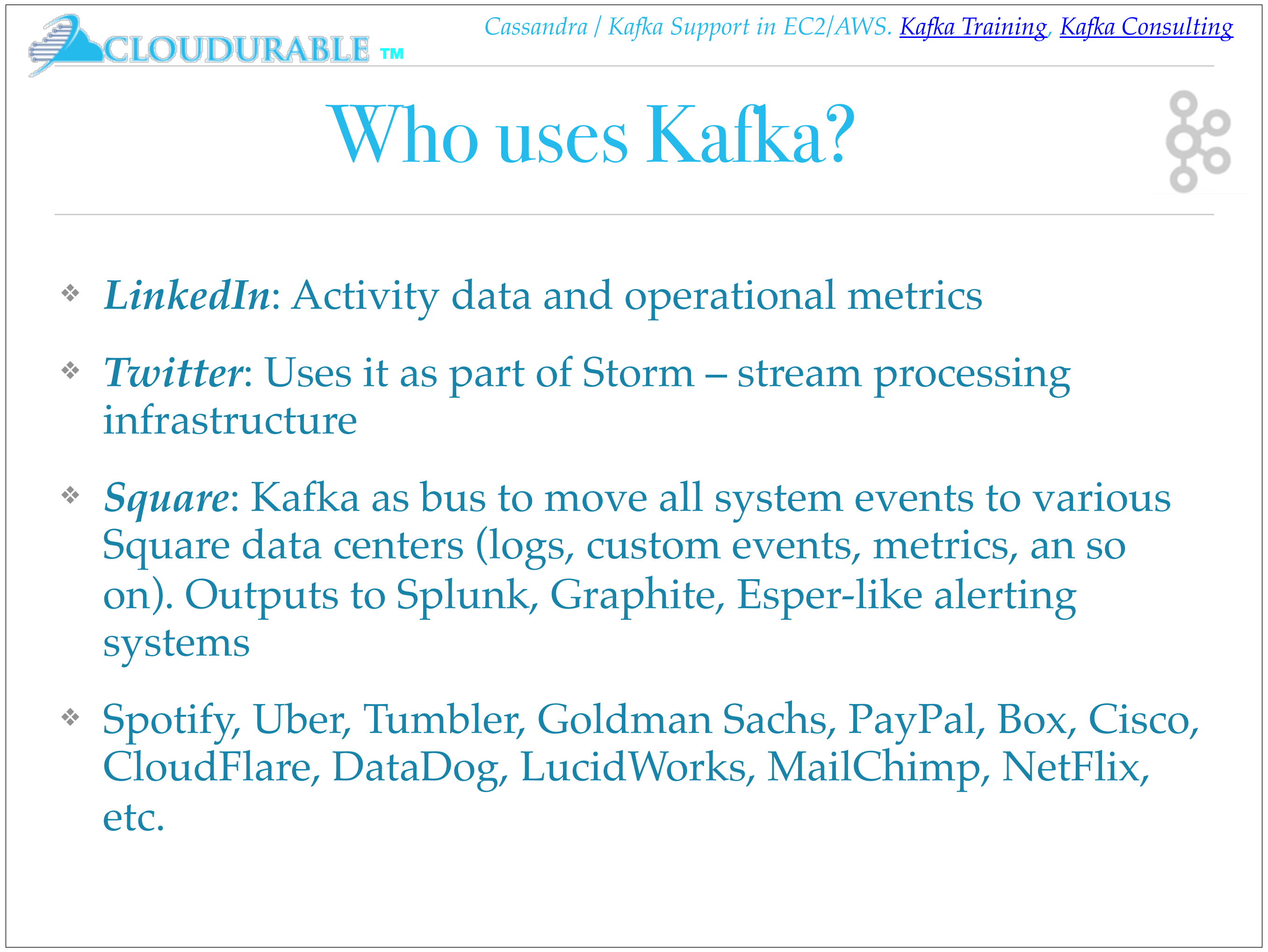
- Kafka can works in combination with
  - Flume/Flafka, Spark Streaming, Storm, HBase and Spark for real-time analysis and processing of streaming data
  - Feed your data lakes with data streams
- Kafka brokers support massive message streams for follow-up analysis in Hadoop or Spark
- Kafka Streaming (subproject) can be used for real-time analytics

Kafka can works with Flume/Flafka, Spark Streaming, Storm, HBase, Flink and Spark for real-time ingesting, analysis and processing of streaming data. Kafka is a data stream used to feed Hadoop BigData lakes. Kafka brokers support massive message streams for low-latency analysis in Hadoop or Spark. Also, Kafka Streaming (a subproject) can be used for real-time analytics.

Kafka Use Cases

- Stream Processing
- Website Activity Tracking
- Metrics Collection and Monitoring
- Log Aggregation
- Real time analytics
- Capture and ingest data into Spark / Hadoop
- CRQS, replay, error recovery
- Guaranteed distributed commit log for in-memory computing

In short, Kafka gets used for stream processing, website activity tracking, metrics collection and monitoring, log aggregation, real-time analytics, CEP systems, ingesting data into Spark, ingesting data into Hadoop, CQRS, replay messages, error recovery, and guaranteed distributed commit log for in-memory computing (microservices).

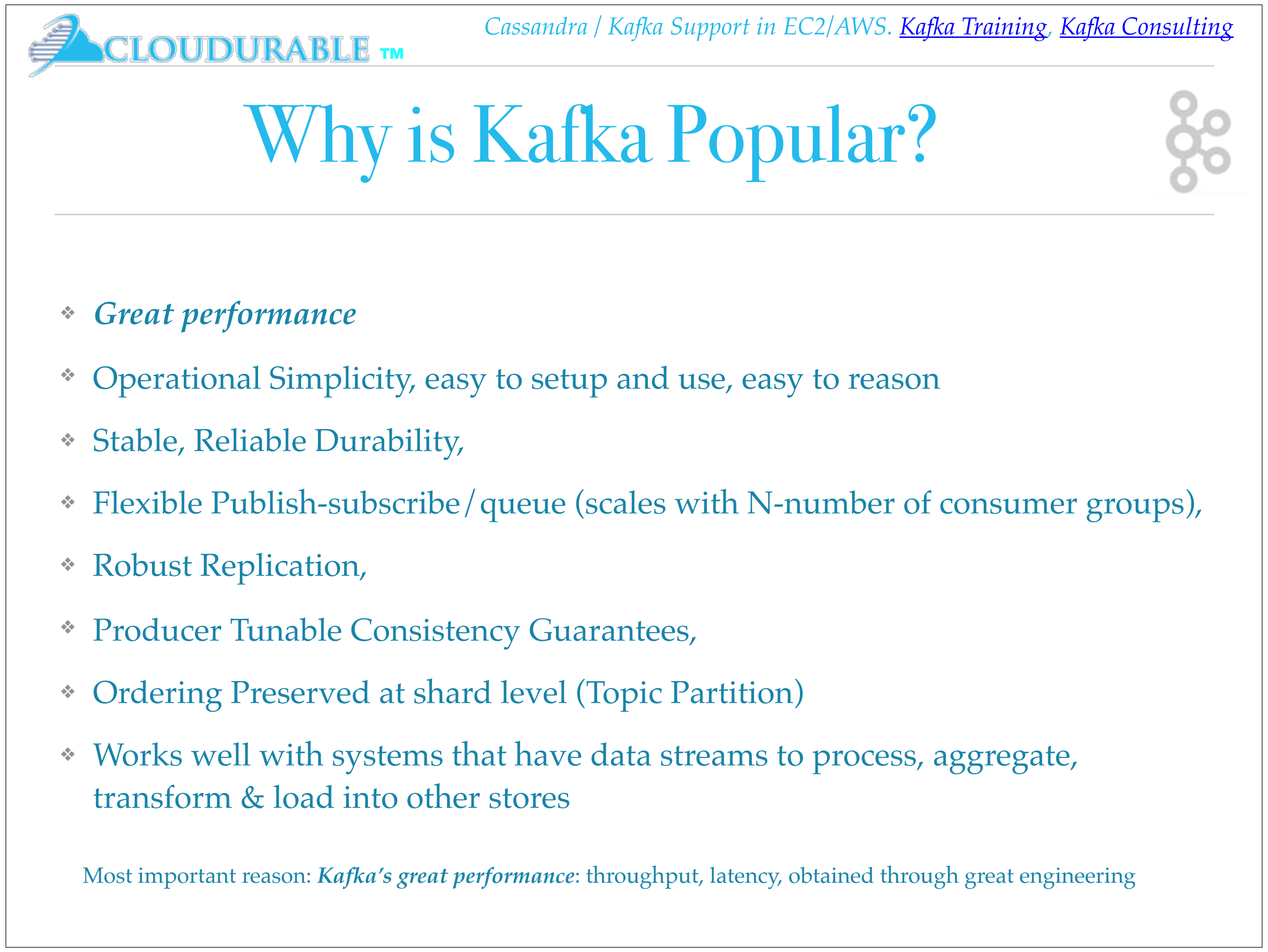


Who uses Kafka?

- ***LinkedIn***: Activity data and operational metrics
- ***Twitter***: Uses it as part of Storm – stream processing infrastructure
- ***Square***: Kafka as bus to move all system events to various Square data centers (logs, custom events, metrics, an so on). Outputs to Splunk, Graphite, Esper-like alerting systems
- Spotify, Uber, Tumbler, Goldman Sachs, PayPal, Box, Cisco, CloudFlare, DataDog, LucidWorks, MailChimp, NetFlix, etc.

A lot of large companies who handle a lot of data use Kafka. LinkedIn, where it originated, uses it to track activity data and operational metrics. Twitter uses it as part of Storm to provide a stream processing infrastructure. Square uses Kafka as a bus to move all system events to various Square data centers (logs, custom events, metrics, and so on), outputs to Splunk, Graphite (dashboards), and to implement an Esper-like/CEP alerting systems. It gets used by other companies too like Spotify, Uber, Tumbler, Goldman Sachs, PayPal, Box, Cisco, CloudFlare, NetFlix, and much more.

https://cwiki.apache.org/confluence/display/KAFKA/Powered+By

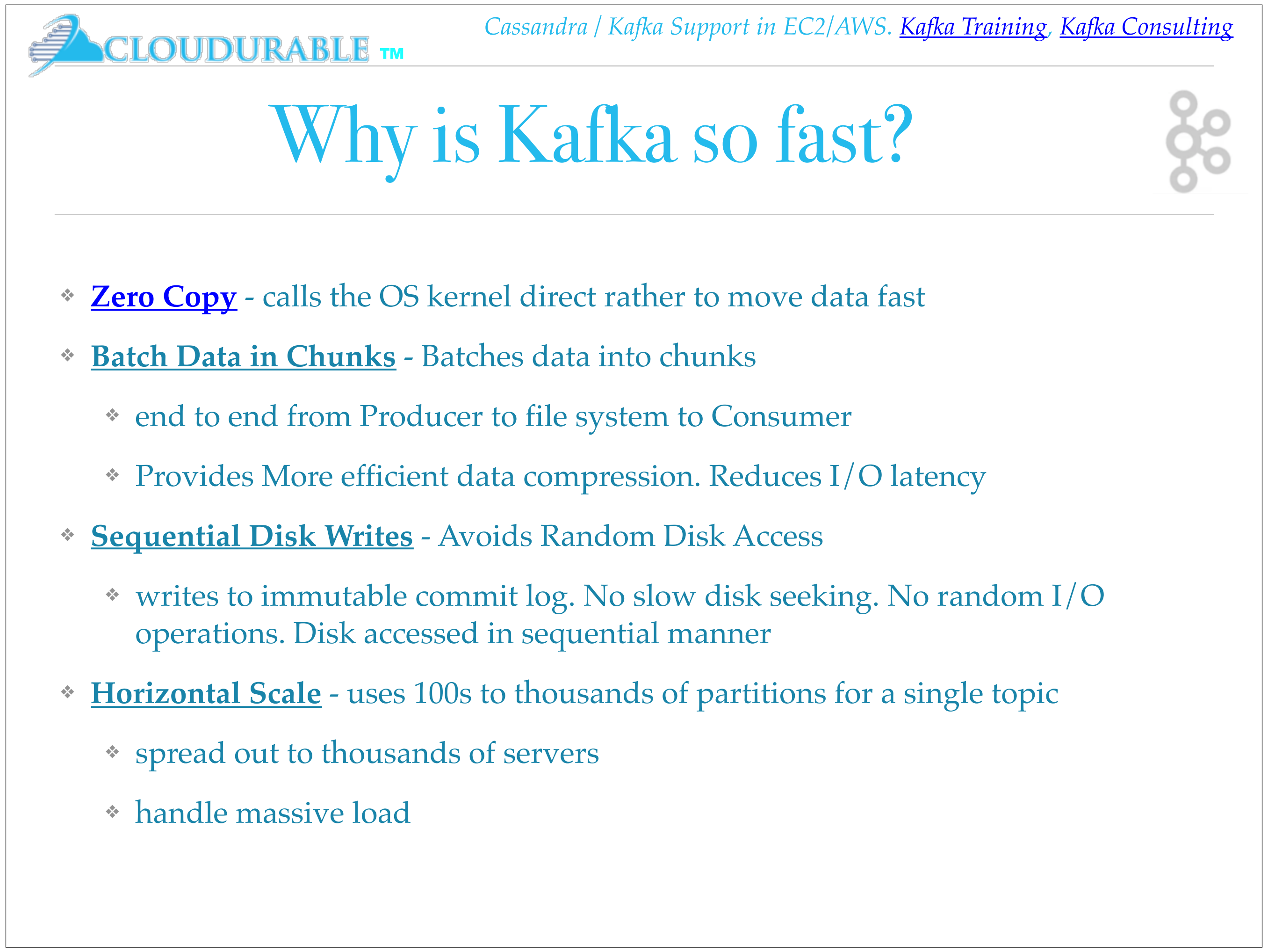


Why is Kafka Popular?

- *Great performance*
- Operational Simplicity, easy to setup and use, easy to reason
- Stable, Reliable Durability,
- Flexible Publish-subscribe/queue (scales with N-number of consumer groups),
- Robust Replication,
- Producer Tunable Consistency Guarantees,
- Ordering Preserved at shard level (Topic Partition)
- Works well with systems that have data streams to process, aggregate, transform & load into other stores

Most important reason: ***Kafka's great performance***: throughput, latency, obtained through great engineering

Kafka has operational simplicity. Kafka is to set up and use, and it is easy to reason how Kafka works. However, the main reason Kafka is very popular is its excellent performance. It has other characteristics as well, but so do other messaging systems. Kafka has great performance, and it is stable, provides reliable durability, has a flexible publish-subscribe/queue that scales well with N-number of consumer groups, has robust replication, provides Producers with tunable consistency guarantees, and it provides preserved ordering at shard level (Kafka Topic Partition). In addition, Kafka works well with systems that have data streams to process and enables those systems to aggregate, transform & load into other stores. But none of those characteristics would matter if Kafka was slow. The most important reason Kafka is popular is Kafka's exceptional performance.

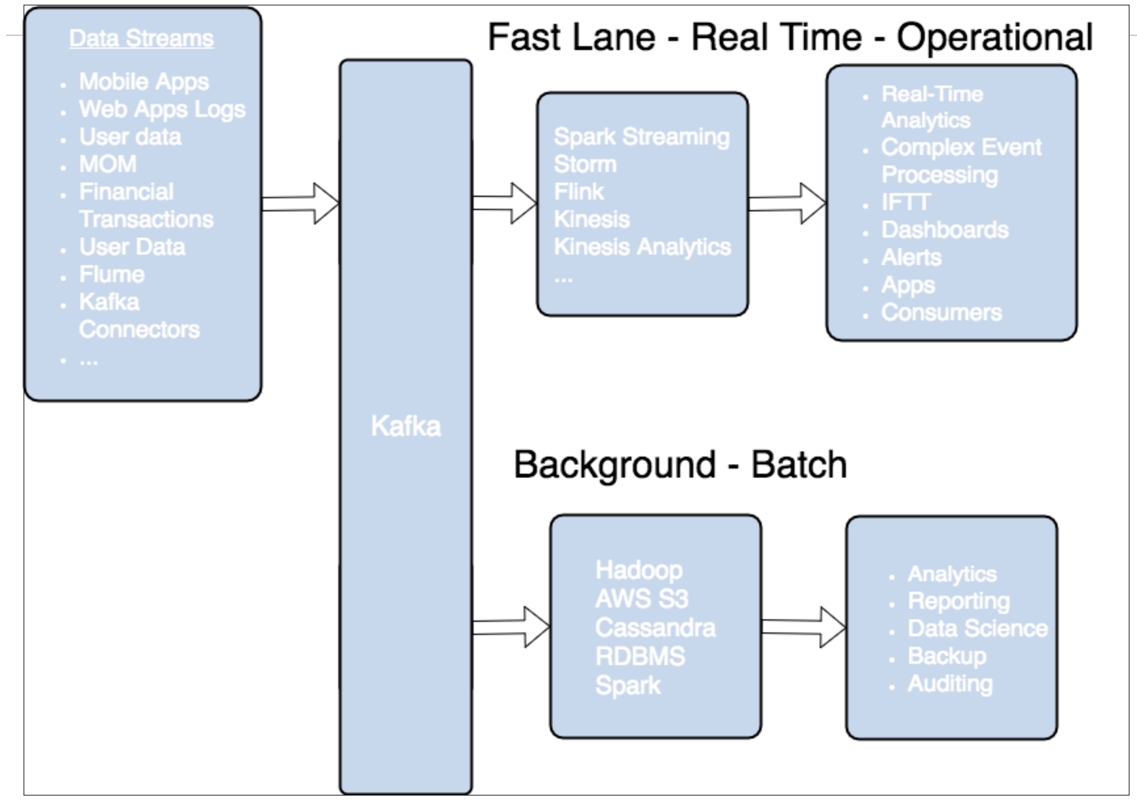


Why is Kafka so fast?

- **Zero Copy** - calls the OS kernel direct rather to move data fast
- **Batch Data in Chunks** - Batches data into chunks
  - end to end from Producer to file system to Consumer
  - Provides More efficient data compression. Reduces I/O latency
- **Sequential Disk Writes** - Avoids Random Disk Access
  - writes to immutable commit log. No slow disk seeking. No random I/O operations. Disk accessed in sequential manner
- **Horizontal Scale** - uses 100s to thousands of partitions for a single topic
  - spread out to thousands of servers
  - handle massive load

Kafka relies heavily on the OS kernel to move data around quickly which keeps the JVM heap size small. It relies on the principals of Zero Copy. Kafka enables you to batch data records into chunks. These batches of data can be seen end to end from Producer to file system (Kafka Topic Log) to the Consumer. Batching allows for more efficient data compression and reduces I/O latency between consumers and producers. Kafka writes to the immutable commit log to the disk sequential; thus, avoids random disk access, slow disk seeking. Kafka provides horizontal Scale through sharding. It shards a Topic Log into hundreds potentially thousands of partitions. This sharding allows Kafka to handle massive load.

Streaming Architecture



Kafka gets used most often for real-time streaming of data into other systems. Kafka is a middle layer to decouple your real-time data pipelines. Kafka core is not good for direct computations such as data aggregations, or CEP. Kafka Streaming which is part of the Kafka ecosystem does provide the ability to do real-time analytics. Kafka can be used to feed fast lane systems (real-time, and operational data systems) like Storm, Flink, Spark Streaming and your services and CEP systems. Kafka is also used to stream data for batch data analysis. Kafka feeds Hadoop. It streams data into your BigData platform or into RDBMS, Cassandra, Spark, or even S3 for some future data analysis. These data stores often support data analysis, reporting, data science crunching, compliance auditing, and backups.

? Why Kafka Review

- Why is Kafka so fast?
- How fast is Kafka usage growing?
- How is Kafka getting used?
- Where does Kafka fit in the Big Data Architecture?
- How does Kafka relate to real-time analytics?
- Who uses Kafka?

Why is Kafka so fast?
Kafka is fast because it avoids copying buffers in-memory (Zero Copy), and streams data to immutable logs instead of using random access.

How fast is Kafka usage growing?
Over 1/3 of fortune 500 companies use Kafka.

How is Kafka getting used?
Kafka is used to feed data lakes like Hadoop, and to feed real-time analytics systems like Flink, Storm and Spark Streaming.

Where does Kafka fit in the Big Data Architecture?
Kafka is a data stream that fills up Big Data's data lakes.

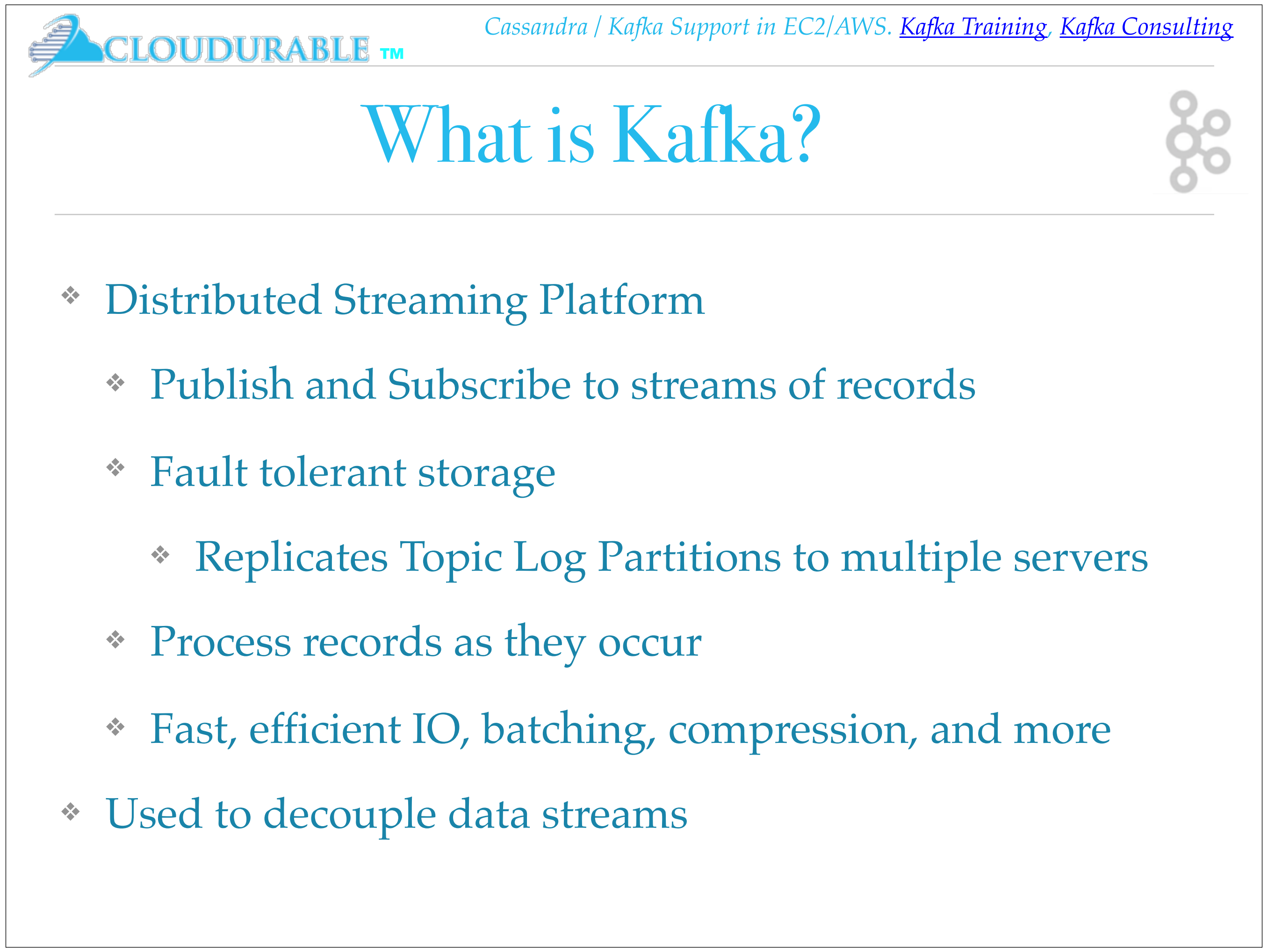
How does Kafka relate to real-time analytics?
Kafka feeds data to real-time analytics systems like Storm, Spark Streaming, Flink, and Kafka Streaming.

Who uses Kafka?
The top ten travel companies, 7 of top ten banks, 8 of top ten insurance companies, 9 of top ten telecom companies, LinkedIn, Microsoft, Netflix and many more companies.

Cassandra / Kafka Support in EC2/AWS

What is Kafka?

Kafka messaging
Kafka Overview

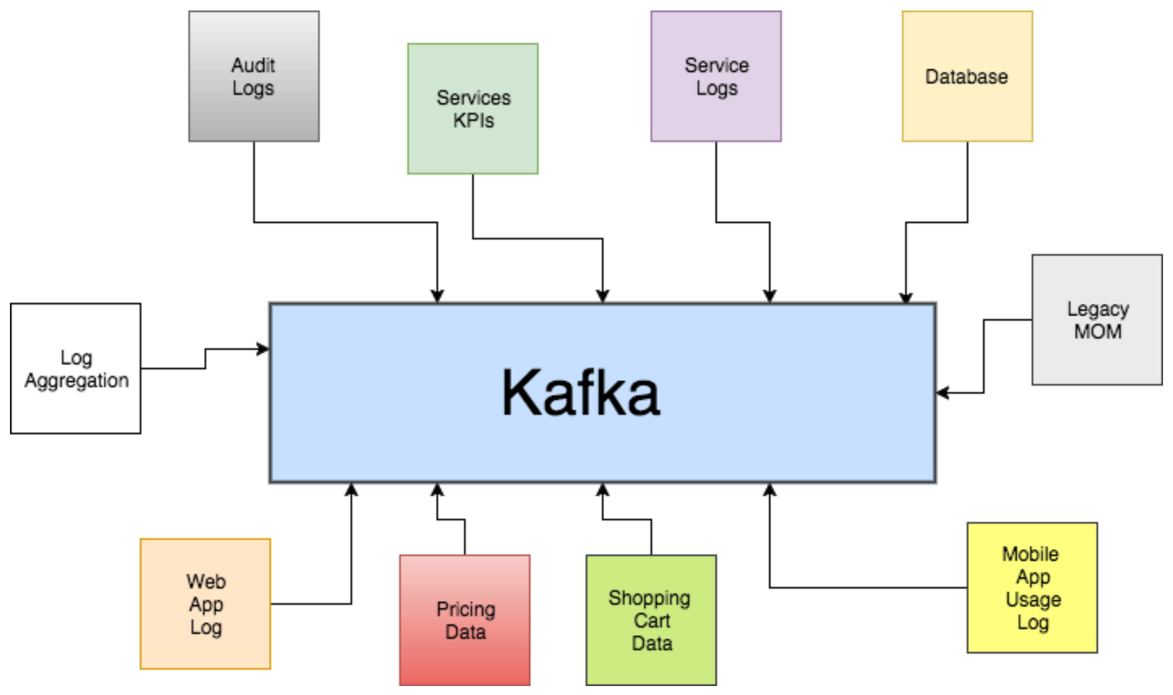


What is Kafka?

- Distributed Streaming Platform
  - Publish and Subscribe to streams of records
  - Fault tolerant storage
    - Replicates Topic Log Partitions to multiple servers
  - Process records as they occur
  - Fast, efficient IO, batching, compression, and more
- Used to decouple data streams

Kafka is a distributed streaming platform that is used publish and subscribe to streams of records. Kafka gets used for fault tolerant storage. Kafka replicates topic log partitions to multiple servers. Kafka is designed to allow your apps to process records as they occur. Kafka is fast, uses IO efficiently by batching, compressing records. Kafka gets used for decoupling data streams. Kafka is used to stream data into data lakes, applications and real-time stream analytics systems.

Kafka Decoupling Data Streams



Kafka decouple streams of data by allowing multiple consumer groups that can each control where in the topic partition they are. The producers don't know about the consumers. Since the Kafka broker delegates the log partition offset (where the consumer is in the record stream) to the clients (Consumers), the message consumption is flexible. This allows you to feed your high-latency daily or hourly data analysis in Spark and Hadoop and the same time you are feeding microservices real-time messages, sending events to your CEP system and feeding data to your real-time analytic systems.

Kafka Polyglot clients / Wire protocol

- Kafka communication from clients and servers wire protocol over TCP protocol
- Protocol versioned
- Maintains backwards compatibility
- Many languages supported
- Kafka REST proxy allows easy integration
- Also provides Avro/Schema registry support via Kafka ecosystem

Kafka communication from clients and servers uses a wire protocol over TCP that is versioned and documented. Kafka promises to maintain backward compatibility with older clients, and many languages are supported. The Kafka ecosystem also provides REST proxy allows easy integration via HTTP and JSON. Kafka also supports Avro via the Confluent Schema Registry for Kafka. Avro and the Schema Registry allows complex records to be produced and read by clients in many programming languages and allows for the evolution of the records. Kafka is truly polyglot.

Kafka Usage

- Build real-time streaming applications that react to streams
  - Real-time data analytics
  - Transform, react, aggregate, join real-time data flows
  - Feed events to CEP for complex event processing
  - Feed data lakes
- Build real-time streaming data pipe-lines
  - Enable in-memory microservices (actors, Akka, Vert.x, Qbit, RxJava)

Kafka allows you to build real-time streaming data pipe-lines. Kafka enable in-memory microservices (actors, Akka, Baratine.io, QBit, reactors, reactive, Vert.x, RxJava, Spring Reactor) Kafka allows you to build real-time streaming applications that react to streams to do real-time data analytics, transform, react, aggregate, join real-time data flows and perform CEP (complex event processing).
You can use Kafka to aid in gathering Metrics/KPIs, aggregate statistics from many sources implement event sourcing, use it with microservices (in-memory) and actor systems to implement in-memory services (external commit log for distributed systems).
You can use Kafka to replicate data between nodes, to re-sync for nodes, to restore state. While it is true, Kafka used for real-time data analytics and stream processing, you can also use it for log aggregation, messaging, click-stream tracking, audit trails, and much more.
In a world where data science and analytics is a big deal, then capturing data to feed into your data lakes and real-time analytics systems is a big deal, and since Kafka can hold up to these kinds of strenuous use cases, Kafka is a big deal.

Kafka Use Cases

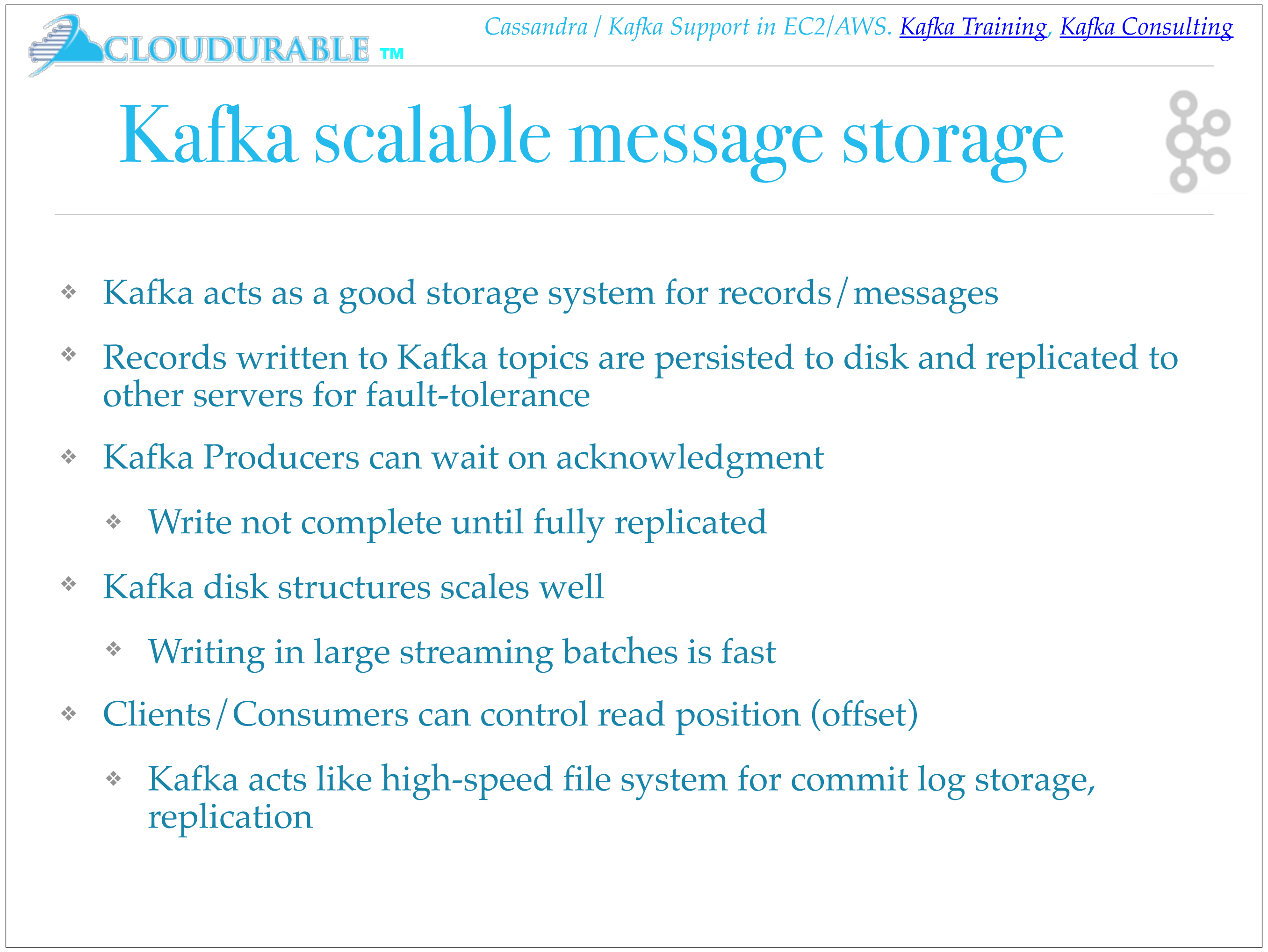
- Metrics / KPIs gathering
  - Aggregate statistics from many sources
- Event Sourcing
  - Used with microservices (in-memory) and actor systems
- Commit Log
  - External commit log for distributed systems. Replicated data between nodes, re-sync for nodes to restore state
- Real-time data analytics, Stream Processing, Log Aggregation, Messaging, Click-stream tracking, Audit trail, etc.

Kafka allows you to build real-time streaming data pipe-lines. Kafka enable in-memory microservices (actors, Akka, Baratine.io, QBit, reactors, reactive, Vert.x, RxJava, Spring Reactor) Kafka allows you to build real-time streaming applications that react to streams to do real-time data analytics, transform, react, aggregate, join real-time data flows and perform CEP (complex event processing).
You can use Kafka to aid in gathering Metrics/KPIs, aggregate statistics from many sources implement event sourcing, use it with microservices (in-memory) and actor systems to implement in-memory services (external commit log for distributed systems).
You can use Kafka to replicate data between nodes, to re-sync for nodes, to restore state. While it is true, Kafka used for real-time data analytics and stream processing, you can also use it for log aggregation, messaging, click-stream tracking, audit trails, and much more.
In a world where data science and analytics is a big deal, then capturing data to feed into your data lakes and real-time analytics systems is a big deal, and since Kafka can hold up to these kinds of strenuous use cases, Kafka is a big deal.

Kafka Record Retention

- Kafka cluster retains all published records
  - Time based – configurable retention period
  - Size based - configurable based on size
  - Compaction - keeps latest record
- Retention policy of three days or two weeks or a month
- It is available for consumption until discarded by time, size or compaction
- Consumption speed not impacted by size

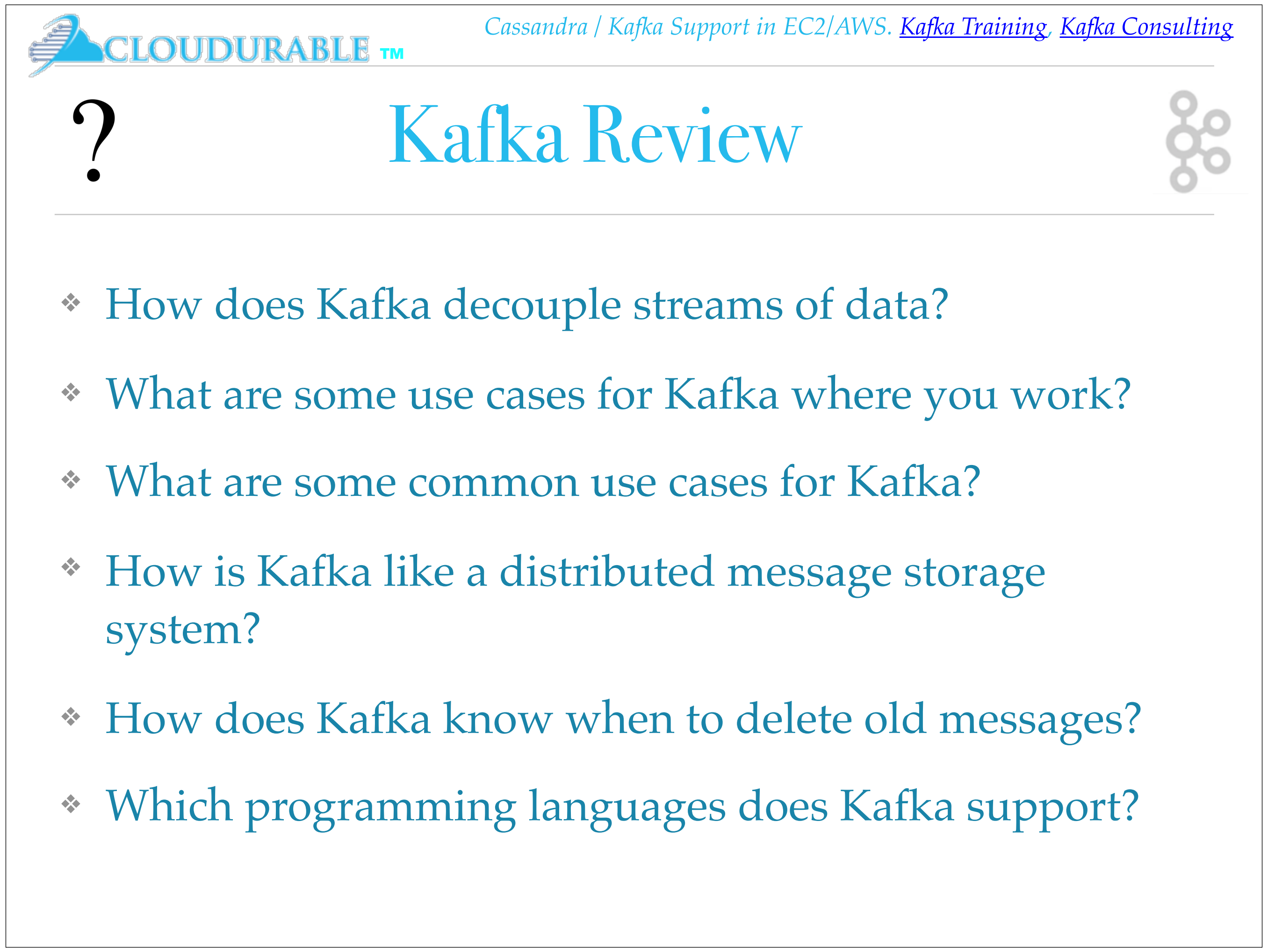
Kafka cluster retains all published records and if you don't set a limit, it will keep records until it runs out of disk space. You can set time-based limits (configurable retention period), size-based limits (configurable based on size), or use compaction (keeps the latest version of record using key). You can, for example, set a retention policy of three days or two weeks or a month. The records in the topic log are available for consumption until discarded by time, size or compaction. The consumption speed not impacted by size as Kafka always writes to the end of the topic log.



Kafka scalable message storage

- Kafka acts as a good storage system for records/messages
- Records written to Kafka topics are persisted to disk and replicated to other servers for fault-tolerance
- Kafka Producers can wait on acknowledgment
    - Write not complete until fully replicated
- Kafka disk structures scales well
    - Writing in large streaming batches is fast
- Clients/Consumers can control read position (offset)
    - Kafka acts like high-speed file system for commit log storage, replication

Kafka is a good storage system for records/messages. Kafka acts like high-speed file system for commit log storage and replication. These characteristics make Kafka useful for all manners of applications.   Records written to Kafka topics are persisted to disk and replicated to other servers for fault-tolerance. Since modern drives are fast and quite large, this fits well and is very useful. Kafka Producers can wait on acknowledgment, so messages are durable as the producer write not complete until the message replicates. The Kafka disk structure scales well. Modern disk drives have very high throughput when writing in large streaming batches. Also, Kafka Clients/Consumers can control read position (offset) which allows for use cases like replaying the log if there was a critical bug (fix the bug and the replay). And since offsets are tracked per consumer group, which we talk about in Kafka Architecture, the consumers can be quite flexible (like replaying the log).



? Kafka Review

- How does Kafka decouple streams of data?
- What are some use cases for Kafka where you work?
- What are some common use cases for Kafka?
- How is Kafka like a distributed message storage system?
- How does Kafka know when to delete old messages?
- Which programming languages does Kafka support?

How does Kafka decouple streams of data?

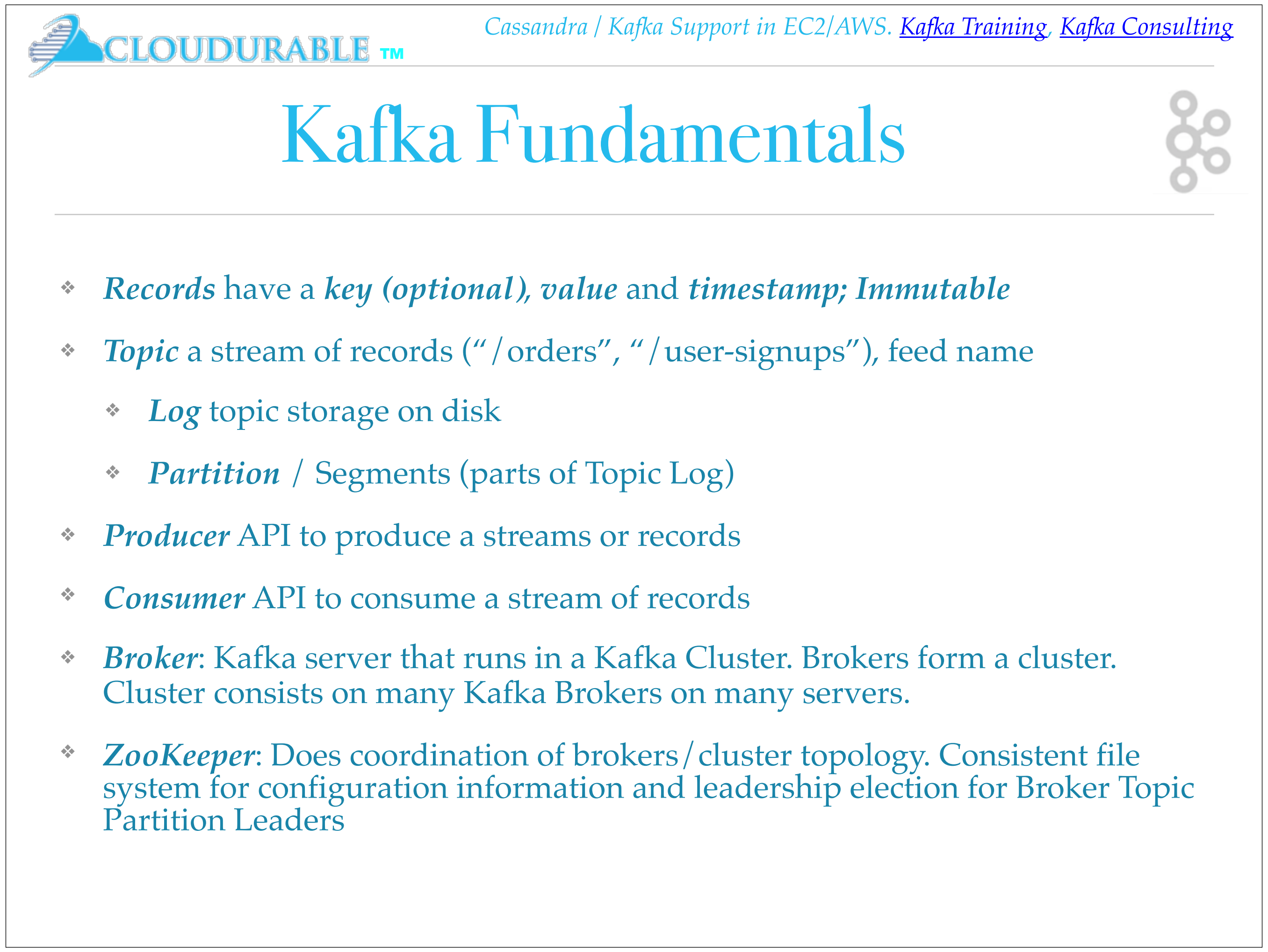
It decouple streams of data by allowing multiple consumer groups that can each control where in the topic partition they are. The producers don't know about the consumers. Since the Kafka broker delegates the log partition offset (where the consumer is in the record stream) to the clients (Consumers), the message consumption is flexible. This allows you to feed your high-latency daily or hourly data analysis in Spark and Hadoop and the same time you are feeding microservices real-time messages, sending events to your CEP system and feeding data to your real-time analytic systems.

What are some common use cases for Kafka?

Kafka feeds data to real-time analytics systems like Storm, Spark Streaming, Flink, and Kafka Streaming.

What are some use cases for Kafka where you work?

Kafka Architecture

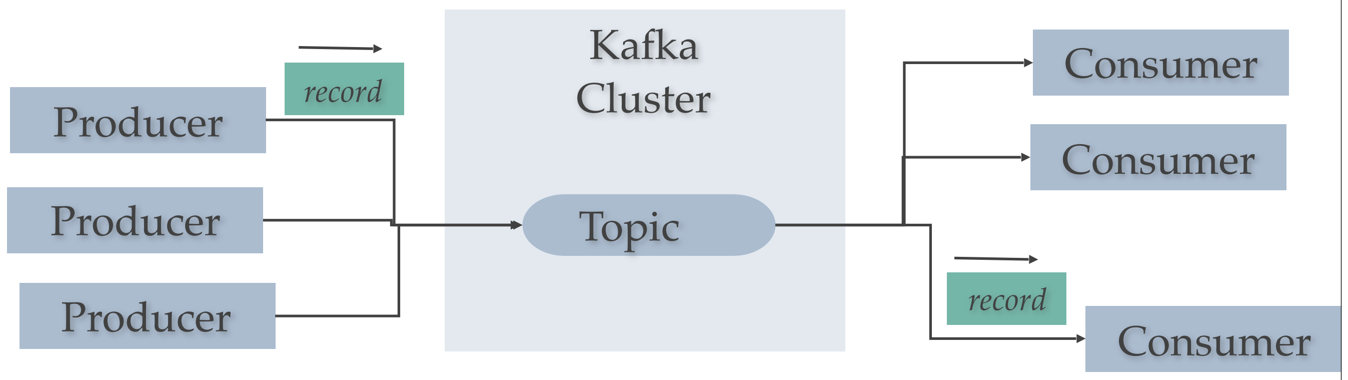


Kafka Fundamentals

- ***Records*** have a ***key (optional)***, ***value*** and ***timestamp; Immutable***
- ***Topic*** a stream of records ("/orders", "/user-signups"), feed name
  - ***Log*** topic storage on disk
  - ***Partition*** / Segments (parts of Topic Log)
- ***Producer*** API to produce a streams or records
- ***Consumer*** API to consume a stream of records
- ***Broker***: Kafka server that runs in a Kafka Cluster. Brokers form a cluster. Cluster consists on many Kafka Brokers on many servers.
- ***ZooKeeper***: Does coordination of brokers/cluster topology. Consistent file system for configuration information and leadership election for Broker Topic Partition Leaders

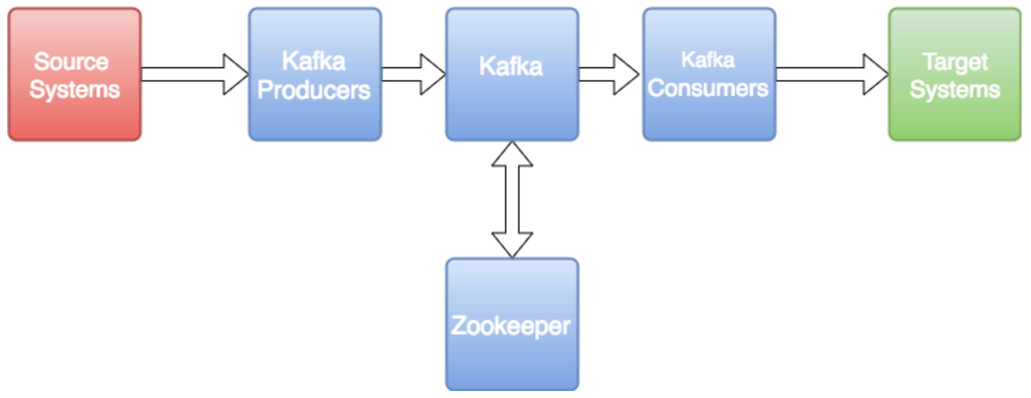
Kafka consists of Records, Topics, Consumers, Producers, Brokers, and Clusters. Records can have key (optional), value and timestamp. Kafka Records are immutable. A Kafka Topic is a stream of records ("/orders", "/user-signups"). You can think of a Topic as a feed name. A topic has a Log which is the topic's storage on disk. A Topic Log is broken up into partitions and segments. The Kafka Producer API is used to produce streams of data records. The Kafka Consumer API is used to consume a stream of records from Kafka. A Broker is a Kafka server that runs in a Kafka Cluster. Kafka Brokers form a cluster. The Kafka Cluster consists of many Kafka Brokers on many servers. Broker sometimes refer to more of a logical system or as Kafka as a whole. Kafka uses ZooKeeper to manage the cluster. ZooKeeper is used to coordinate the brokers/cluster topology. ZooKeeper is a consistent file system for configuration information. ZooKeeper gets used for leadership election for Broker Topic Partition Leaders.

Kafka: Topics, Producers, and Consumers



The above shows producers sending records to a Kafka Topic and Consumers consuming records from the Kafka cluster.

Core Kafka

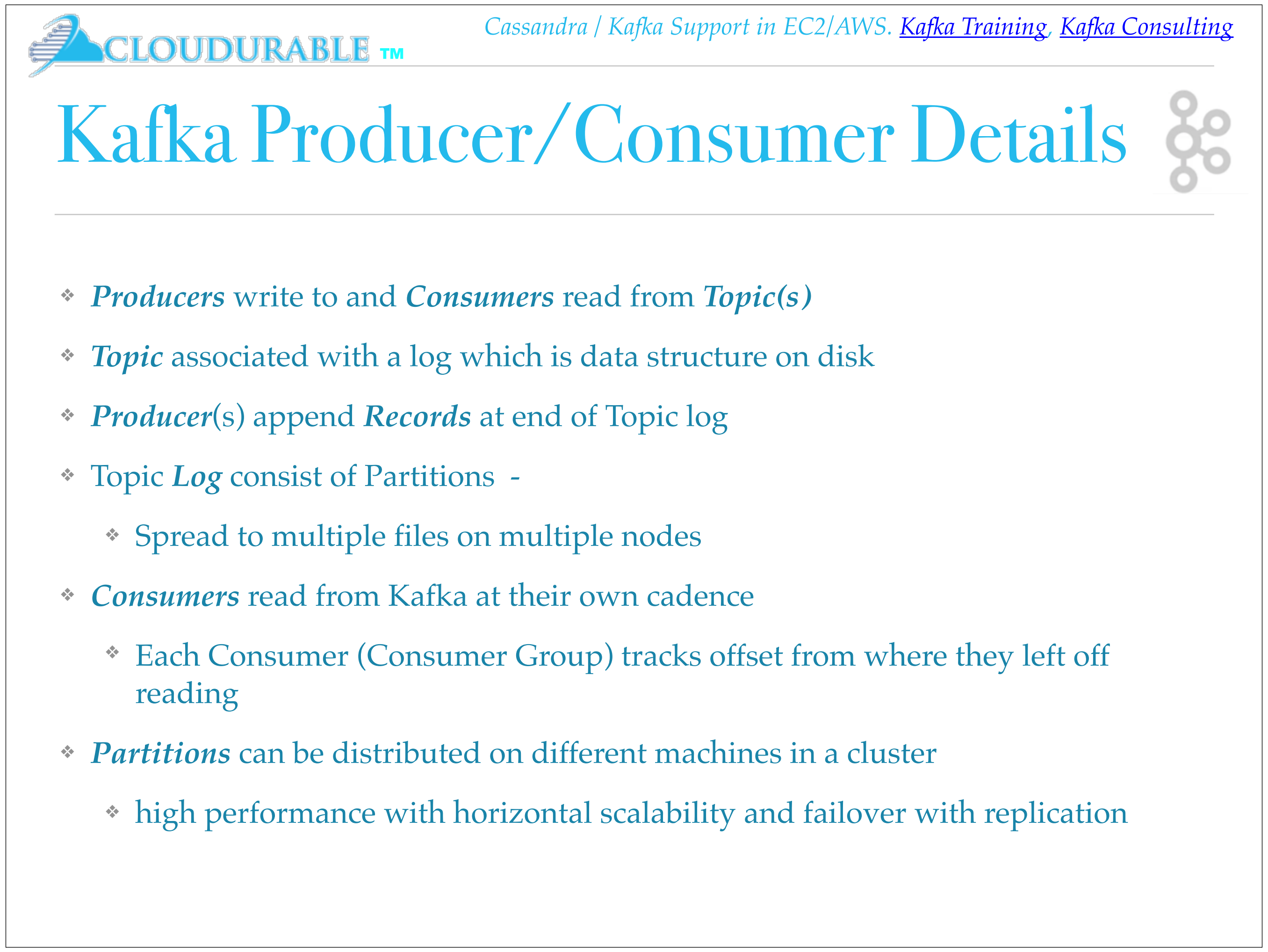


The above is the core Kafka system. Kafka has Producers, Kafka (Topics, Nodes, Cluster, Broker), and Kafka uses ZooKeeper.

Kafka needs Zookeeper

- Zookeeper helps with leadership election of Kafka Broker and Topic Partition pairs
- Zookeeper manages service discovery for Kafka Brokers that form the cluster
- Zookeeper sends changes to Kafka
  - New Broker join, Broker died, etc.
  - Topic removed, Topic added, etc.
- Zookeeper provides in-sync view of Kafka Cluster configuration

Kafka uses Zookeeper to do leadership election of Kafka Broker and Topic Partition pairs. Kafka uses Zookeeper to manage service discovery for Kafka Brokers that form the cluster.
Zookeeper sends changes of the topology to Kafka, so each node in the cluster knows when a new broker joined, a Broker died, a topic was removed or a topic was added, etc. Zookeeper provides an in-sync view of Kafka Cluster configuration.



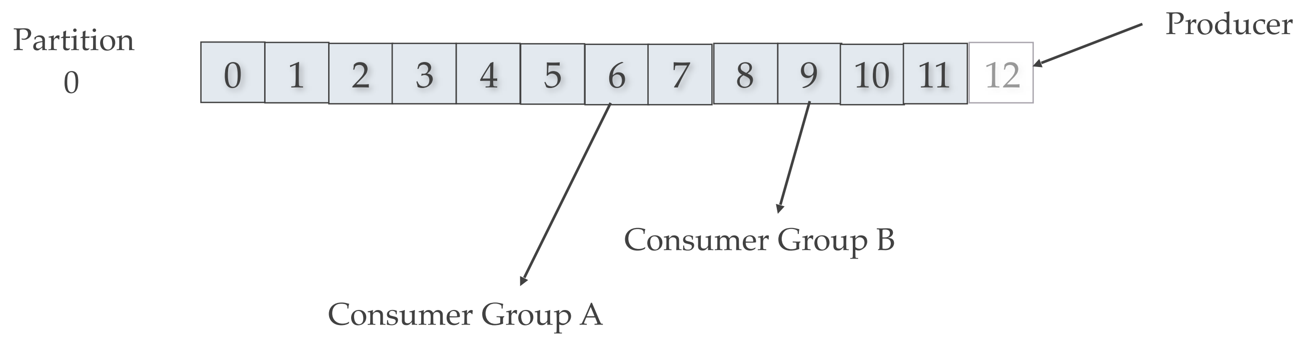
Kafka Producer/Consumer Details

- ***Producers*** write to and ***Consumers*** read from ***Topic(s)***
- ***Topic*** associated with a log which is data structure on disk
- ***Producer***(s) append ***Records*** at end of Topic log
- Topic ***Log*** consist of Partitions -
  - Spread to multiple files on multiple nodes
- ***Consumers*** read from Kafka at their own cadence
  - Each Consumer (Consumer Group) tracks offset from where they left off reading
- ***Partitions*** can be distributed on different machines in a cluster
  - high performance with horizontal scalability and failover with replication

Kafka producers write to Topics. Kafka consumers read from Topics. A topic is associated with a log which is data structure on disk. Kafka appends records from a producer(s) to the end of a topic log. A topic log consists of many partitions that are spread over multiple files which can be spread on multiple Kafka cluster nodes. Consumers read from Kafka topics at their cadence and can pick where they are (offset) in the topic log.

Each consumer group tracks offset from where they left off reading. Kafka distributes topic log partitions on different nodes in a cluster for high performance with horizontal scalability. Spreading partitions aids in writing data quickly. Topic log partitions are Kafka way to shard reads and writes to the topic log. Also, partitions are needed to have multiple consumers in a consumer group work at the same time. Kafka replicates partitions to many nodes to provide failover.

Kafka Topic Partition, Consumers, Producers



Consumer groups remember offset where they left off.
Consumers groups each have their own offset.

Producer writing to offset 12 of Partition 0 while…
Consumer Group A is reading from offset 6.
Consumer Group B is reading from offset 9.

Consumer groups remember offset where they left off.
Consumers groups each have their own offset.
Producer writing to offset 12 of Partition 0 while.
Consumer Group A is reading from offset 6.
Consumer Group B is reading from offset 9.

Kafka Scale and Speed

- How can Kafka scale if multiple producers and consumers read/write to same Kafka Topic log?
- Writes fast: Sequential writes to filesystem are ***fast*** (700 MB or more a second)
- Scales writes and reads by ***sharding:***
    - Topic logs into ***Partitions*** (parts of a Topic log)
    - Topics logs can be split into multiple Partitions ***different machines/ different disks***
    - Multiple Producers can write to different Partitions of the same Topic
    - Multiple Consumers Groups can read from different partitions efficiently

How can Kafka scale if multiple producers and consumers read/write to the same Kafka Topic log? First Kafka is fast, Kafka writes to filesystem sequentially which is fast. On a modern fast drive, Kafka can easily write up to 700 MB or more bytes of data a second. Kafka scales writes and reads by sharding topic logs into partitions (parts of a Topic log). Recall Topics logs can be split into multiple partitions which can be stored on multiple different servers, and those servers can use multiple disks. Multiple producers can write to different partitions of the same Topic. Multiple consumers from multiple consumer groups can read from different partitions efficiently.

Kafka Brokers

- Kafka Cluster is made up of multiple Kafka Brokers
- Each Broker has an ID (number)
- Brokers contain topic log partitions
- Connecting to one broker bootstraps client to entire cluster
- Start with at least three brokers, cluster can have, 10, 100, 1000 brokers if needed

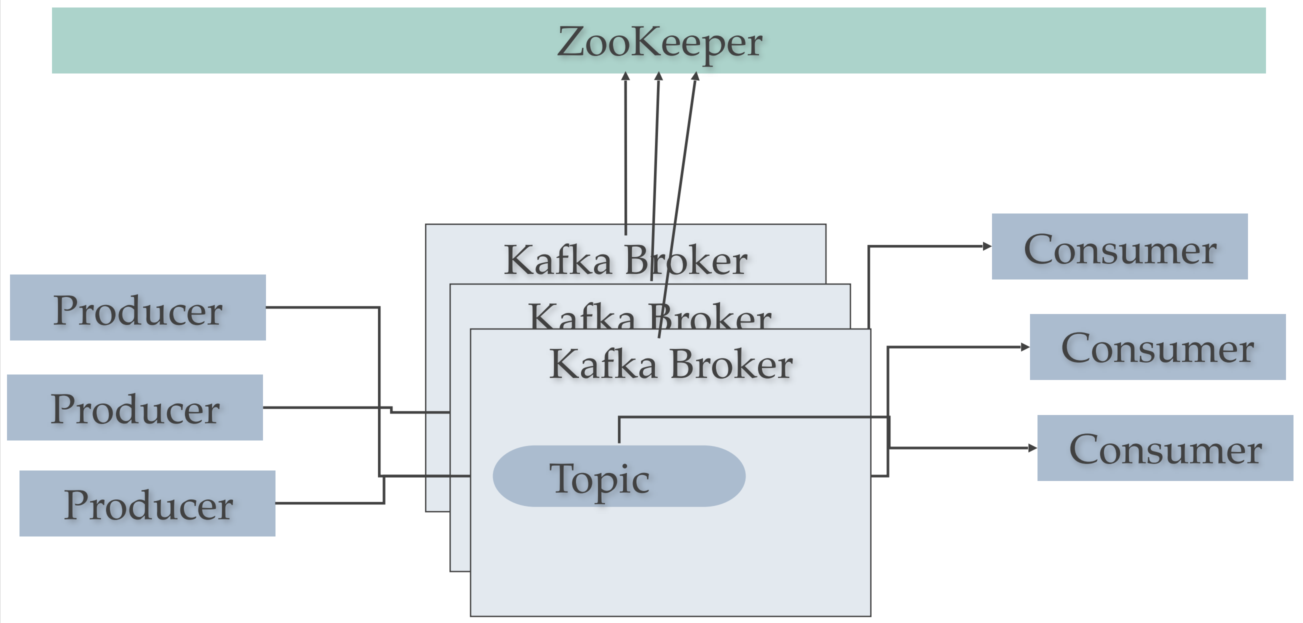
Kafka Cluster is made up of multiple Kafka Brokers.
Each Broker has an ID (number). Kafka Brokers contain topic log partitions. Connecting to one broker bootstraps a client to the entire Kafka cluster. For failover, you want to start with at least three to five brokers. A Kafka cluster can have, 10, 100, or 1,000 brokers in a cluster if needed.

Kafka Cluster, Failover, ISRs

- Topic ***Partitions*** can be ***replicated***
  - across ***multiple nodes*** for failover
- Topic should have a replication factor greater than 1
  - (2, or 3)
- ***Failover***
  - if one Kafka Broker goes down then Kafka Broker with ISR (in-sync replica) can serve data

Kafka supports replication to support failover.
Recall that Kafka uses ZooKeeper to form Kafka Brokers into a cluster and each node in Kafka cluster is called a Kafka Broker.
Topic partitions can be replicated across multiple nodes for failover. The topic should have a replication factor greater than 1 (2, or 3). For example, if you are running in AWS, you would want to be able to survive a single availability zone outage.
If one Kafka Broker goes down, then the Kafka Broker which is an ISR (in-sync replica) can serve data.

ZooKeeper does coordination for Kafka Cluster



Failover vs. Disaster Recovery

- Replication of Kafka Topic Log partitions allows for failure of a rack or AWS availability zone
  - You need a replication factor of at least 3
- ***Kafka Replication*** is for ***Failover***
- ***Mirror Maker*** is used for ***Disaster Recovery***
- Mirror Maker replicates a Kafka cluster to another data-center or AWS region
  - Called mirroring since replication happens within a cluster

Kafka uses replication for failover. Replication of Kafka topic log partitions allows for failure of a rack or AWS availability zone (AZ). You need a replication factor of at least 3 to survive a single AZ failure. You need to use Mirror Maker, a Kafka utility that ships with Kafka core, for disaster recovery. Mirror Maker replicates a Kafka cluster to another data-center or AWS region.
They call what Mirror Maker does mirroring as not to be confused with replication. We cover mirroring and disaster recovery quite a bit.

? Kafka Review

- How does Kafka decouple streams of data?
- What are some use cases for Kafka where you work?
- What are some common use cases for Kafka?
- What is a Topic?
- What is a Broker?
- What is a Partition? Offset?
- Can Kafka run without Zookeeper?
- How do implement failover in Kafka?
- How do you implement disaster recovery in Kafka?

Kafka versus

Kafka vs JMS, SQS, RabbitMQ Messaging

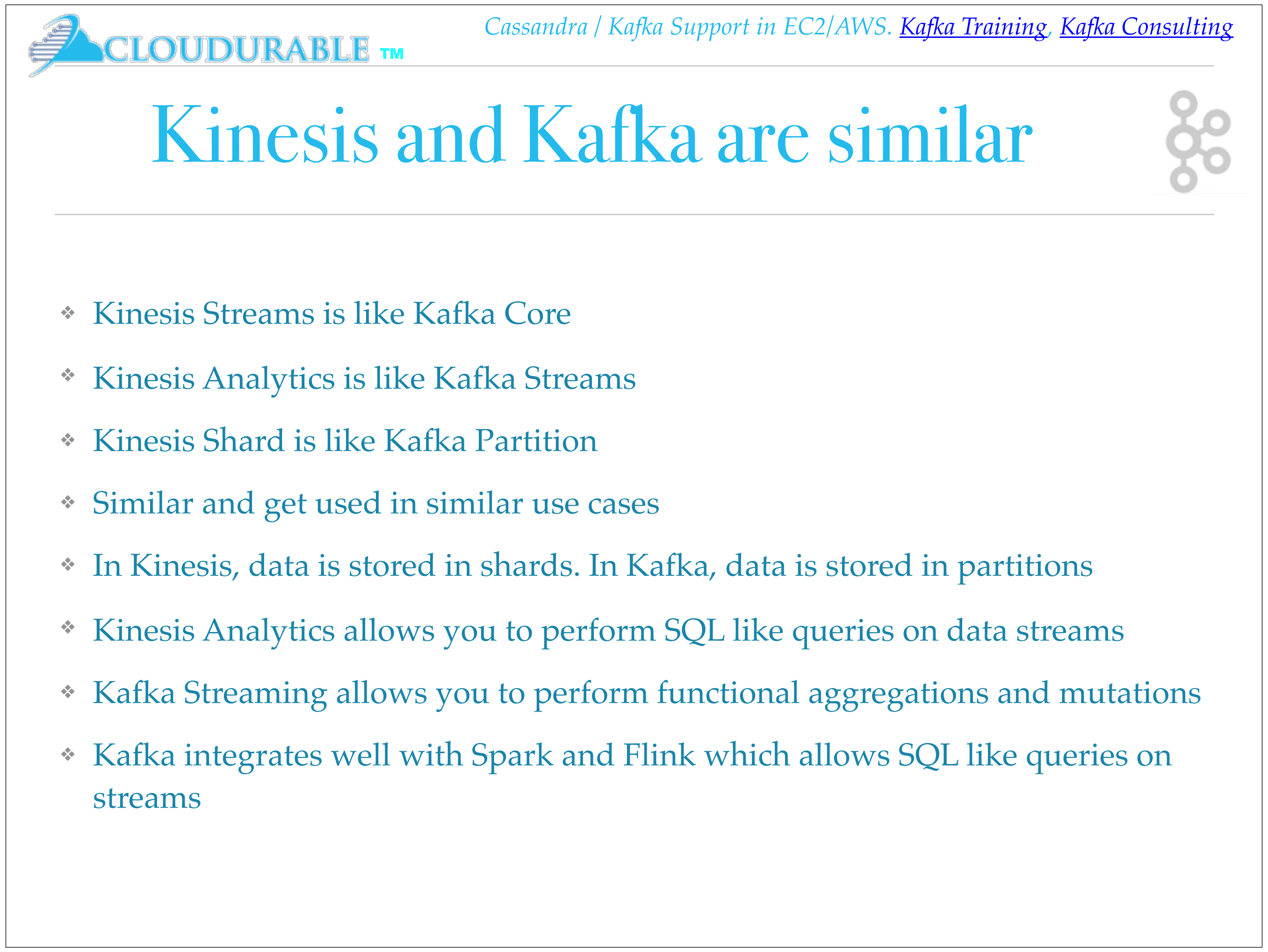
- Is Kafka a Queue or a Pub/Sub/Topic?
  - Yes
- Kafka is like a Queue per consumer group
  - Kafka is a queue system per consumer in consumer group so load balancing like JMS, RabbitMQ queue
- Kafka is like Topics in JMS, RabbitMQ, MOM
  - Topic/pub/sub by offering Consumer Groups which act like subscriptions
  - Broadcast to multiple consumer groups
- MOM = JMS, ActiveMQ, RabbitMQ, IBM MQ Series, Tibco, etc.

Is Kafka a queue or a publish and subscribe system? Yes. It can be both. Kafka is like a queue for consumer groups, which we cover later. Basically, Kafka is a queue system per consumer in consumer group so it can do load balancing like JMS, RabbitMQ, etc. Kafka is like topics in JMS, RabbitMQ, and other MOM systems for multiple consumer groups.
Kafka has topics and producers publish to the topics and the subscribers (Consumer Groups) read from the topics. Kafka offers consumer groups, which is a named group of consumers. A consumer group acts as a subscription. Kafka broadcast to multiple consumer groups using a single queue as each consumer group tracks its own offset into the log. MOM is message oriented middleware, and includes JMS implementors, ActiveMQ, RabbitMQ, IBM MQ Series, and Tibco.

Kafka vs MOM

- By design, Kafka is better suited for scale than traditional MOM systems due to partition topic log
  - Load divided among Consumers for read by partition
  - Handles parallel consumers better than traditional MOM
- Also by moving location (partition offset) in log to client/consumer side of equation instead of the broker, less tracking required by Broker and more flexible consumers
- Kafka written with mechanical sympathy, modern hardware, cloud in mind
  - Disks are faster
  - Servers have tons of system memory
  - Easier to spin up servers for scale out

By design, Kafka is better suited for scale than traditional MOM systems due to partition topic log. Kafka can divide among Consumers by partition and send those message/records in batches. Kafka handles parallel consumers better than traditional MOM, and can even handle failover for consumers in a consumer group. Also, by moving location (partition offset) in log (message queue) to client/consumer side of the equation instead of the broker, less tracking required by Broker and you have more flexible consumers. Kafka was written with mechanical sympathy, modern hardware, cloud in mind: disk drives are faster and bigger, servers have tons of system memory, and it is easier to spin up servers for scale-out then when traditional MOM systems were first designed and built. When IBM MQ Series was first released, your mobile phone would be considered a super computer. Expect messaging systems to catch up and for new messaging systems to come out.



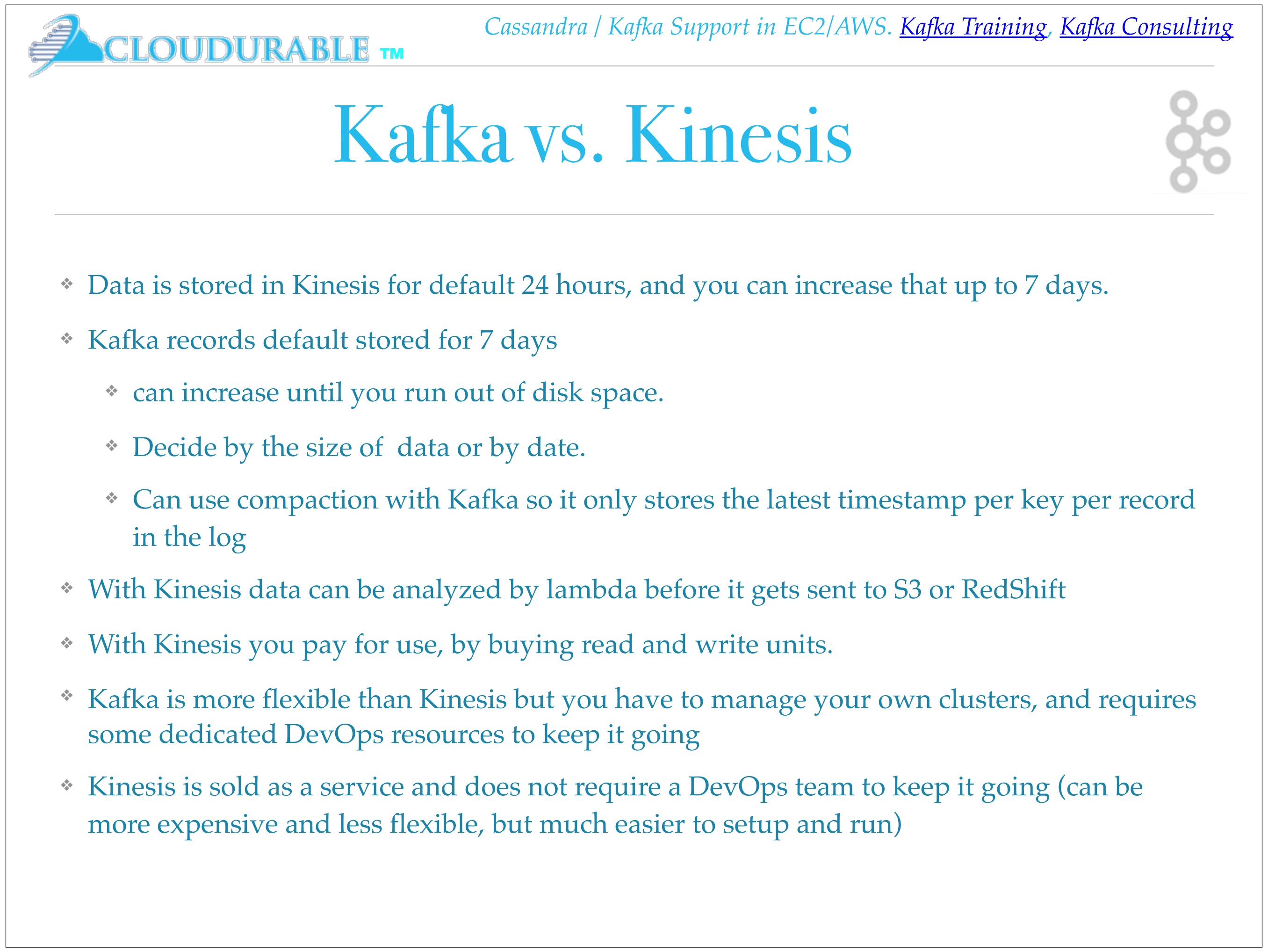
Kinesis and Kafka are similar

- Kinesis Streams is like Kafka Core
- Kinesis Analytics is like Kafka Streams
- Kinesis Shard is like Kafka Partition
- Similar and get used in similar use cases
- In Kinesis, data is stored in shards. In Kafka, data is stored in partitions
- Kinesis Analytics allows you to perform SQL like queries on data streams
- Kafka Streaming allows you to perform functional aggregations and mutations
- Kafka integrates well with Spark and Flink which allows SQL like queries on streams

Kinesis works with streaming data.

- Stock prices
- Game data (scores from game)
- Social network data
- Geospatial data like Uber data where you are
- IOT sensors

Kafka works with streaming data too. Kinesis Streams is like Kafka Core. Kinesis Analytics is like Kafka Streams. A Kinesis Shard is like Kafka Partition. They are similar and get used in similar use cases. In Kinesis, data is stored in shards. In Kafka, data is stored in partitions.



Kafka vs. Kinesis

- Data is stored in Kinesis for default 24 hours, and you can increase that up to 7 days.
- Kafka records default stored for 7 days
    - can increase until you run out of disk space.
    - Decide by the size of data or by date.
    - Can use compaction with Kafka so it only stores the latest timestamp per key per record in the log
- With Kinesis data can be analyzed by lambda before it gets sent to S3 or RedShift
- With Kinesis you pay for use, by buying read and write units.
- Kafka is more flexible than Kinesis but you have to manage your own clusters, and requires some dedicated DevOps resources to keep it going
- Kinesis is sold as a service and does not require a DevOps team to keep it going (can be more expensive and less flexible, but much easier to setup and run)

Data is stored in Kinesis for default 24 hours, and you can increase that up to 7 days.
Kafka records are by default stored for 7 days and you can increase that until you run out of disk space. In fact, you can decide by the size of the data or by date. You can even use compaction with Kafka so it only stores the latest timestamp per key per record in the log.

With Kinesis data can be analyzed by lambda before it gets sent to S3 or RedShift.
With Kinesis you pay for use, by buying read and write units.
Kinesis Analytics allows you to perform SQL like queries on data. Kafka Streaming allows you to perform functional aggregations and mutations. Kafka integrates with Flink and Spark which allow querying streams with SQL like queries.

Kafka is more flexible than Kinesis but you have to manage your own clusters, and requires some dedicated DevOps resources to keep it going. Kinesis is sold as a service and does not require a DevOps team to keep it going. You pay for this. Depending on the use case, Kinesis could be faster to get started and much easier than hiring staff to manage a Kafka cluster. However, with less flexibility and potentially the expense. It really depends on your use case and volume of data on which is the better fit.

Kafka Topics Architecture

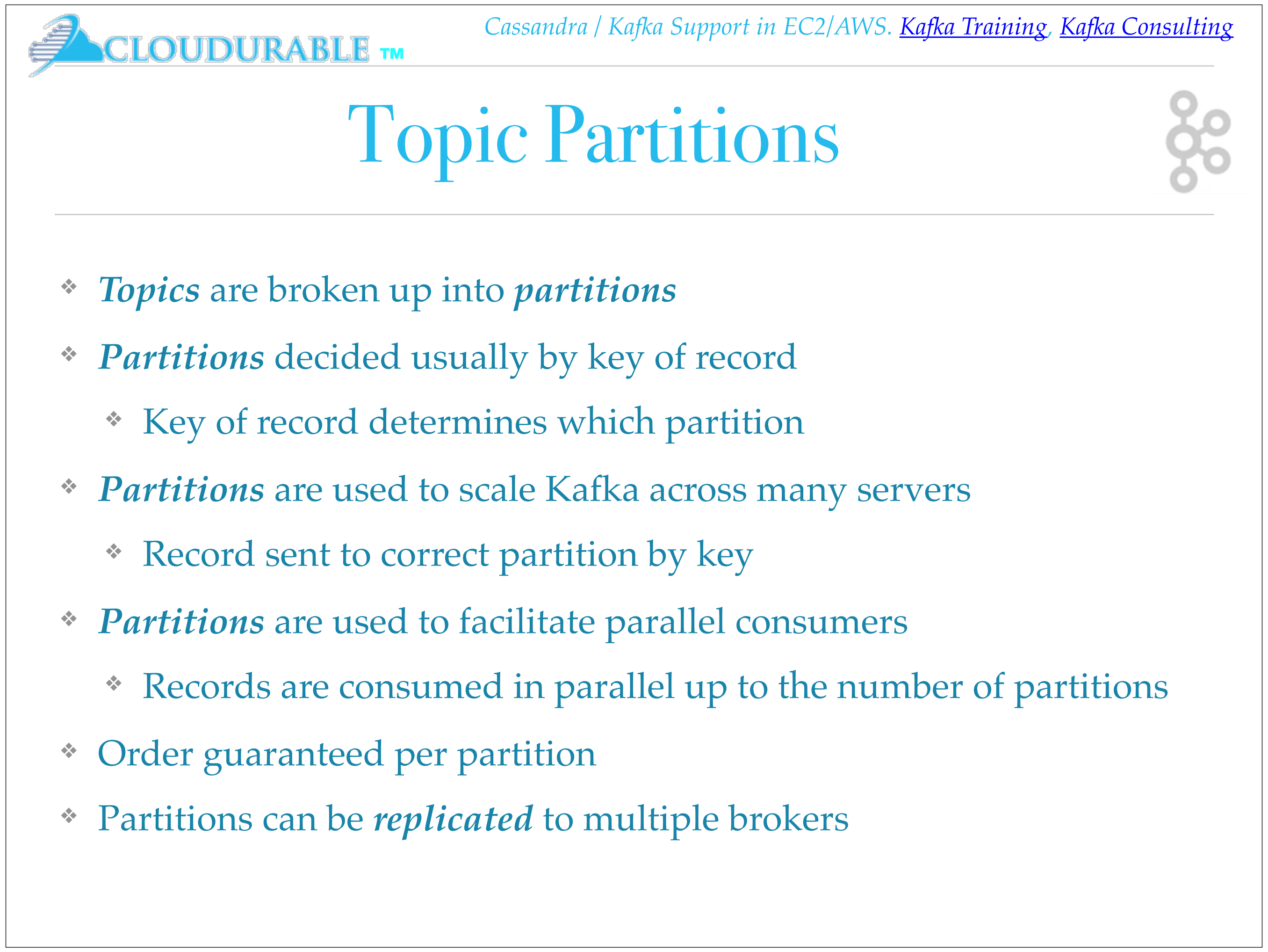
Replication
Failover
Parallel processing
Kafka Topic Architecture

Topics, Logs, Partitions

- Kafka ***Topic*** is a stream of records
- ***Topics*** stored in log
- ***Log*** broken up into ***partitions*** and ***segments***
- ***Topic*** is a category or stream name or feed
- Topics are pub/sub
  - Can have zero or many subscribers - consumer groups
- ***Topics*** are broken up and spread by partitions for speed and size

Recall that a Kafka topic is a named stream of records.
Kafka stores topics in logs. A topic log is broken up into partitions.
Kafka spreads log's partitions across multiple servers or disks. Think of a topic as a category, stream name or feed.

Topics are inherently published and
subscribe style messaging. A Topic can have zero or many subscribers called consumer groups. Topics are broken up into partitions for speed, scalability, and size.



Cassandra / Kafka Support in EC2/AWS. Kafka Training, Kafka Consulting

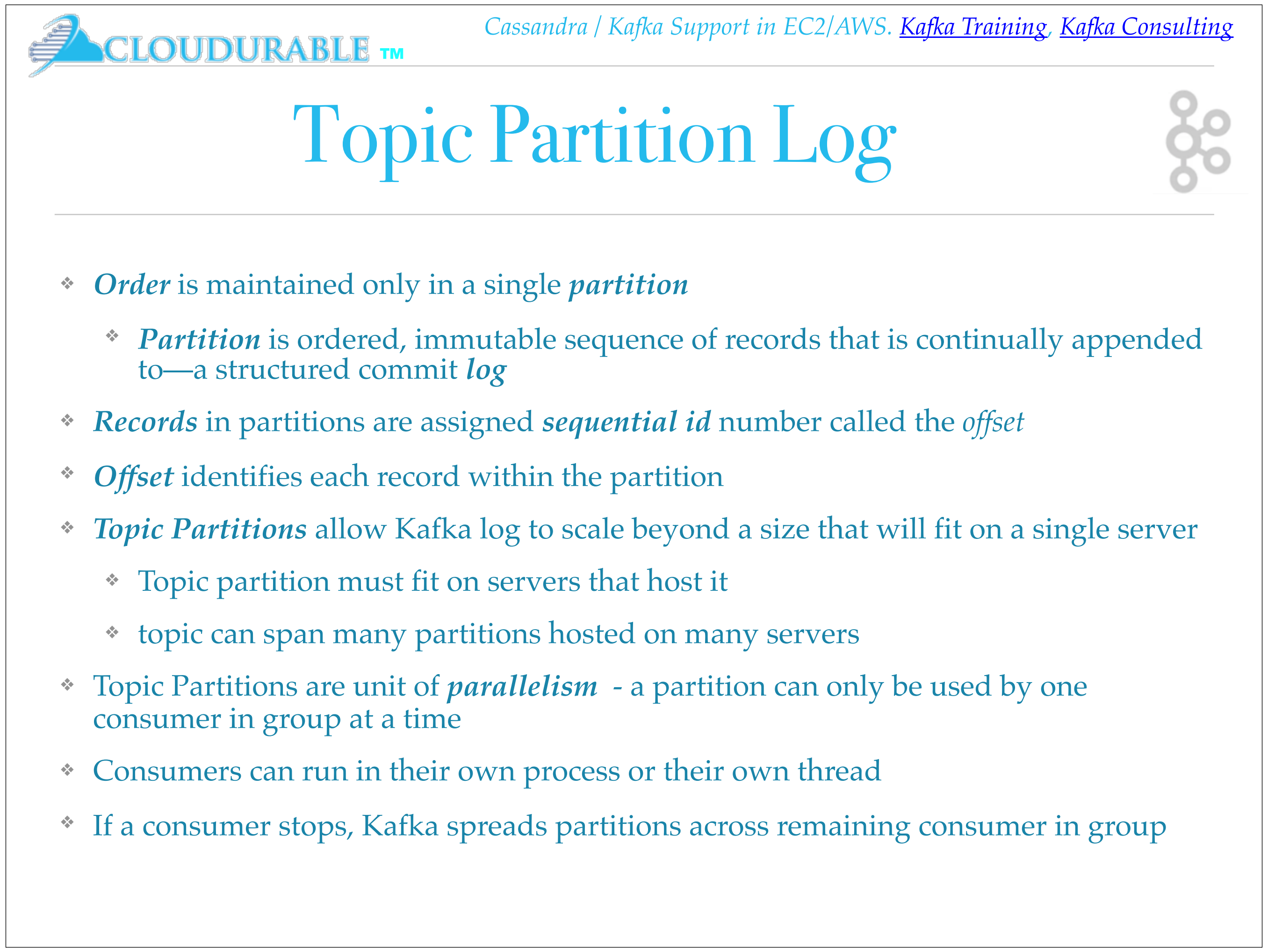
Topic Partitions

- ***Topics*** are broken up into ***partitions***
- ***Partitions*** decided usually by key of record
  - Key of record determines which partition
- ***Partitions*** are used to scale Kafka across many servers
  - Record sent to correct partition by key
- ***Partitions*** are used to facilitate parallel consumers
  - Records are consumed in parallel up to the number of partitions
- Order guaranteed per partition
- Partitions can be ***replicated*** to multiple brokers

Kafka breaks topics up into partitions. A record is stored on a partition usually by record key if the key is present and round-robin if the key is missing (default behavior). The record key, by default, determines which partition a producer sends the record.

Kafka uses partitions to scale a topic across many servers for producer writes. Also, Kafka also uses partitions to facilitate parallel consumers. Consumers consume records in parallel up to the number of partitions.

The order guaranteed per partition. If partitioning by key then all records for the key will be on the same partition which is useful if you ever have to replay the log. Kafka can replicate partitions to multiple brokers for failover.

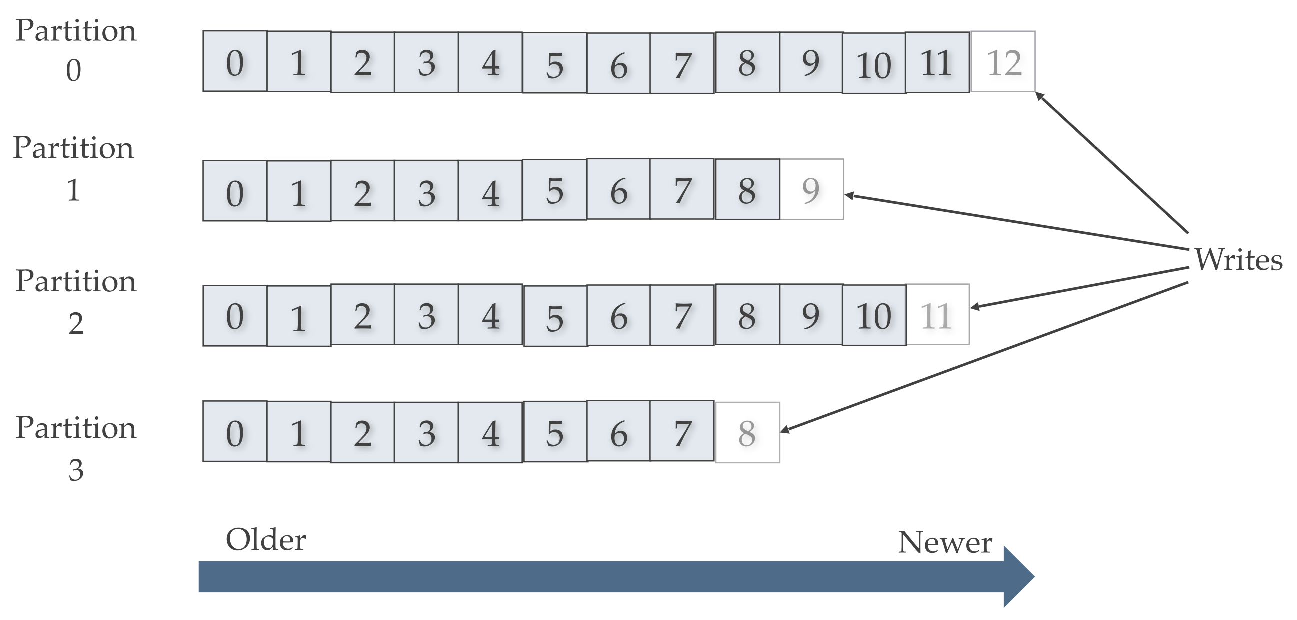


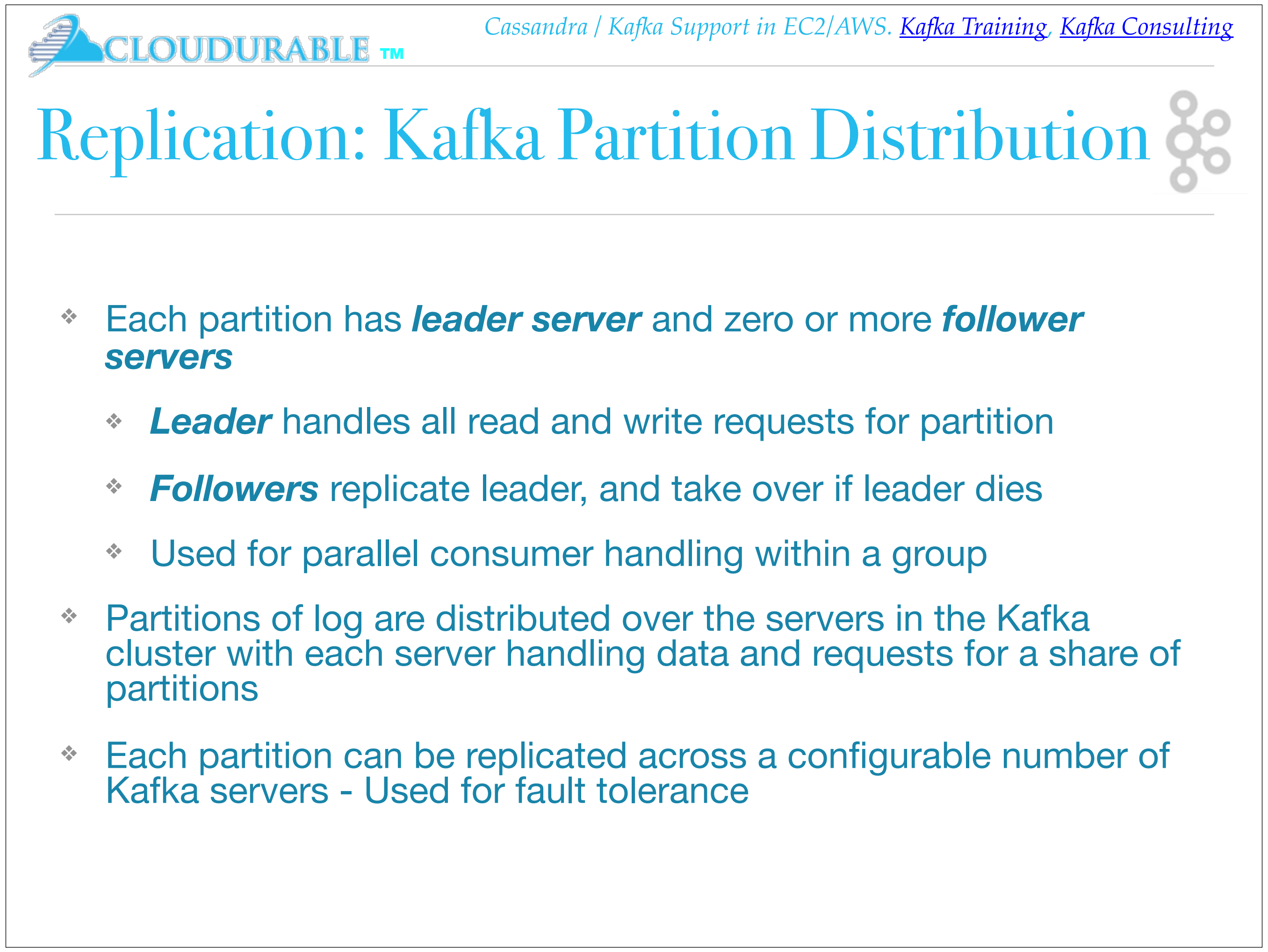
Topic Partition Log

- ***Order*** is maintained only in a single ***partition***
    - ***Partition*** is ordered, immutable sequence of records that is continually appended to—a structured commit ***log***
- ***Records*** in partitions are assigned ***sequential id*** number called the *offset*
- ***Offset*** identifies each record within the partition
- ***Topic Partitions*** allow Kafka log to scale beyond a size that will fit on a single server
    - Topic partition must fit on servers that host it
    - topic can span many partitions hosted on many servers
- Topic Partitions are unit of ***parallelism*** - a partition can only be used by one consumer in group at a time
- Consumers can run in their own process or their own thread
- If a consumer stops, Kafka spreads partitions across remaining consumer in group

Kafka maintains record order only in a single partition. A partition is an ordered, immutable record sequence. Kafka continually appended to partitions using the partition as a structured commit log. Records in partitions are assigned sequential id number called the offset. The offset identifies each record location within the partition. Topic partitions allow Kafka log to scale beyond a size that will fit on a single server. Topic partitions must fit on servers that host it, but topics can span many partitions hosted on many servers. Also, topic partitions are a unit of parallelism - a partition can only be worked on by one consumer in a consumer group at a time. Consumers can run in their own process or their own thread. If a consumer stops, Kafka spreads partitions across the remaining consumer in the same consumer group.

Kafka Topic Partitions Layout





Replication: Kafka Partition Distribution

- Each partition has ***leader server*** and zero or more ***follower servers***
  - ***Leader*** handles all read and write requests for partition
  - ***Followers*** replicate leader, and take over if leader dies
  - Used for parallel consumer handling within a group
- Partitions of log are distributed over the servers in the Kafka cluster with each server handling data and requests for a share of partitions
- Each partition can be replicated across a configurable number of Kafka servers - Used for fault tolerance

Each partition can be replicated across a configurable number of Kafka servers - used for fault tolerance. Each partition has a leader server and zero or more follower servers. Leaders handle all read and write requests for a partition. Followers replicate leaders and take over if the leader dies. Kafka uses also uses partitions for parallel consumer handling within a group. Kafka distributes topic log partitions over servers in the Kafka cluster. Each server handles its share of data and requests by sharing partition leadership.

Replication: Kafka Partition Leader

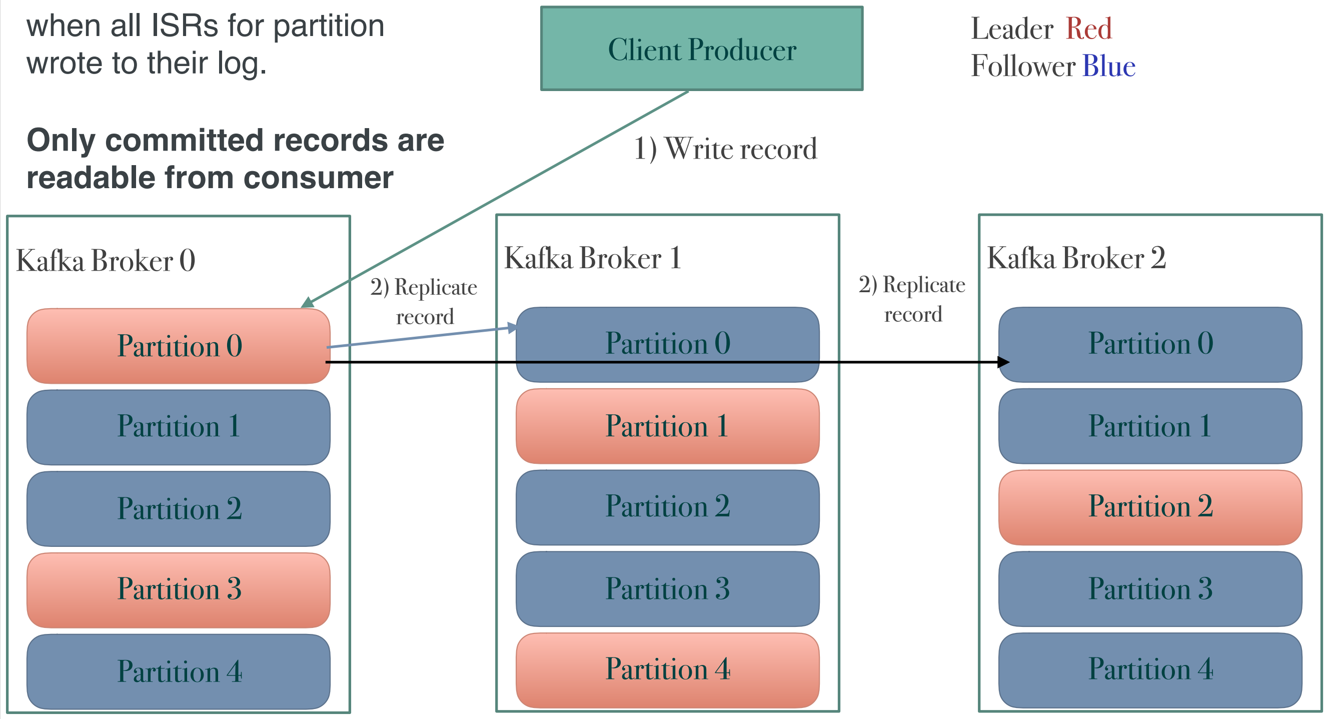
- One node/partition's replicas is chosen as ***leader***
- Leader handles all reads and writes of Records for partition
- Writes to partition are ***replicated*** to ***followers*** (node/partition pair)
- An ***follower*** that is ***in-sync*** is called an ***ISR (in-sync replica)***
- If a partition leader fails, one ISR is chosen as new leader

Kafka chooses one broker's partition's replicas as leader using ZooKeeper. The broker that has the partition leader handles all reads and writes of records for the partition. Kafka replicates writes to the leader partition to followers (node/partition pair). A follower that is in-sync is called an ISR (in-sync replica). If a partition leader fails, Kafka chooses a new ISR as the new leader.

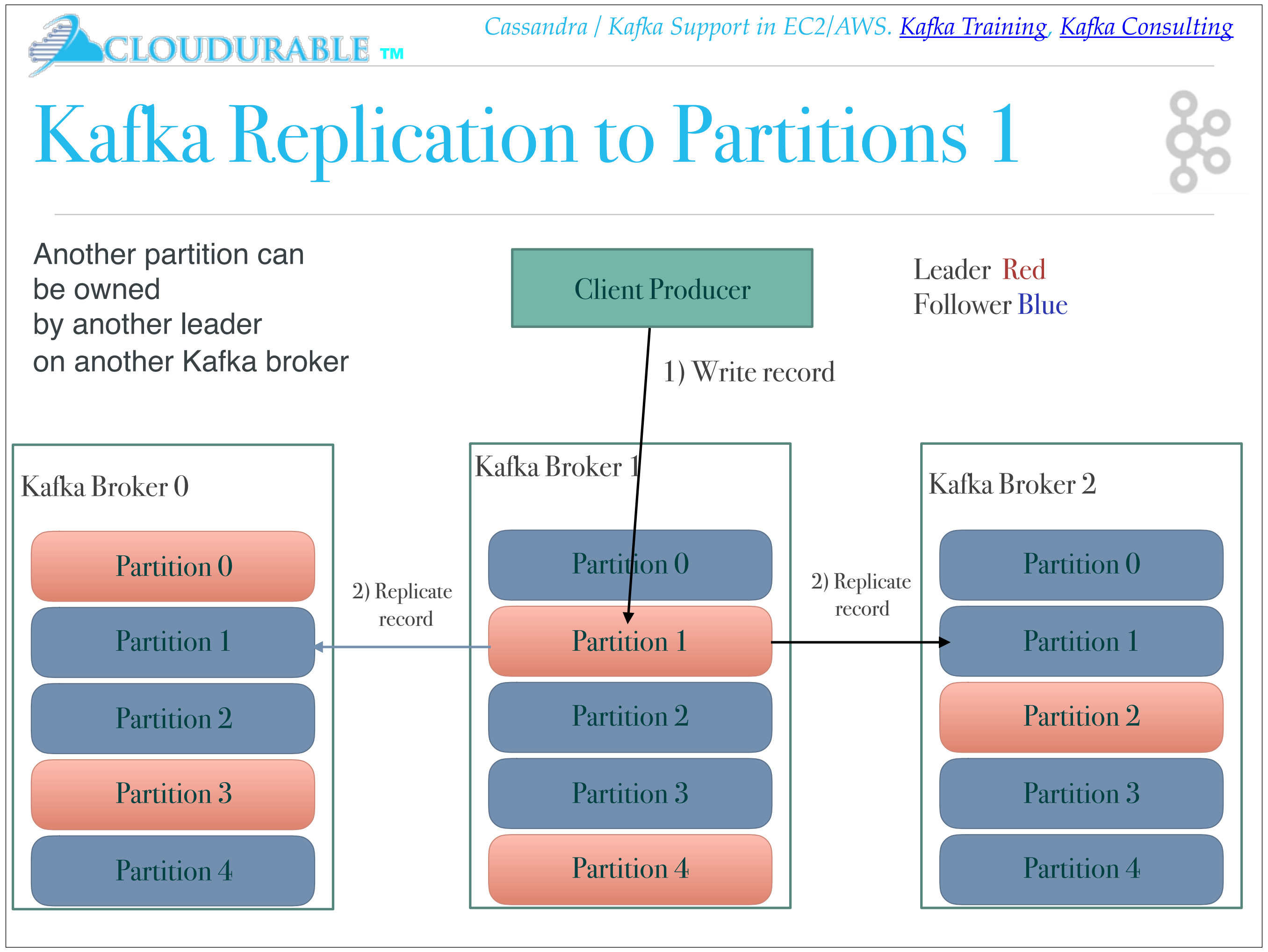
Kafka Replication to Partition 0

Record is considered "committed" when all ISRs for partition wrote to their log.

Only committed records are readable from consumer



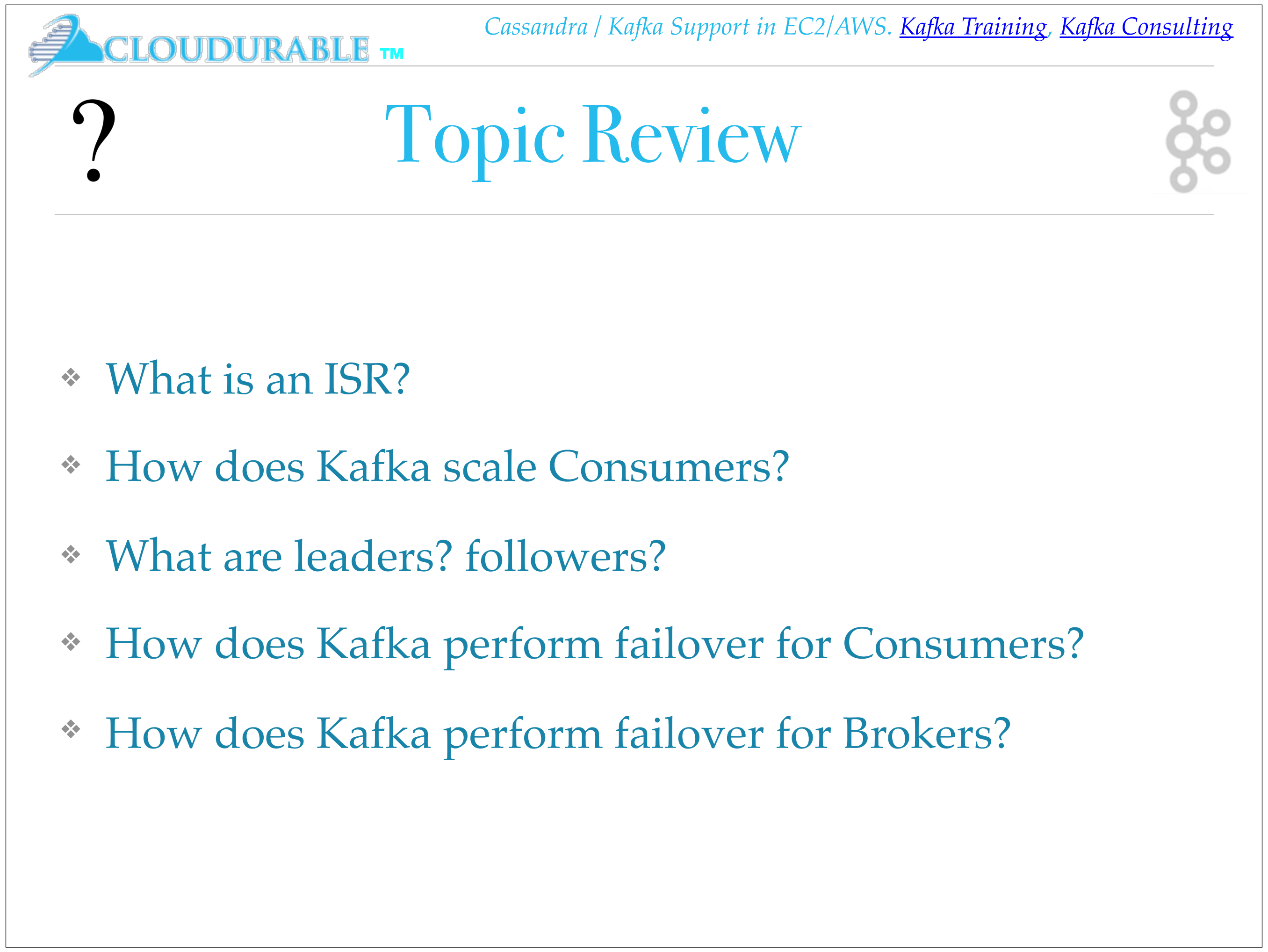
The record is considered "committed" when all ISRs for partition wrote to their log.
Only committed records are readable from consumer.



Kafka Replication to Partitions 1

Another partition can
be owned
by another leader
on another Kafka broker

Another partition can be owned by another leader on another Kafka broker.



? Topic Review

- What is an ISR?
- How does Kafka scale Consumers?
- What are leaders? followers?
- How does Kafka perform failover for Consumers?
- How does Kafka perform failover for Brokers?

What is an ISR?

An ISR is an in-sync replica. If a leader fails, an ISR is picked to be a new leader.

How does Kafka scale consumers?

Kafka scales consumers by partition such that each consumer gets its share of partitions. A consumer can have more than one partition, but a partition can only be used by one consumer in a consumer group at a time. If you only have one partition, then you can only have one consumer.

What are leaders? Followers?

Leaders perform all reads and writes to a particular topic partition. Followers replicate leaders.

How does Kafka perform failover for consumers?

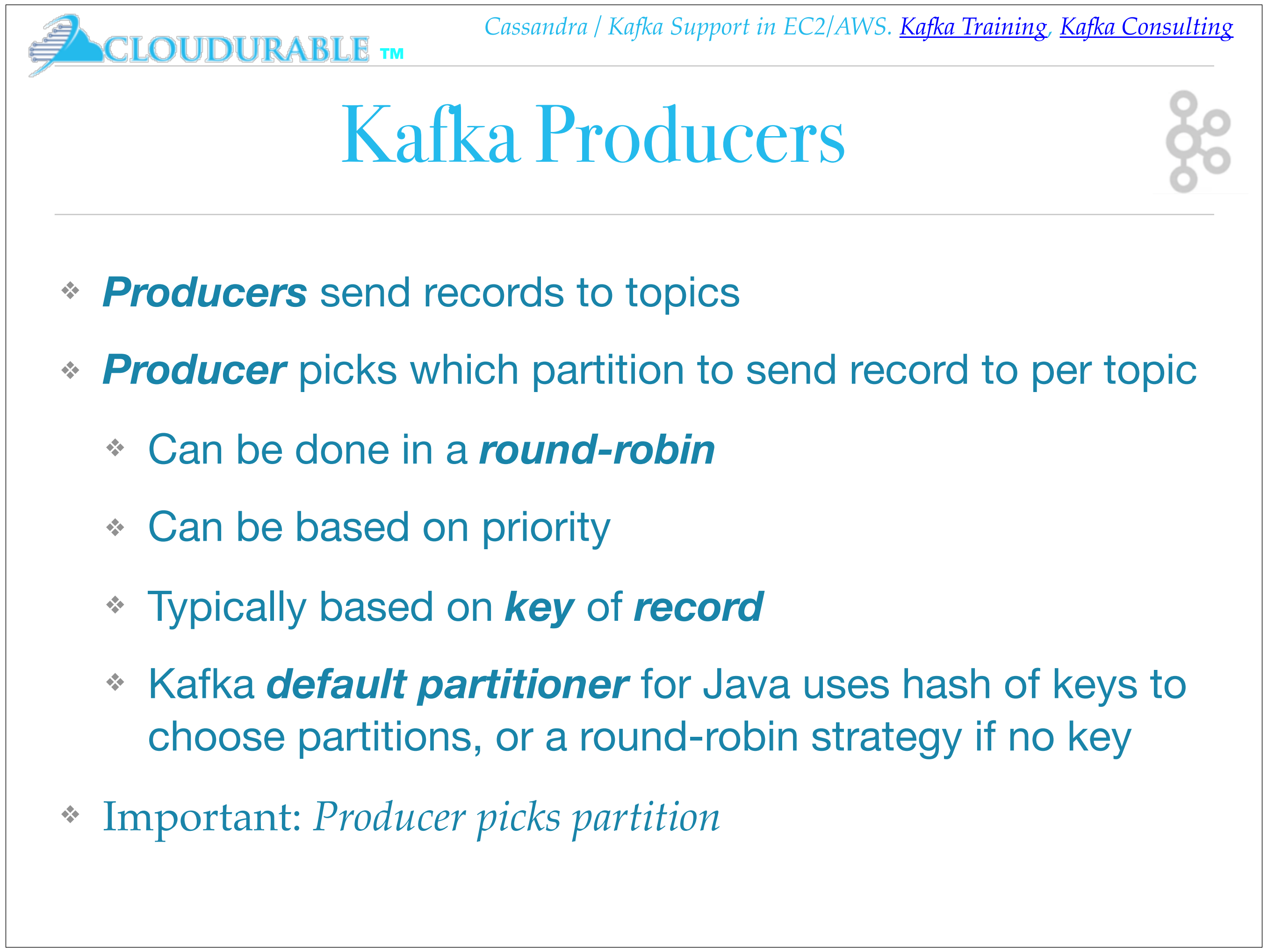
If a consumer in a consumer group dies, the partitions assigned to that consumer is divided up amongst the remaining consumers in that group.

How does Kafka perform failover for Brokers?

If a broker dies, then Kafka divides up leadership of its topic partitions to the remaining brokers in the cluster.

Kafka Producers

Partition selection
Durability
Kafka Producer Architecture



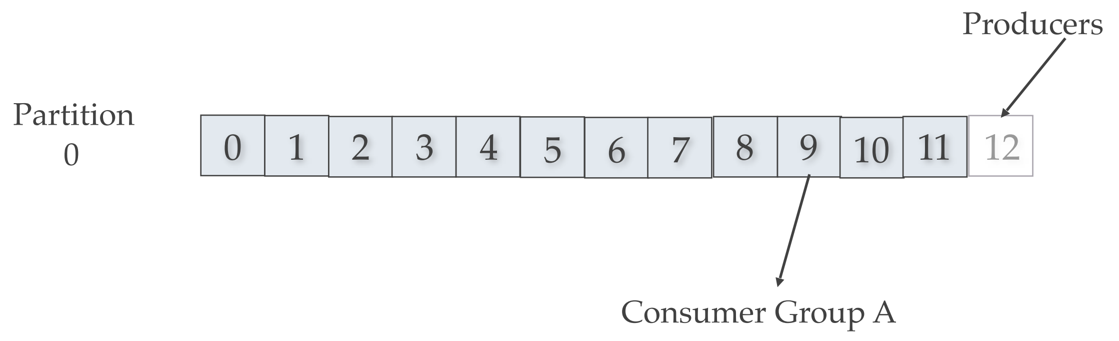
Kafka Producers

- ***Producers*** send records to topics
- ***Producer*** picks which partition to send record to per topic
  - Can be done in a ***round-robin***
  - Can be based on priority
  - Typically based on ***key*** of ***record***
  - Kafka ***default partitioner*** for Java uses hash of keys to choose partitions, or a round-robin strategy if no key
- Important: *Producer picks partition*

Kafka producers send records to topics. The records are sometimes referred to as messages. The producer picks which partition to send a record to per topic. The producer can send records round-robin. The producer could implement priority systems based on sending records to certain partitions based on the priority of the record.

Generally speaking, producers send records to a partition based on the record's key. The default partitioner for Java uses a hash of the record's key to choose the partition or uses a round-robin strategy if the record has no key. The important concept here is that the producer picks partition.

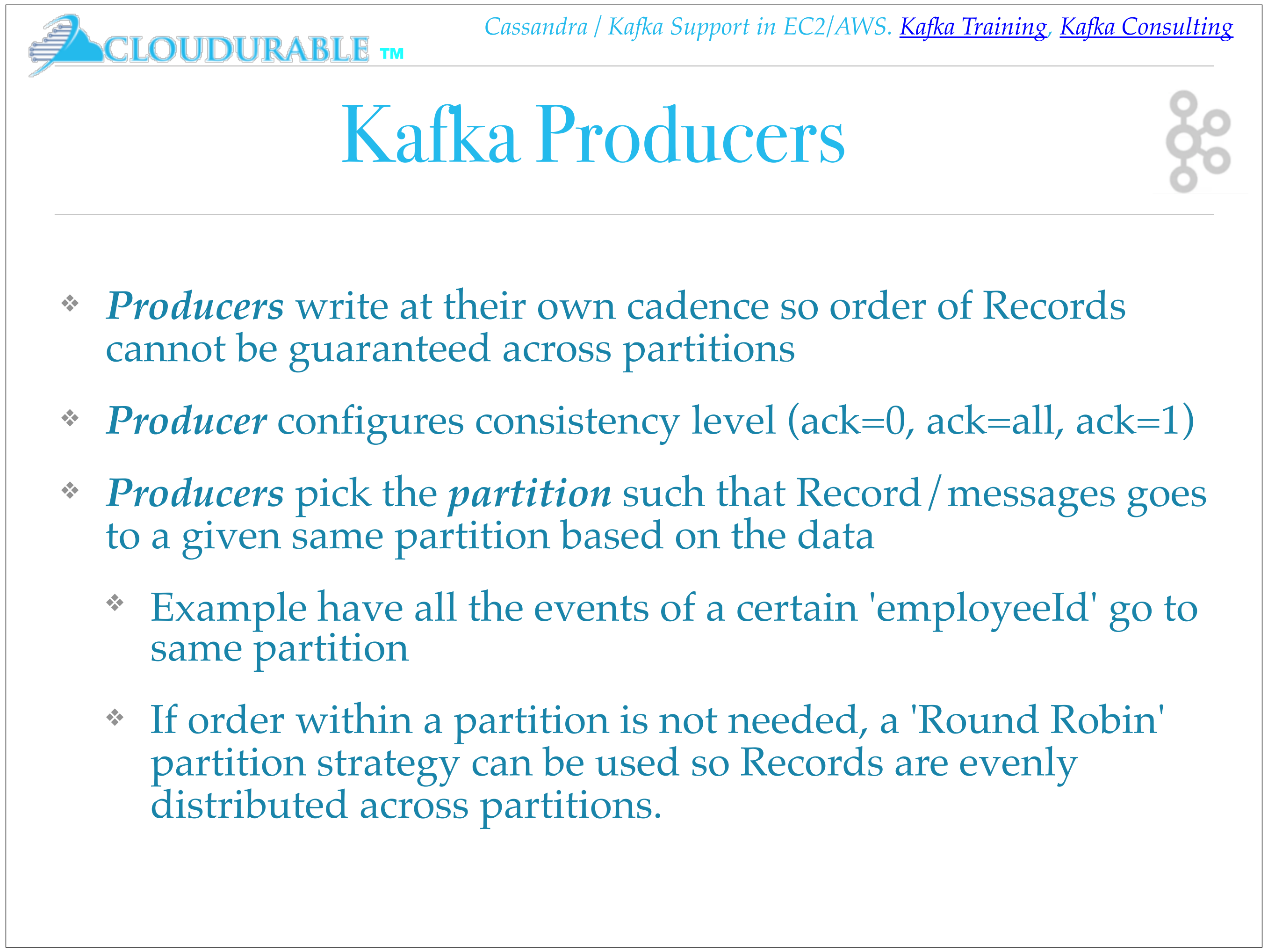
Kafka Producers and Consumers



Producers are writing at Offset 12

Consumer Group A is Reading from Offset 9.

Producers are writing at Offset 12
Consumer Group A is Reading from Offset 9.



Kafka Producers

- ***Producers*** write at their own cadence so order of Records cannot be guaranteed across partitions
- ***Producer*** configures consistency level (ack=0, ack=all, ack=1)
- ***Producers*** pick the ***partition*** such that Record / messages goes to a given same partition based on the data
  - Example have all the events of a certain 'employeeId' go to same partition
  - If order within a partition is not needed, a 'Round Robin' partition strategy can be used so Records are evenly distributed across partitions.

We will cover acks in more detail later.

Producers write at their cadence so the order of Records cannot be guaranteed across partitions. The producers get to configure their consistency/durability level (ack=0, ack=all, ack=1), which we will cover later. Producers pick the partition such that Record/messages go to a given partition based on the data. For example, you could have all the events of a certain 'employeeId' go to the same partition. If order within a partition is not needed, a 'Round Robin' partition strategy can be used, so Records get evenly distributed across partitions.

? Producer Review

- Can Producers occasionally write faster than consumers?
- What is the default partition strategy for Producers without using a key?
- What is the default partition strategy for Producers using a key?
- What picks which partition a record is sent to?

Can producers occasionally write faster than consumers?

Yes. A producer could have a burst of records, and a consumer does not have to be on the same page as the consumer.

What is the default partition strategy for producers without using a key?

Round-Robin

What is the default partition strategy for Producers using a key?

Records with the same key get sent to the same partition.

What picks which partition a record is sent to?

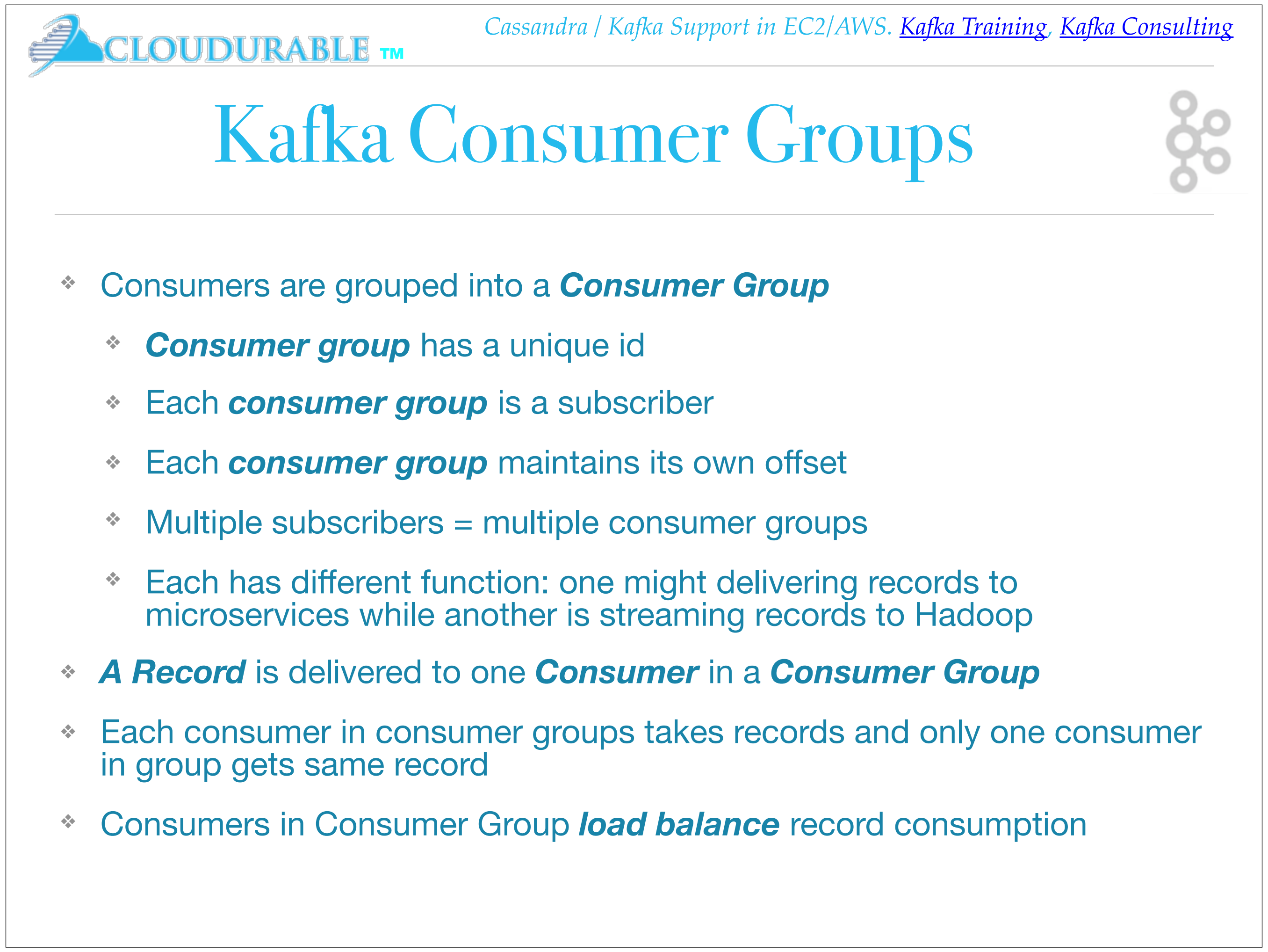
The Producer picks which partition a record goes to.

Kafka Consumers

Load balancing consumers
Failover for consumers

Offset management per consumer group

Kafka Consumer Architecture



Kafka Consumer Groups

- Consumers are grouped into a ***Consumer Group***
  - ***Consumer group*** has a unique id
  - Each ***consumer group*** is a subscriber
  - Each ***consumer group*** maintains its own offset
  - Multiple subscribers = multiple consumer groups
  - Each has different function: one might delivering records to microservices while another is streaming records to Hadoop
- ***A Record*** is delivered to one ***Consumer*** in a ***Consumer Group***
- Each consumer in consumer groups takes records and only one consumer in group gets same record
- Consumers in Consumer Group ***load balance*** record consumption

You group consumers into a consumer group by use case or function of the group. One consumer group might be responsible for delivering records to high-speed, in-memory microservices while another consumer group is streaming those same records to Hadoop. Consumer groups have names to identify them from other consumer groups.

A consumer group has a unique id. Each consumer group is a subscriber to one or more Kafka topics. Each consumer group maintains its offset per topic partition.
If you need multiple subscribers, then you have multiple consumer groups. A record gets delivered to only one consumer in a consumer group.

Each consumer in a consumer group processes records and only one consumer in that group will get the same record.
Consumers in a consumer group load balance record consumption.

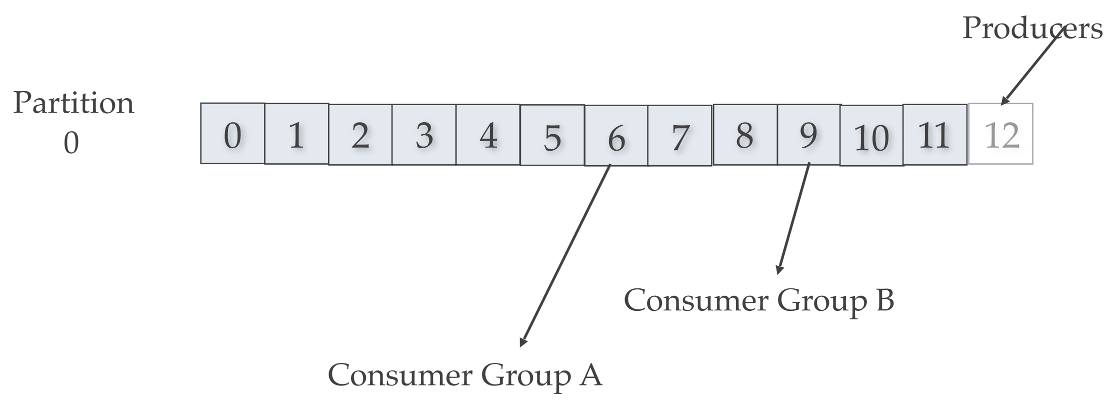
Recall that order is only guaranteed within a single partition. Since records are typically stored by key into a partition then order per partition is sufficient for most use cases

Kafka Consumer Load Share

- Kafka ***Consumer*** consumption ***divides*** partitions over consumers in a Consumer Group
- Each ***Consumer*** is exclusive consumer of a **"*fair share*" *of partitions***
- This is Load Balancing
- ***Consumer*** membership in ***Consumer Group*** is handled by the Kafka protocol dynamically
- If new Consumers ***join*** Consumer group, it gets a share of partitions
- If Consumer ***dies***, its partitions are split among remaining live Consumers in Consumer Group

Kafka consumer consumption divides partitions over consumer instances within a consumer group.
Each consumer in the consumer group is an exclusive consumer of a "fair share" of partitions.
This is how Kafka does load balancing of consumers in a consumer group.
Consumer membership within a consumer group is handled by the Kafka protocol dynamically.
If new consumers join a consumer group, it gets a share of partitions. If a
consumer dies, its partitions are split among the remaining live consumers in the consumer group.
This is how Kafka does fail over of consumers in a consumer group.

Kafka Consumer Groups



Consumers remember offset where they left off.

Consumers groups each have their own offset per partition.

Consumers remember offset where they left off.
Consumers groups each have their own offset per partition.

Kafka Consumer Groups Processing

- How does Kafka divide up topic so multiple ***Consumers*** in a ***Consumer Group*** can process a topic?
- You group consumers into consumers group with a group id
- ***Consumers*** with same id belong in same ***Consumer Group***
- One ***Kafka broker*** becomes ***group coordinator*** for Consumer Group
  - assigns partitions when new members arrive (older clients would talk direct to ZooKeeper now broker does coordination)
  - or reassign partitions when group members leave or topic changes (config / meta-data change
- When ***Consumer group*** is created, offset set according to reset policy of topic

How does Kafka divide up a topic so multiple consumers in a consumer group can process a topic in parallel? Kafka makes you group consumers into consumer groups with a group id. Consumers with the same id are in the same consumer group.

A single Kafka broker becomes the group coordinator for the consumer group. This broker assigns partitions when new members arrive note that older Kafka clients would talk directly to ZooKeeper now broker does coordination and uses ZooKeeper in the background. This broker can reassign partitions when group members leave or if the topic configuration changes. When the consumer group is created, the offset is set according to reset policy of topic.

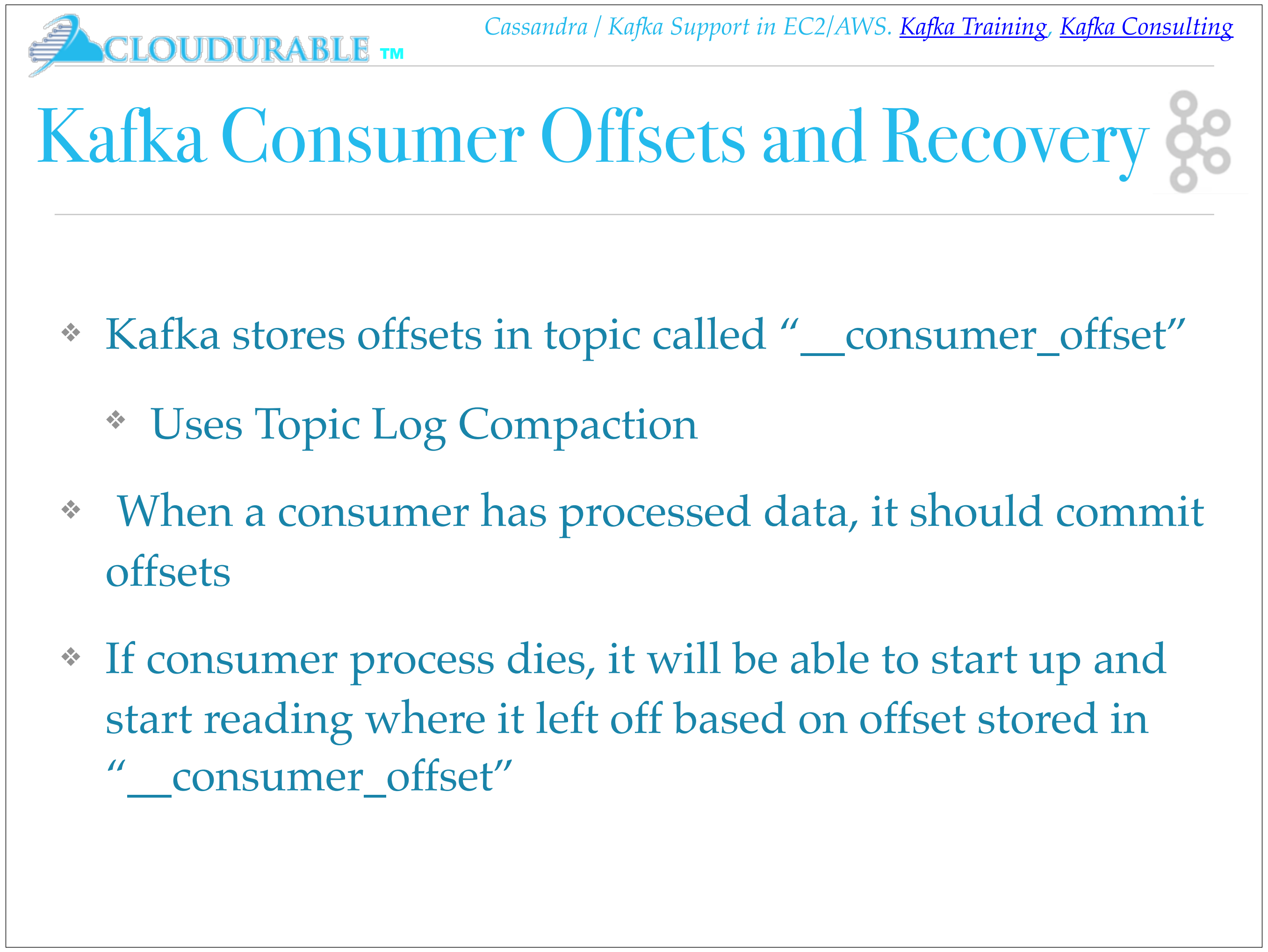
Kafka Consumer Failover

- ***Consumers*** notify broker when it successfully processed a record
  - advances offset
- If ***Consumer*** fails before sending commit offset to Kafka broker,
  - different ***Consumer*** can continue from the last committed offset
  - some Kafka records could be reprocessed
  - ***at least once behavior***
  - ***messages should be idempotent***

Consumers notify the Kafka broker when they have successfully processed a record, which advances the offset.

If a consumer fails before sending commit offset to Kafka broker, then a different consumer can continue from the last committed offset.

If a consumer fails after processing the record but before sending the commit to the broker, then some Kafka records could be reprocessed. In this scenario, Kafka implements the at least once behavior, and you should make sure the messages (record deliveries) are idempotent.



Kafka Consumer Offsets and Recovery

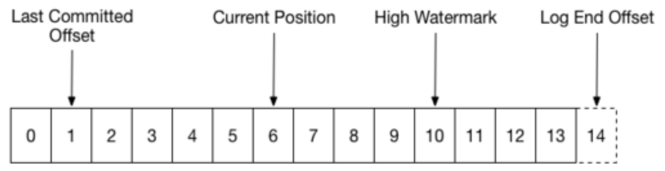
- Kafka stores offsets in topic called “__consumer_offset”
    - Uses Topic Log Compaction
- When a consumer has processed data, it should commit offsets
- If consumer process dies, it will be able to start up and start reading where it left off based on offset stored in “__consumer_offset”

Kafka stores offset data in a topic called "__consumer_offset".
These topics use log compaction, which means they only save the most recent value per key.

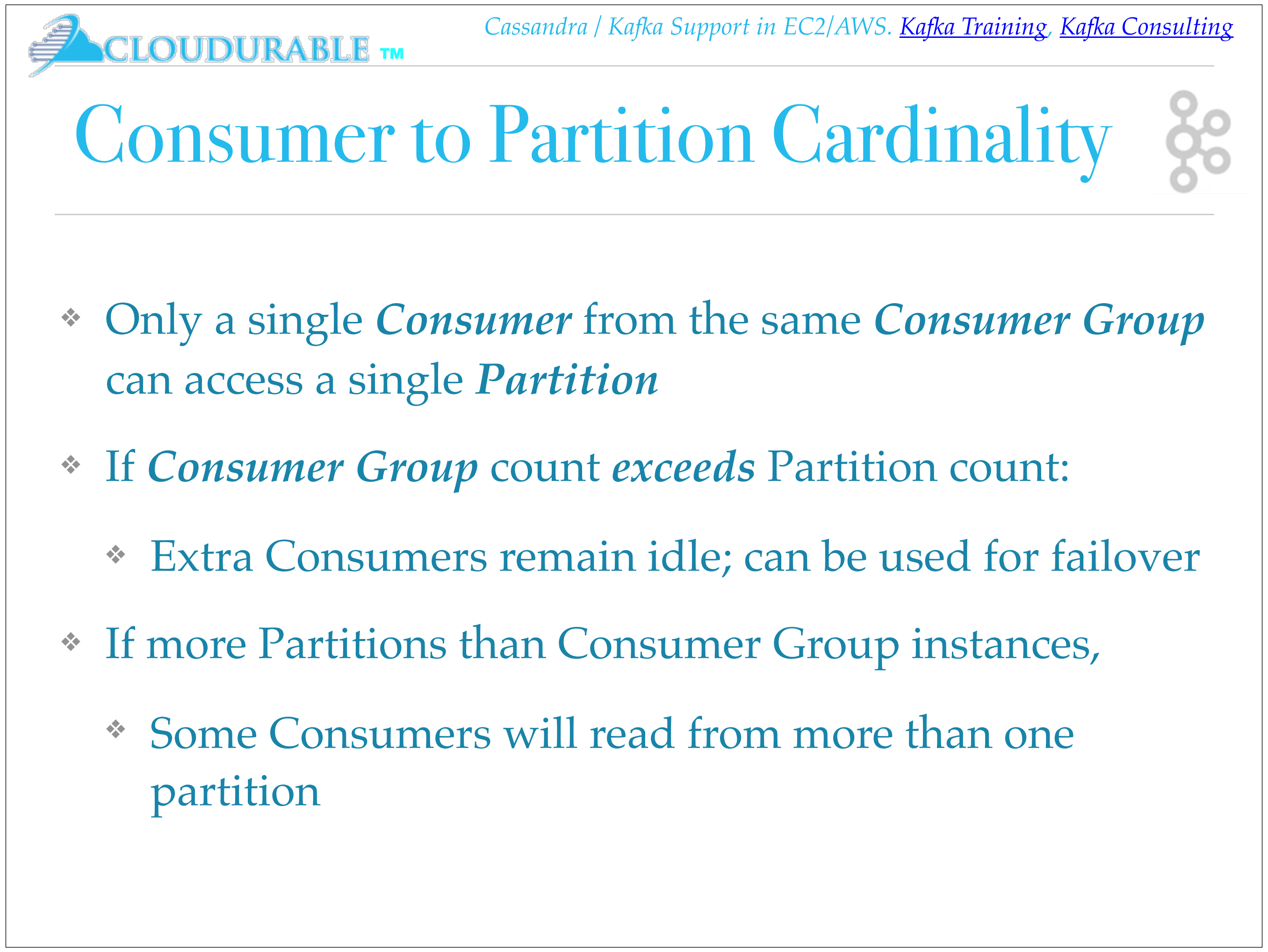
When a consumer has processed data, it should commit offsets.
If consumer process dies, it will be able to start up and start reading where it left off based on offset stored in "__consumer_offset" or as discussed another consumer in the group can take over.

Kafka Consumer: 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.

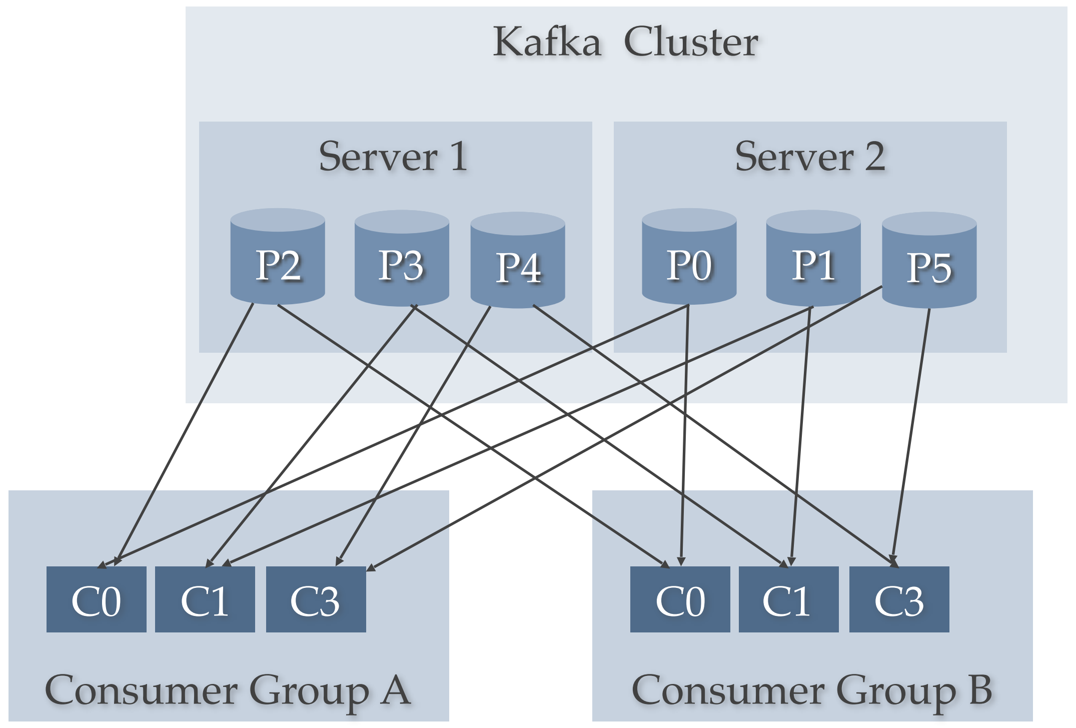


Consumer to Partition Cardinality

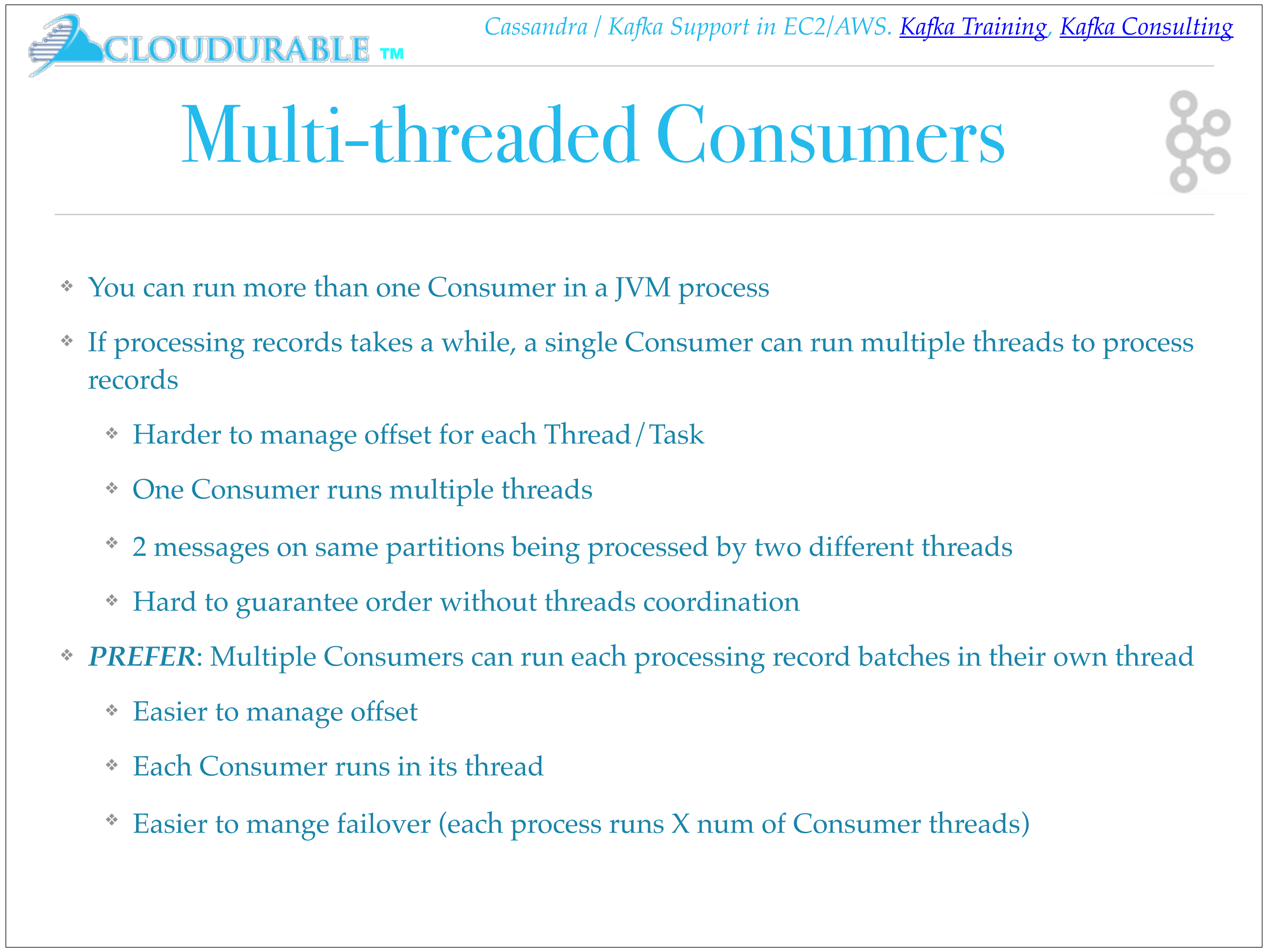
- Only a single ***Consumer*** from the same ***Consumer Group*** can access a single ***Partition***
- If ***Consumer Group*** count ***exceeds*** Partition count:
  - Extra Consumers remain idle; can be used for failover
- If more Partitions than Consumer Group instances,
  - Some Consumers will read from more than one partition

Only a single consumer from the same consumer group can access a single partition. If consumer group count exceeds the partition count, then the extra consumers remain idle. Kafka can use the idle consumers for failover.
If there are more partitions than consumer group, then
some consumers will read from more than one partition.

2 server Kafka cluster hosting 4 partitions (P0-P5)



Notice server 1 has topic partition P2, P3, and P4 while server 2 has partition P0, P1, and P5.
Notice that Consumer C0 from Consumer Group A is processing records from P0 and P2.
Notice that no single partition is shared by any consumer from any consumer group.
Notice that each partition gets its fair share of partitions for the topics.



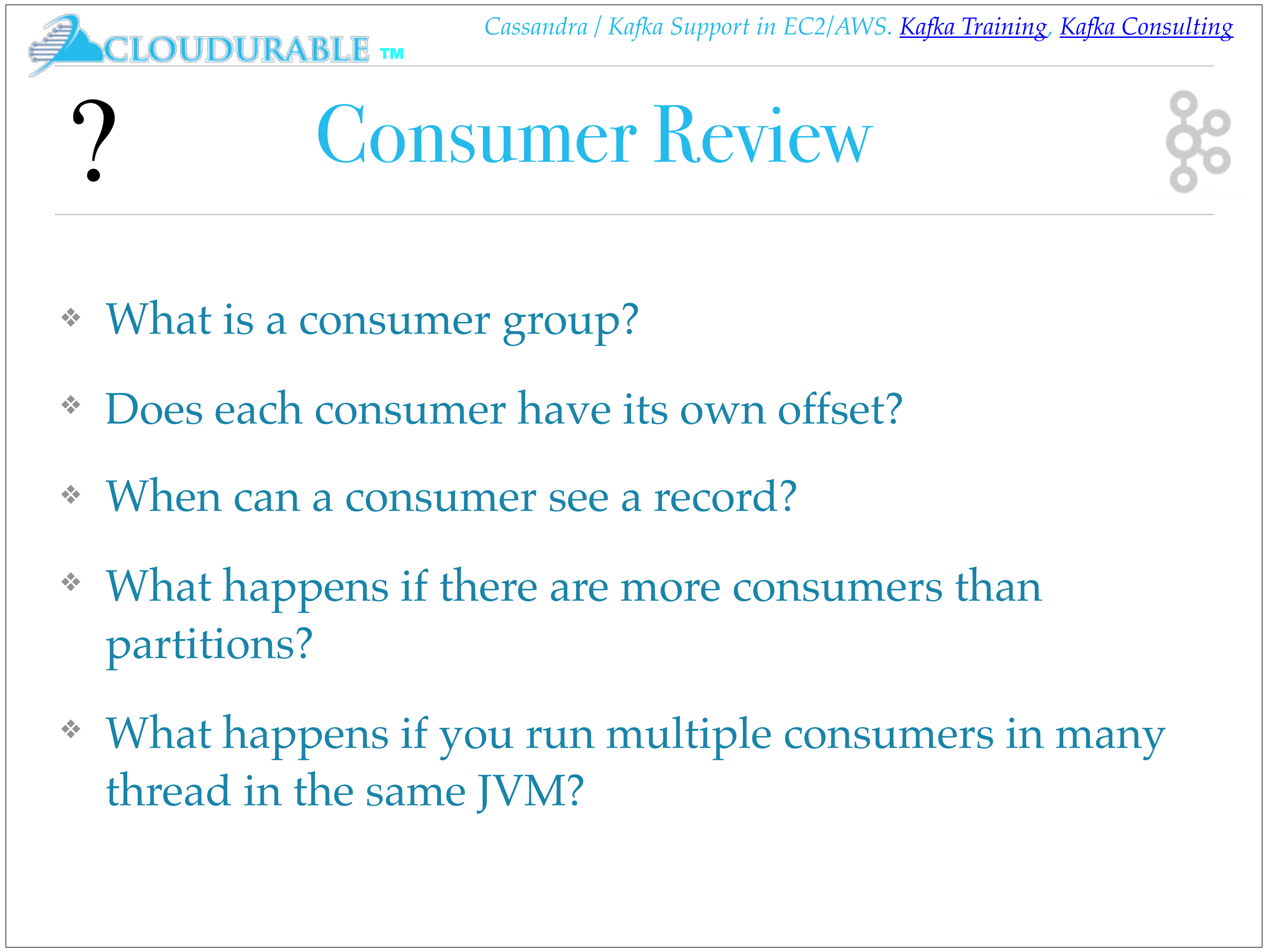
Multi-threaded Consumers

- You can run more than one Consumer in a JVM process
- If processing records takes a while, a single Consumer can run multiple threads to process records
  - Harder to manage offset for each Thread / Task
  - One Consumer runs multiple threads
  - 2 messages on same partitions being processed by two different threads
  - Hard to guarantee order without threads coordination
- ***PREFER***: Multiple Consumers can run each processing record batches in their own thread
  - Easier to manage offset
  - Each Consumer runs in its thread
  - Easier to mange failover (each process runs X num of Consumer threads)

You can run more than one Consumer in a JVM process by using threads.

If processing a record takes a while, a single Consumer can run multiple threads to process records, but it is harder to manage offset for each Thread/Task. If one consumer runs multiple threads, then two messages on the same partitions could be processed by two different threads which make it hard to guarantee record delivery order without complex thread coordination. This setup might be appropriate if processing
a single task takes a long time, but try to avoid it.

If you need to run multiple consumers, then run each consumer in their own thread. This way Kafka can deliver record batches to the consumer and the consumer does not have to worry about the offset ordering. A thread per consumer makes it easier to manage offsets. It is also simpler to manage failover (each process runs X num of consumer threads) as you can allow Kafka to do the brunt of the work.



? Consumer Review

- What is a consumer group?
- Does each consumer have its own offset?
- When can a consumer see a record?
- What happens if there are more consumers than partitions?
- What happens if you run multiple consumers in many thread in the same JVM?

What is a consumer group?

A consumer group is a group of related consumers that perform a task, like putting data into Hadoop or sending messages to a service. Consumer groups each have unique offsets per partition. Different consumer groups can read from different locations in a partition.

Does each consumer group have its own offset?
Yes. The consumer groups have their own offset for every partition in the topic which is unique to what other consumer groups have.

When can a consumer see a record?

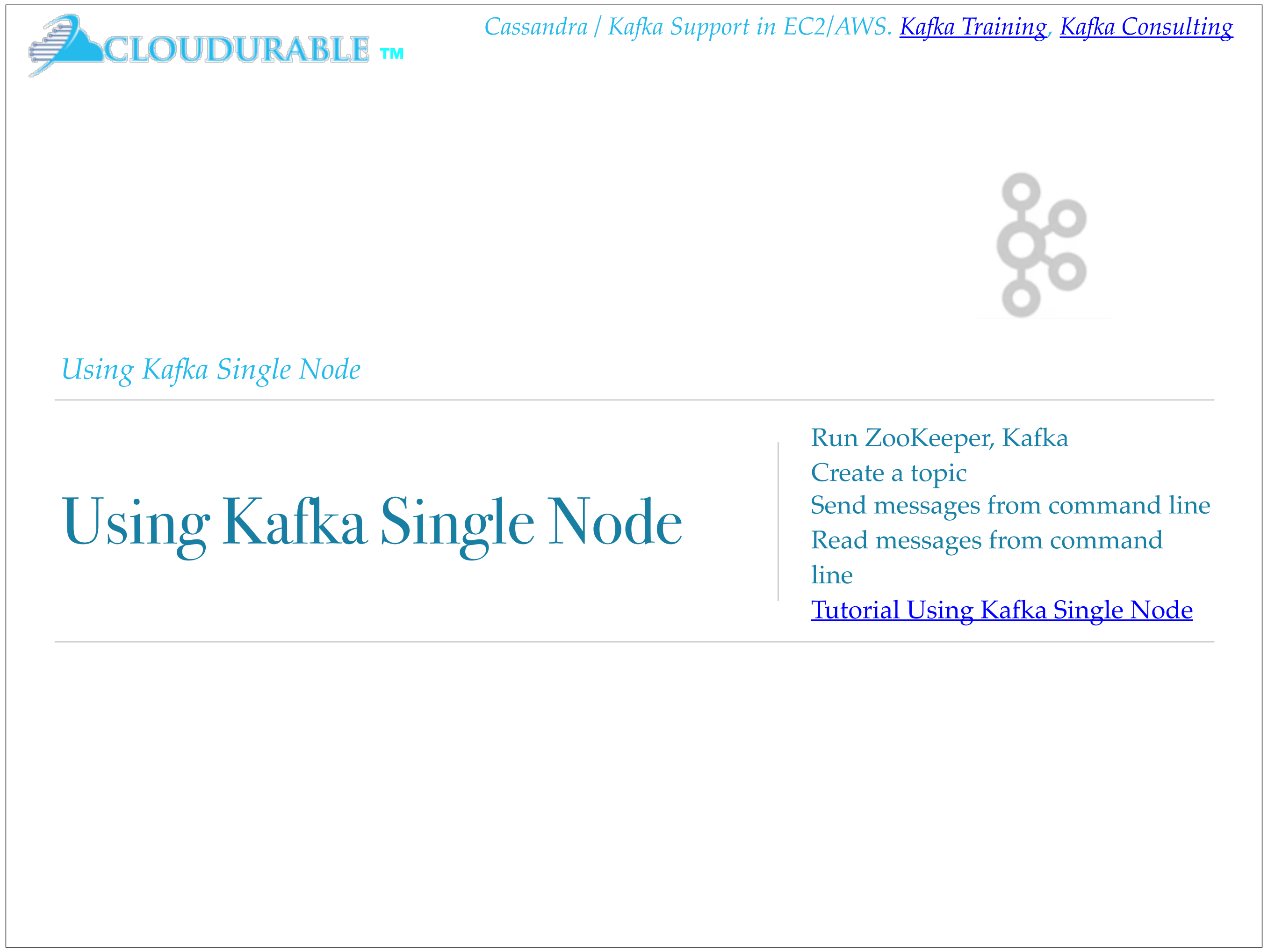
A consumer can see a record after the record gets fully replicated to all followers.

What happens if there are more consumers than partitions?

The extra consumers remain idle until another consumer dies.

What happens if you run multiple consumers in many threads in the same JVM?

Each thread manages a share of partitions for that consumer group.

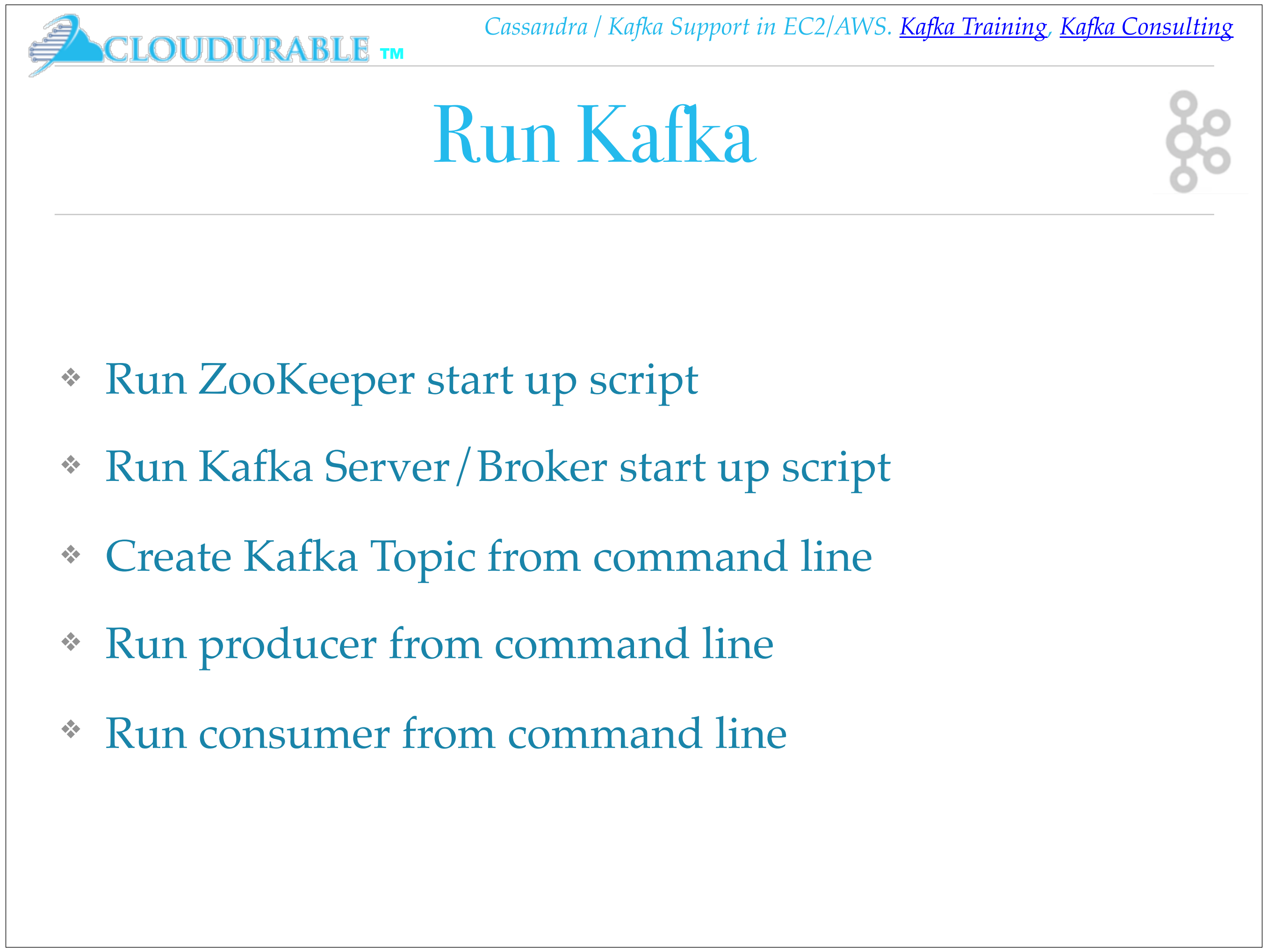


Using Kafka Single Node

Using Kafka Single Node

Run ZooKeeper, Kafka
Create a topic
Send messages from command line
Read messages from command line
Tutorial Using Kafka Single Node

Let’s show a simple example using producers and consumers from the Kafka command line.
Download Kafka 0.10.2.x from the Kafka download page. Later versions will likely work, but this was example was done with 0.10.2.x.
We assume that you have Java SDK 1.8.x installed.
We unzipped the Kafka download and put it in ~/kafka-training/, and then renamed the Kafka install folder to kafka. Please do the same.

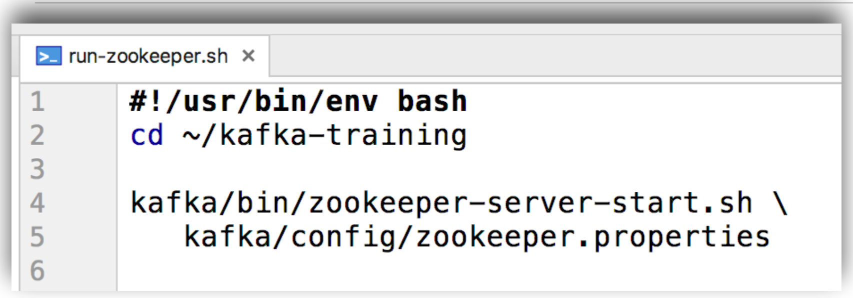


Run Kafka

- Run ZooKeeper start up script
- Run Kafka Server/Broker start up script
- Create Kafka Topic from command line
- Run producer from command line
- Run consumer from command line

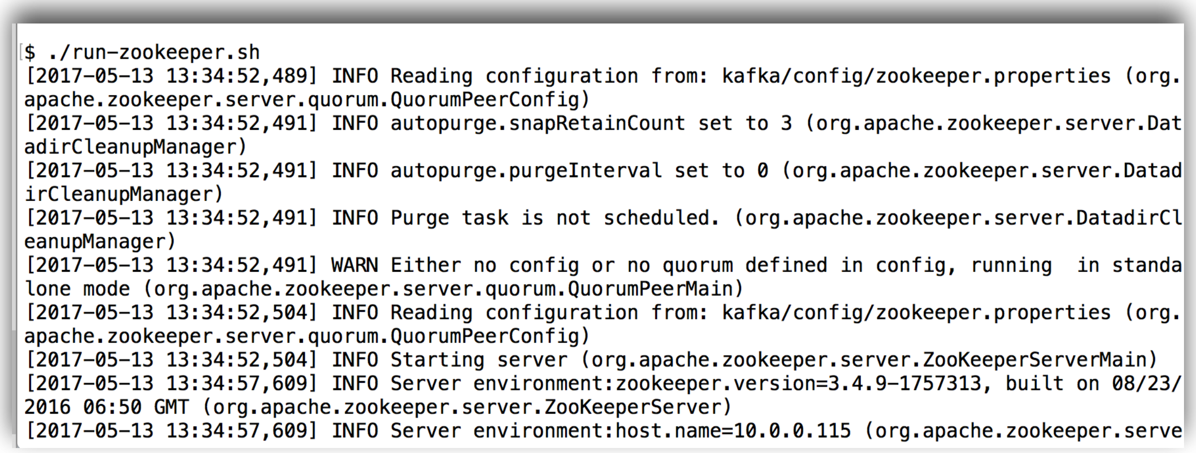
Next, we are going to run ZooKeeper and then run Kafka Server/Broker. We will use some Kafka command line utilities, to create Kafka topics, send messages via a producer and consume messages from the command line.

Run ZooKeeper



run-zookeeper.sh

```
#!/usr/bin/env bash
cd ~/kafka-training

kafka/bin/zookeeper-server-start.sh \
   kafka/config/zookeeper.properties
```



```
$ ./run-zookeeper.sh
[2017-05-13 13:34:52,489] INFO Reading configuration from: kafka/config/zookeeper.properties (org.
apache.zookeeper.server.quorum.QuorumPeerConfig)
[2017-05-13 13:34:52,491] INFO autopurge.snapRetainCount set to 3 (org.apache.zookeeper.server.Dat
adirCleanupManager)
[2017-05-13 13:34:52,491] INFO autopurge.purgeInterval set to 0 (org.apache.zookeeper.server.Datad
irCleanupManager)
[2017-05-13 13:34:52,491] INFO Purge task is not scheduled. (org.apache.zookeeper.server.DatadirCl
eanupManager)
[2017-05-13 13:34:52,491] WARN Either no config or no quorum defined in config, running  in standa
lone mode (org.apache.zookeeper.server.quorum.QuorumPeerMain)
[2017-05-13 13:34:52,504] INFO Reading configuration from: kafka/config/zookeeper.properties (org.
apache.zookeeper.server.quorum.QuorumPeerConfig)
[2017-05-13 13:34:52,504] INFO Starting server (org.apache.zookeeper.server.ZooKeeperServerMain)
[2017-05-13 13:34:57,609] INFO Server environment:zookeeper.version=3.4.9-1757313, built on 08/23/
2016 06:50 GMT (org.apache.zookeeper.server.ZooKeeperServer)
[2017-05-13 13:34:57,609] INFO Server environment:host.name=10.0.0.115 (org.apache.zookeeper.serve
```

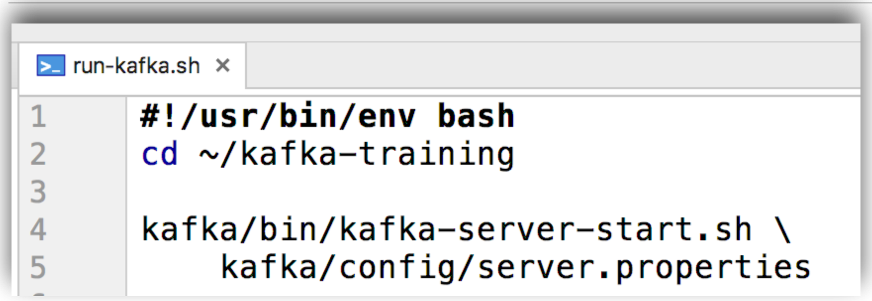
Kafka relies on ZooKeeper. To keep things simple, we will use a single ZooKeeper node. Kafka provides a startup script for ZooKeeper called zookeeper-server-start.sh which is located at ~/kafka-training/kafka/bin/zookeeper-server-start.sh.
The Kafka distribution also provide a ZooKeeper config file which is setup to run single node.

To run ZooKeeper, we create this script in kafka-training and run it.
~/kafka-training/run-zookeeper.sh

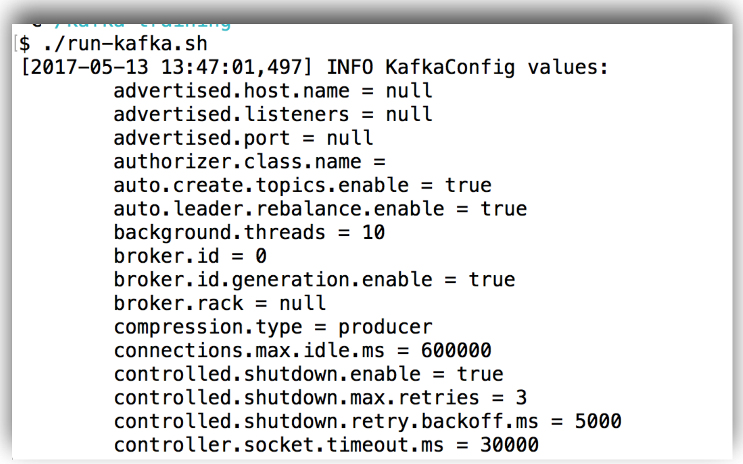
```
#!/usr/bin/env bash
cd ~/kafka-training

kafka/bin/zookeeper-server-start.sh \
   kafka/config/zookeeper.properties
```

Run run-zookeeper.sh
~/kafka-training
$ ./run-zookeeper.sh
Wait about 30 seconds or so for ZooKeeper to startup.

Run Kafka Server



```
#!/usr/bin/env bash
cd ~/kafka-training

kafka/bin/kafka-server-start.sh \
    kafka/config/server.properties
```



```
$ ./run-kafka.sh
[2017-05-13 13:47:01,497] INFO KafkaConfig values:
        advertised.host.name = null
        advertised.listeners = null
        advertised.port = null
        authorizer.class.name =
        auto.create.topics.enable = true
        auto.leader.rebalance.enable = true
        background.threads = 10
        broker.id = 0
        broker.id.generation.enable = true
        broker.rack = null
        compression.type = producer
        connections.max.idle.ms = 600000
        controlled.shutdown.enable = true
        controlled.shutdown.max.retries = 3
        controlled.shutdown.retry.backoff.ms = 5000
        controller.socket.timeout.ms = 30000
```

Kafka also provides a startup script for the Kafka server called kafka-server-start.sh which is located at ~/kafka-training/kafka/bin/kafka-server-start.sh.

The Kafka distribution also provide a Kafka config file which is setup to run Kafka single node, and points to ZooKeeper running on localhost:2181.

To run Kafka, we create this script in kafka-training and run it in another terminal window.

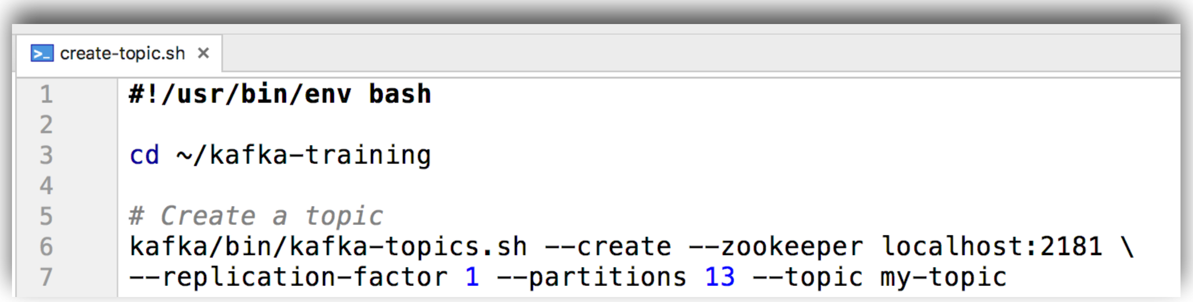
```
#!/usr/bin/env bash
cd ~/kafka-training

kafka/bin/zookeeper-server-start.sh \
  kafka/config/zookeeper.properties
```

Run run-kafka.sh

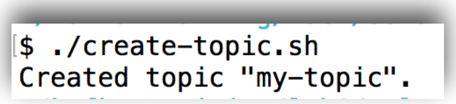
```
$ ./run-kafka.sh
```

Wait about 30 seconds or so for Kafka to startup.
Now let's create the topic that we will send records on.

Create Kafka Topic


create-topic.sh

```
#!/usr/bin/env bash

cd ~/kafka-training

# Create a topic
kafka/bin/kafka-topics.sh --create --zookeeper localhost:2181 \
--replication-factor 1 --partitions 13 --topic my-topic
```



```
$ ./create-topic.sh
Created topic "my-topic".
```

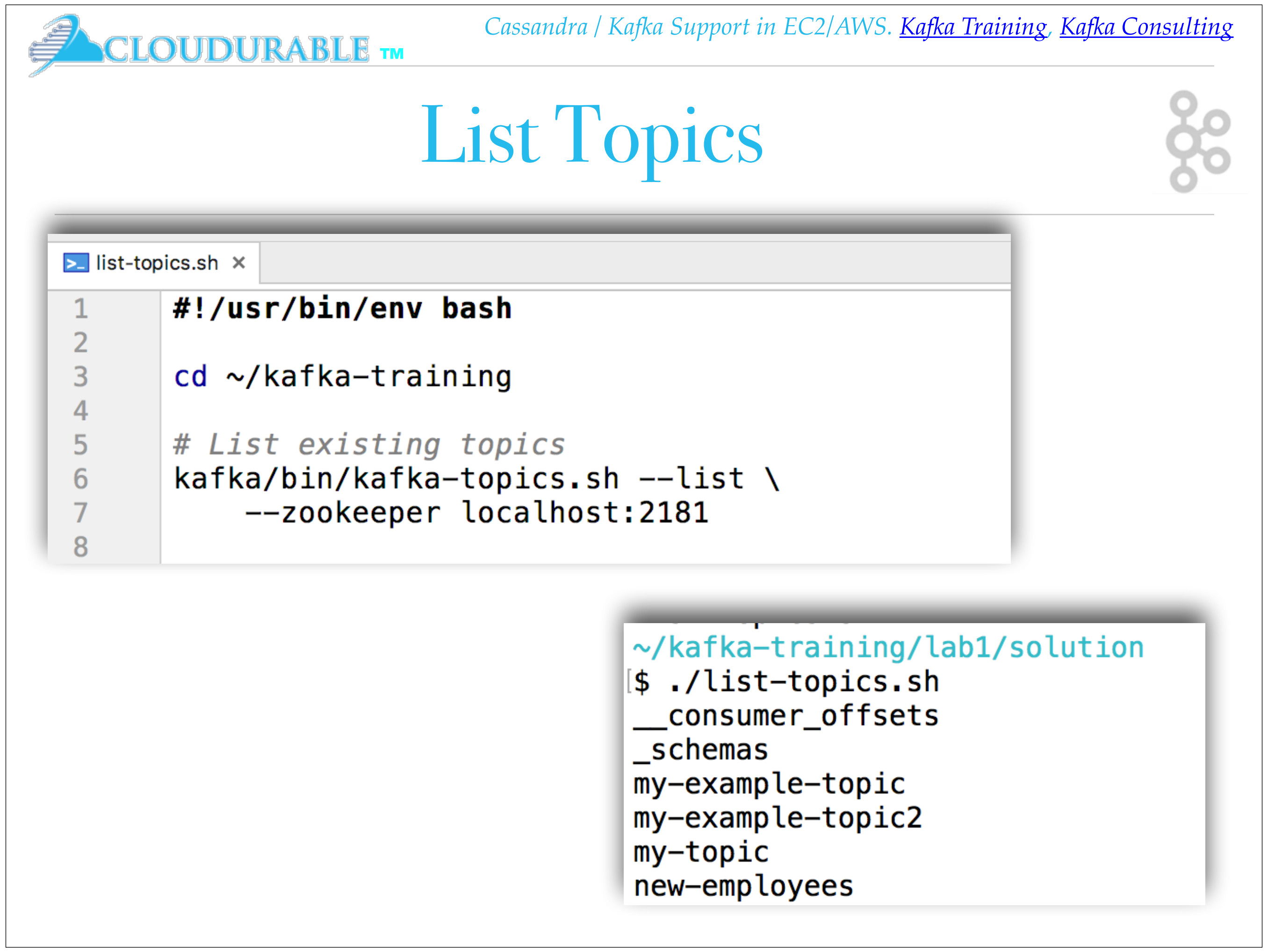
Kafka also provides a utility to work with topics called kafka-topics.sh which is located at ~/kafka-training/kafka/bin/kafka-topics.sh. We will use this tool to create a topic called my-topic with a replication factor of 1 since we only have one server. We will use thirteen partitions for my-topic, which means we could have up to 13 Kafka consumers.
To run Kafka, create this script in kafka-training\lab1, and run it in another terminal window.

```
#!/usr/bin/env bash
cd ~/kafka-training
# Create a topic
kafka/bin/kafka-topics.sh --create \
  --zookeeper localhost:2181 \
  --replication-factor 1 --partitions 13 \
  --topic my-topic
```

Run create-topic.sh
~/kafka-training/lab1

```
$ ./create-topic.sh
```

```
Created topic "my-topic".
```



List Topics

```
#!/usr/bin/env bash

cd ~/kafka-training

# List existing topics
kafka/bin/kafka-topics.sh --list \
    --zookeeper localhost:2181

```

```
~/kafka-training/lab1/solution
$ ./list-topics.sh
__consumer_offsets
_schemas
my-example-topic
my-example-topic2
my-topic
new-employees
```

You can see which topics that Kafka is managing using kafka-topics.sh as follows.
Create the file in ~/kafka-training/lab1/list-topics.sh. and run it.

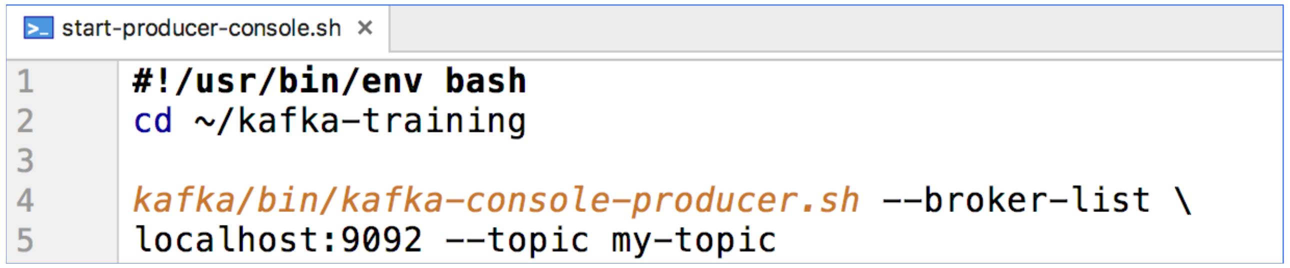
```
#!/usr/bin/env bash
cd ~/kafka-training
# List existing topics
kafka/bin/kafka-topics.sh --list \
   --zookeeper localhost:2181
```

Notice that we have to specify the location of the ZooKeeper cluster node which is running on localhost port 2181.

```
~/kafka-training/lab1
$ ./list-topics.sh
__consumer_offsets
_schemas
my-topic
```

You can see the topic my-topic in the list of topics.

Run Kafka Producer


start-producer-console.sh

```
#!/usr/bin/env bash
cd ~/kafka-training

kafka/bin/kafka-console-producer.sh --broker-list \
localhost:9092 --topic my-topic
```

The Kafka distribution provides a command utility to send messages from the command line. It start up a terminal window where everything you type is sent to the Kafka topic. Kafka provides the utility kafka-console-producer.sh which is located at ~/kafka-training/kafka/bin/kafka-console-producer.sh to send messages to a topic on the command line.
Create the file in ~/kafka-training/lab1/start-producer-console.sh and run it.

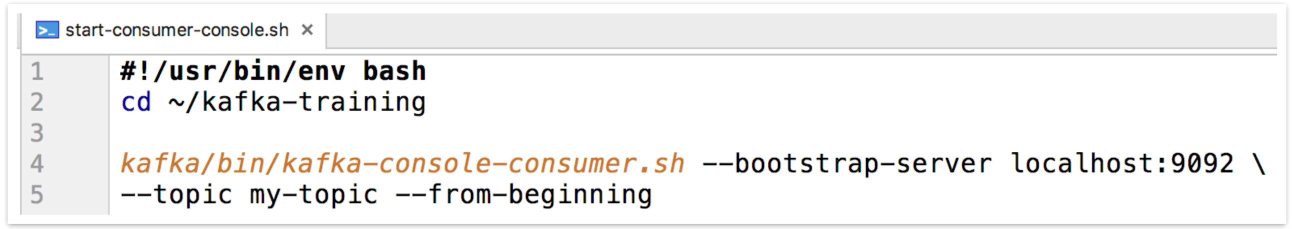
```
#!/usr/bin/env bash
cd ~/kafka-training
kafka/bin/kafka-console-producer.sh \
    --broker-list localhost:9092 \
    --topic my-topic
```

Notice that we specify the Kafka node which is running at localhost:9092.. Next run start-producer-console.sh and send at least four messages.

```
~/kafka-training/lab1
$ ./start-producer-console.sh
This is message 1
This is message 2
…
```

In order to see these messages, we will need to run the consumer console.

Run Kafka Consumer



start-consumer-console.sh

```
#!/usr/bin/env bash
cd ~/kafka-training

kafka/bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 \
--topic my-topic --from-beginning
```

The Kafka distribution provides a command utility to see messages from the command line. It displays the messages in various modes. Kafka provides the utility kafka-console-consumer.sh which is located at ~/kafka-training/kafka/bin/kafka-console-producer.sh to receive messages from a topic on the command line. Create the file in ~/kafka-training/lab1/start-consumer-console.sh and run it.

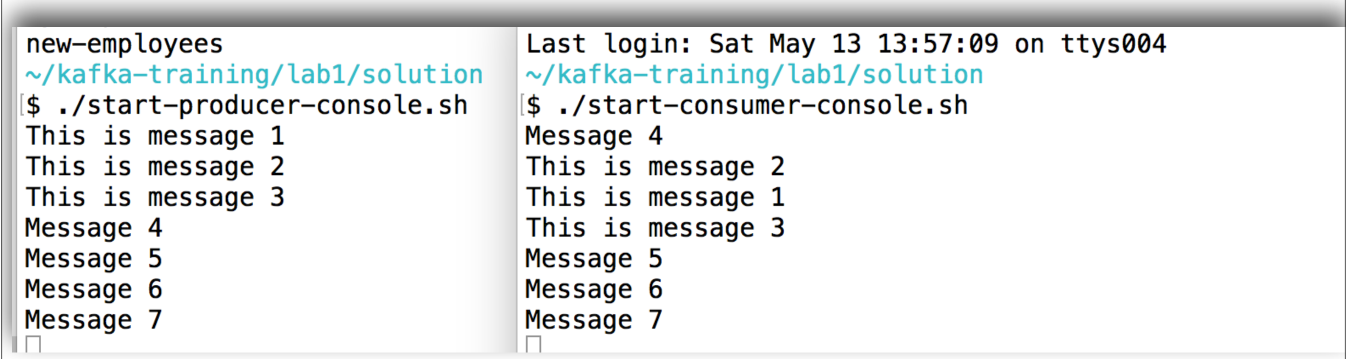
```
#!/usr/bin/env bash
cd ~/kafka-training
kafka/bin/kafka-console-consumer.sh \
    --bootstrap-server localhost:9092 \
    --topic my-topic \
    --from-beginning
```

Notice that we specify the Kafka node which is running at localhost:9092 like we did before, but we also specify to read all of the messages from my-topic from the beginning —from-beginning. Next, run start-consumer-console.sh in another terminal

```
$ ./start-consumer-console.sh
This is message 2
This is message 1
…
```

Why are the messages coming out of order?

Running Kafka Producer and Consumer

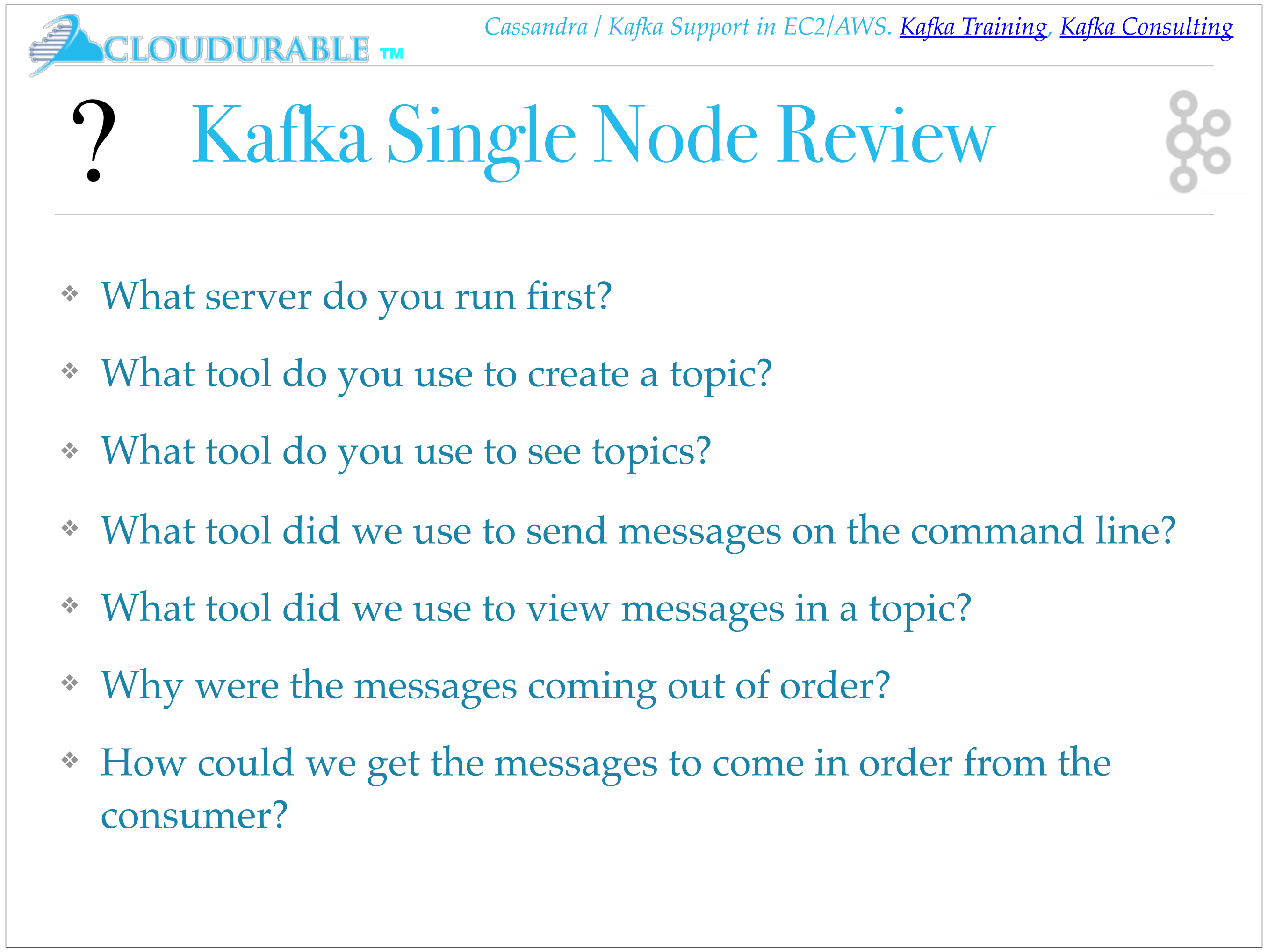


```
new-employees
~/kafka-training/lab1/solution
$ ./start-producer-console.sh
This is message 1
This is message 2
This is message 3
Message 4
Message 5
Message 6
Message 7
```

```
Last login: Sat May 13 13:57:09 on ttys004
~/kafka-training/lab1/solution
$ ./start-consumer-console.sh
Message 4
This is message 2
This is message 1
This is message 3
Message 5
Message 6
Message 7
```

Do you know why the messages are coming out of order?

Notice that the messages are not coming in order. This is because we only have one consumer so it is reading the messages from all 13 partitions. Order is only guaranteed within a partition.



? Kafka Single Node Review

- What server do you run first?
- What tool do you use to create a topic?
- What tool do you use to see topics?
- What tool did we use to send messages on the command line?
- What tool did we use to view messages in a topic?
- Why were the messages coming out of order?
- How could we get the messages to come in order from the consumer?

What server do you run first?

You need to run ZooKeeper then Kafka.

What tool do you use to create a topic?

kafka-topics.sh

What tool do you use to see topics?

kafka-topics.sh

What tool did we use to send messages on the command line?

kafka-console-producer.sh

What tool did we use to view messages in a topic?

kafka-console-consumer.sh

Why were the messages coming out of order?

The messages were being sharded among 13 partitions.

How could we get the messages to come in order from the consumer?

Use Kafka to send and receive messages

Lab Use Kafka

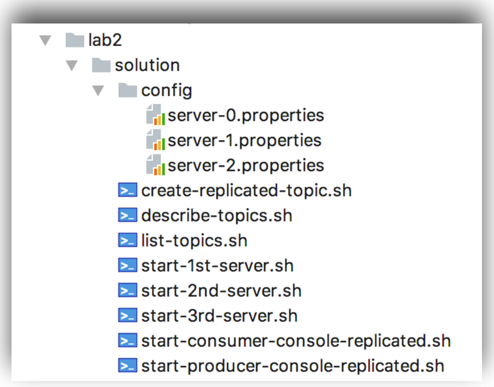
Use single server version of Kafka.
Setup single node.
Single ZooKeeper.
Create a topic.
Produce and consume messages from the command line.

Using Kafka Cluster and Failover

Demonstrate Kafka Cluster
Create topic with replication
Show consumer failover
Show broker failover
Kafka Tutorial Cluster and Failover

Objectives

- Run many Kafka Brokers
- Create a replicated topic
- Demonstrate Pub / Sub
- Demonstrate load balancing consumers
- Demonstrate consumer failover
- Demonstrate broker failover

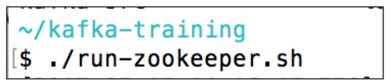


```
lab2
  solution
    config
      server-0.properties
      server-1.properties
      server-2.properties
    create-replicated-topic.sh
    describe-topics.sh
    list-topics.sh
    start-1st-server.sh
    start-2nd-server.sh
    start-3rd-server.sh
    start-consumer-console-replicated.sh
    start-producer-console-replicated.sh
```

In this lab, we are going to run many Kafka Nodes on our development laptop so that you will need at least 16 GB of RAM for local dev machine. You can run just two servers if you have less memory than 16 GB. We are going to create a replicated topic. We then demonstrate consumer failover and broker failover. We also demonstrate load balancing Kafka consumers. We show how, with many groups, Kafka acts like a Publish/Subscribe. But, when we put all of our consumers in the same group, Kafka will load share the messages to the consumers in the same group (more like a queue than a topic in a traditional MOM sense).

Running many nodes

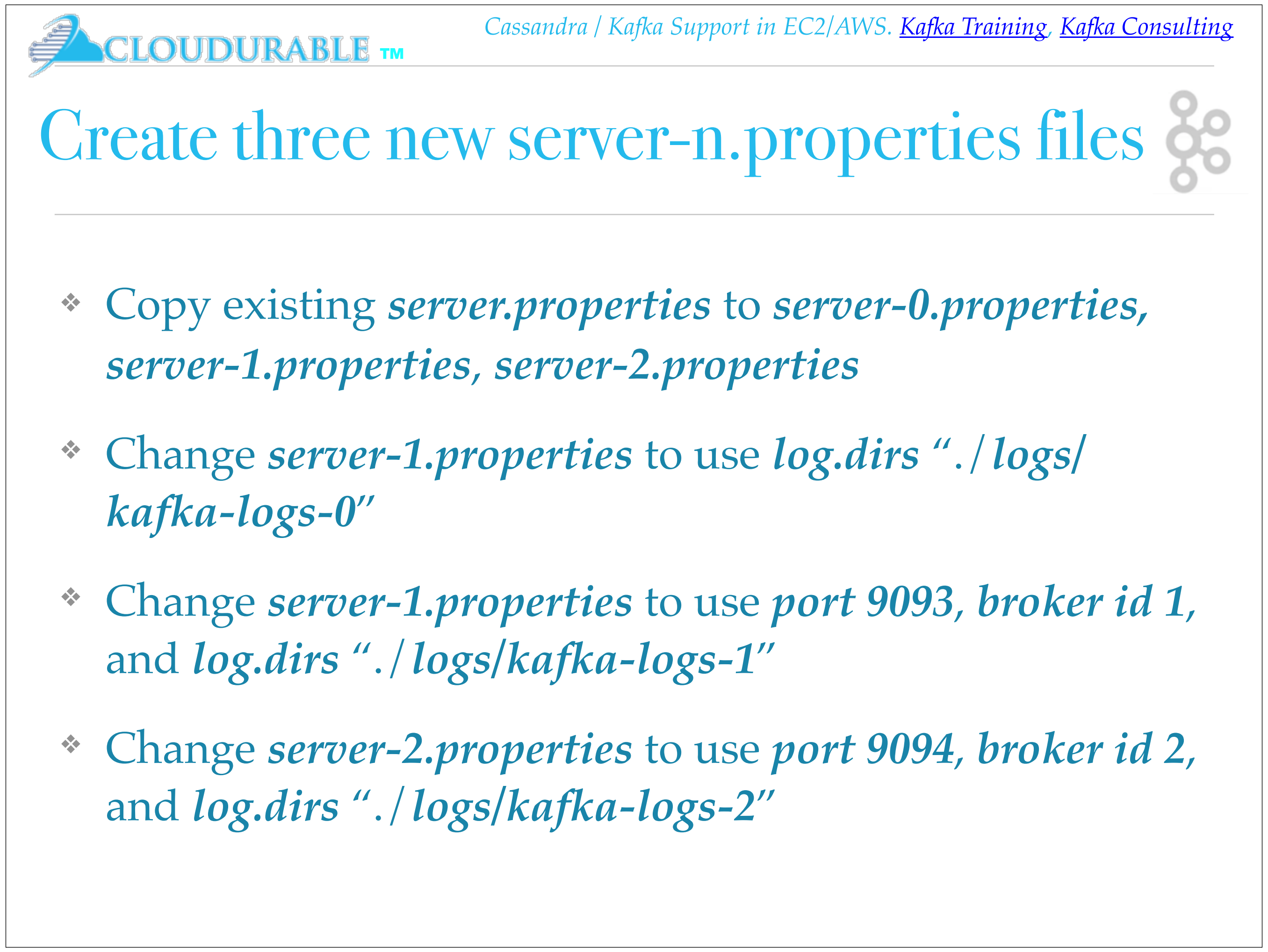
- If not already running, start up ZooKeeper
  - Shutdown Kafka from first lab
- Copy server properties for three brokers
  - Modify properties files, Change port, Change Kafka log location
- Start up many Kafka server instances
- Create Replicated Topic
- Use the replicated topic



```
~/kafka-training
$ ./run-zookeeper.sh
```

If not already running, then start up ZooKeeper (./run-zookeeper.sh from the first tutorial). Also, shut down Kafka from the first lab.

Next, you need to copy server properties for three brokers (detailed instructions to follow). Then we will modify these Kafka server properties to add unique Kafka ports, Kafka log locations, and unique Broker ids. Then we will create three scripts to start these servers up using these properties, and then start the servers. Lastly, we create replicated topic and use it to demonstrate Kafka consumer failover, and Kafka broker failover.



Create three new server-n.properties files

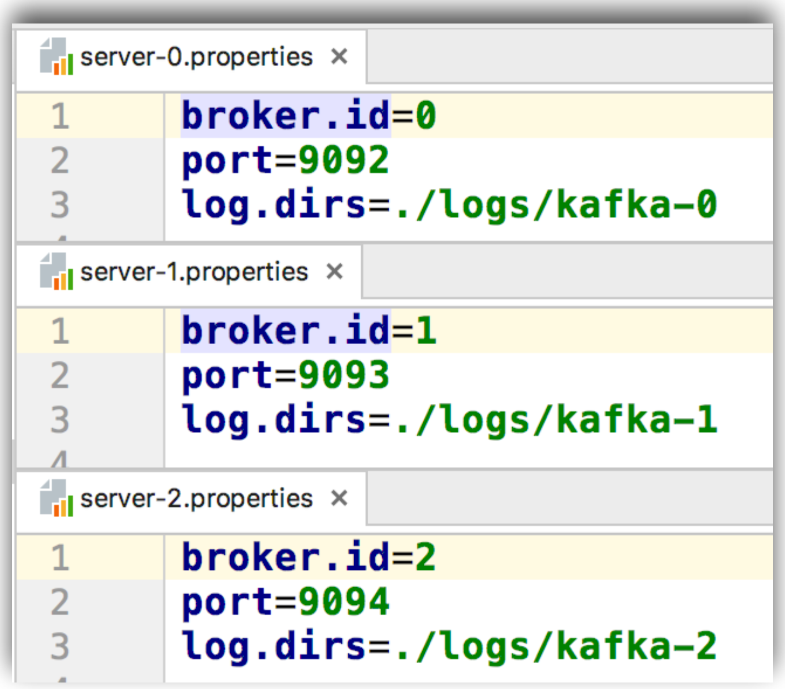
- Copy existing *server.properties* to *server-0.properties, server-1.properties, server-2.properties*
- Change *server-1.properties* to use *log.dirs "./logs/kafka-logs-0"*
- Change *server-1.properties* to use *port 9093, broker id 1,* and *log.dirs "./logs/kafka-logs-1"*
- Change *server-2.properties* to use *port 9094, broker id 2,* and *log.dirs "./logs/kafka-logs-2"*

Create three new Kafka server-n.properties files
In this section, we will copy the existing Kafka server.properties to server-0.properties, server-1.properties, and server-2.properties. Then we change server-0.properties to set log.dirs to “./logs/kafka-0. Then we modify server-1.properties to set port to 9093, broker id to 1, and log.dirs to “./logs/kafka-1”. Lastly modify server-2.propertiesto use port 9094, broker id 2, and log.dirs “./logs/kafka-2”.

Copy server properties file.

```
$ ~/kafka-training
$ mkdir -p lab2/config
$ cp kafka/config/server.properties kafka/lab2/config/server-0.properties
$ cp kafka/config/server.properties kafka/lab2/config/server-1.properties
$ cp kafka/config/server.properties kafka/lab2/config/server-2.properties
```

Modify server-x.properties


server-0.properties
```
broker.id=0
port=9092
log.dirs=./logs/kafka-0
```
server-1.properties
```
broker.id=1
port=9093
log.dirs=./logs/kafka-1
```
server-2.properties
```
broker.id=2
port=9094
log.dirs=./logs/kafka-2
```

- Each have different *broker.id*
- Each have different *log.dirs*
- Each had different *port*

With your favorite text editor change server-0.properties so that log.dirs is set to ./logs/kafka-0. Leave the rest of the file the same. Make sure log.dirs is only defined once. Modify ~/kafka-training/lab2/config/server-0.properties as follows:

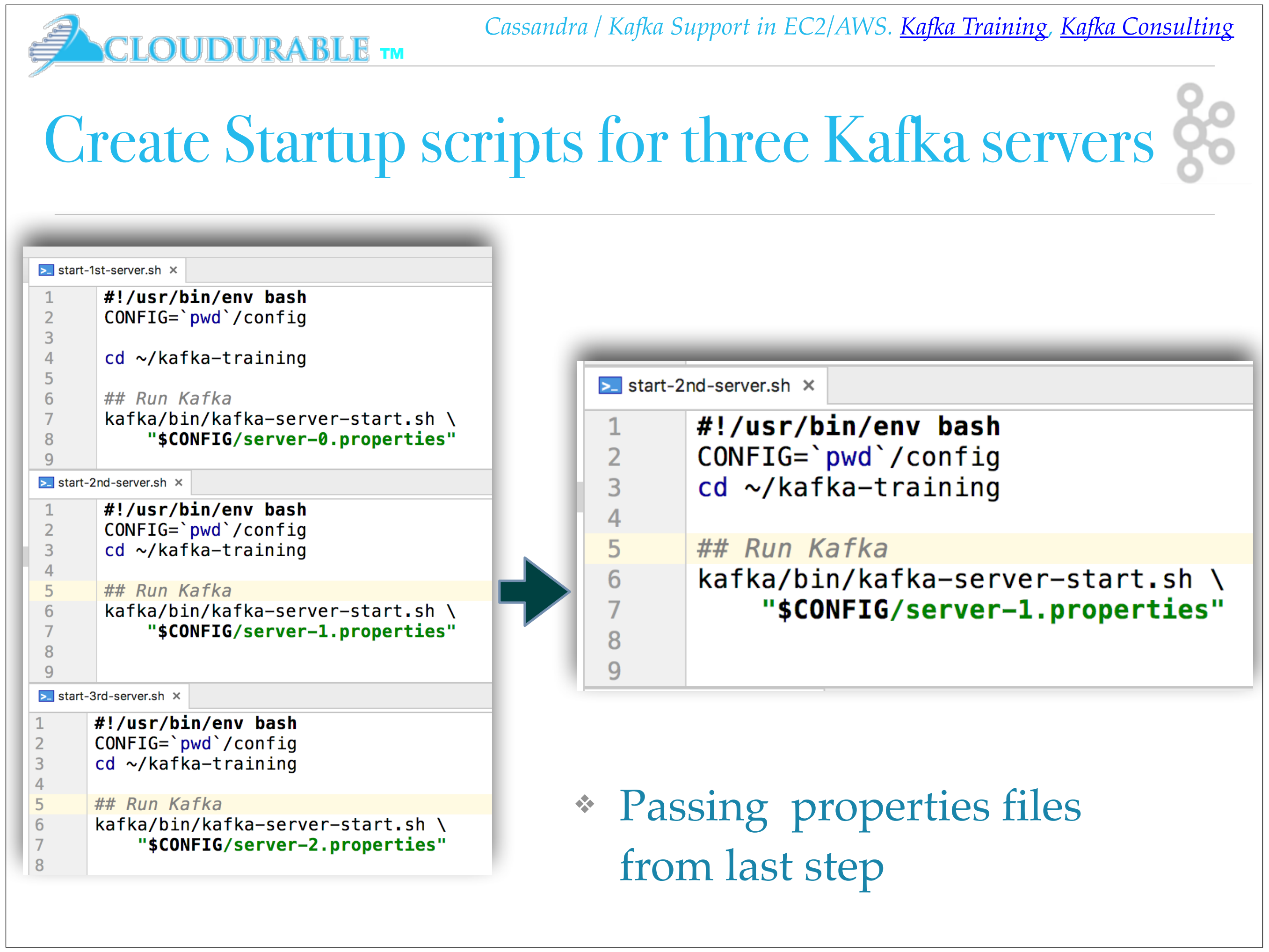
```
broker.id=0
port=9092
log.dirs=./logs/kafka-0
...
```

With your favorite text editor change log.dirs, broker.id and and log.dirs of server-1.properties as follows. Modify ~/kafka-training/lab2/config/server-1.properties as follows:

```
broker.id=1
port=9093
log.dirs=./logs/kafka-1
...
```

With your favorite text editor change log.dirs, broker.id and and log.dirs of server-2.properties as follows. Modify ~/kafka-training/lab2/config/server-2.properties as follows.

```
broker.id=2
port=9094
log.dirs=./logs/kafka-2
...
```



Create Startup scripts for three Kafka servers

start-1st-server.sh

```
#!/usr/bin/env bash
CONFIG=`pwd`/config

cd ~/kafka-training

## Run Kafka
kafka/bin/kafka-server-start.sh \
    "$CONFIG/server-0.properties"
```

start-2nd-server.sh

```
#!/usr/bin/env bash
CONFIG=`pwd`/config
cd ~/kafka-training

## Run Kafka
kafka/bin/kafka-server-start.sh \
    "$CONFIG/server-1.properties"
```

start-3rd-server.sh

```
#!/usr/bin/env bash
CONFIG=`pwd`/config
cd ~/kafka-training

## Run Kafka
kafka/bin/kafka-server-start.sh \
    "$CONFIG/server-2.properties"
```

start-2nd-server.sh

```
#!/usr/bin/env bash
CONFIG=`pwd`/config
cd ~/kafka-training

## Run Kafka
kafka/bin/kafka-server-start.sh \
    "$CONFIG/server-1.properties"
```

- Passing properties files from last step

Create Startup scripts for three Kafka servers. The startup scripts will just run kafka-server-start.sh with the corresponding properties file.

Create ~/kafka-training/lab2/start-1st-server.sh as follows:

```
#!/usr/bin/env bash
CONFIG=`pwd`/config
cd ~/kafka-training
## Run Kafka
kafka/bin/kafka-server-start.sh \
    "$CONFIG/server-0.properties"
```

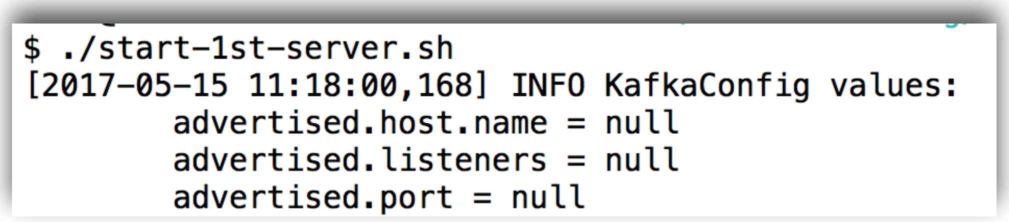
Create ~/kafka-training/lab2/start-2nd-server.sh as follows:

```
#!/usr/bin/env bash
CONFIG=`pwd`/config
cd ~/kafka-training
## Run Kafka
kafka/bin/kafka-server-start.sh \
    "$CONFIG/server-1.properties"
```

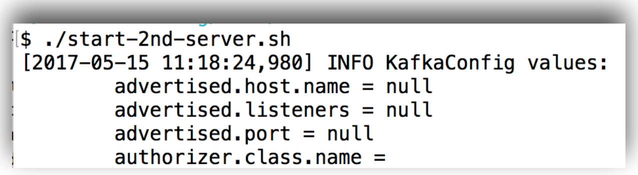
Create ~/kafka-training/lab2/start-3rd-server.sh like the last two but use server-2.properties for the config.

Notice we are passing the Kafka server properties files that we created in the last step.
Now run all three in separate terminals/shells.

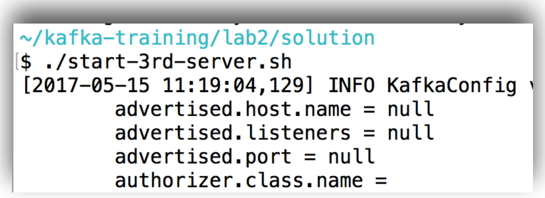
Run Servers



```
$ ./start-1st-server.sh
[2017-05-15 11:18:00,168] INFO KafkaConfig values:
        advertised.host.name = null
        advertised.listeners = null
        advertised.port = null
```



```
$ ./start-2nd-server.sh
[2017-05-15 11:18:24,980] INFO KafkaConfig values:
        advertised.host.name = null
        advertised.listeners = null
        advertised.port = null
        authorizer.class.name =
```

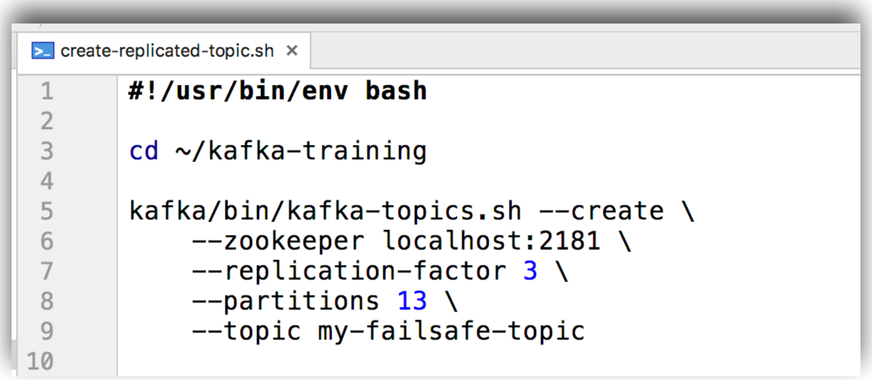


```
~/kafka-training/lab2/solution
$ ./start-3rd-server.sh
[2017-05-15 11:19:04,129] INFO KafkaConfig v
        advertised.host.name = null
        advertised.listeners = null
        advertised.port = null
        authorizer.class.name =
```

Run Kafka servers each in own terminal from ~/kafka-training/lab2
$ ./start-1st-server.sh
...
$ ./start-2nd-server.sh
...
$ ./start-3rd-server.sh

Give the servers a minute to startup and connect to ZooKeeper.

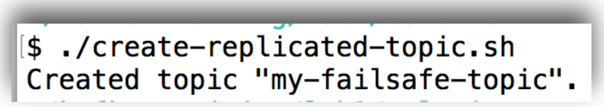
Create Kafka replicated topic **my-failsafe-topic**



create-replicated-topic.sh

```
#!/usr/bin/env bash

cd ~/kafka-training

kafka/bin/kafka-topics.sh --create \
    --zookeeper localhost:2181 \
    --replication-factor 3 \
    --partitions 13 \
    --topic my-failsafe-topic

```

- ***Replication Factor*** is set to 3
- Topic name is ***my-failsafe-topic***
- ***Partitions*** is 13



```
$ ./create-replicated-topic.sh
Created topic "my-failsafe-topic".
```

Create Kafka replicated topic my-failsafe-topic. Now create a replicated topic that the console producers and console consumers can use.

Create ~/kafka-training/lab2/create-replicated-topic.sh as follows:

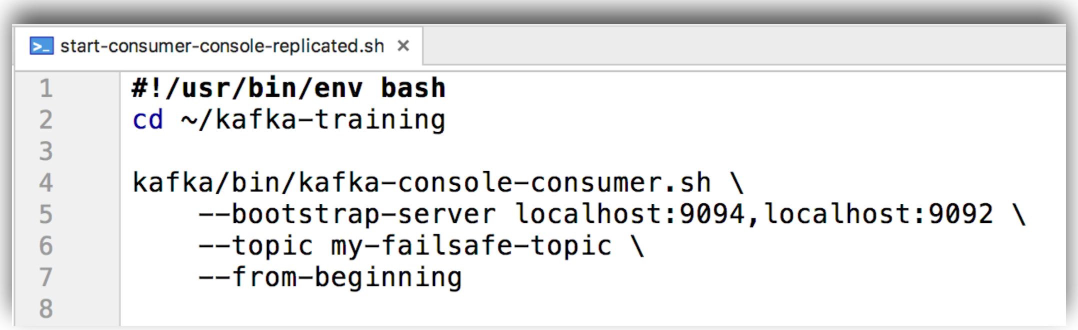
```
#!/usr/bin/env bash
cd ~/kafka-training
kafka/bin/kafka-topics.sh --create \
  --zookeeper localhost:2181 \
  --replication-factor 3 \
  --partitions 13 \
  --topic my-failsafe-topic
```

Notice that the replication factor gets set to 3, and the topic name is my-failsafe-topic, and like before it has 13 partitions. Then we just have to run the script to create the topic.

Run create-replicated-topic.sh as follows:

```
~/kafka-training/lab2
$ ./create-replicated-topic.sh
```

Start Kafka Consumer


start-consumer-console-replicated.sh

```
#!/usr/bin/env bash
cd ~/kafka-training

kafka/bin/kafka-console-consumer.sh \
    --bootstrap-server localhost:9094,localhost:9092 \
    --topic my-failsafe-topic \
    --from-beginning
```

- Pass list of Kafka servers to bootstrap-server
- We pass two of the three
- Only one needed, it learns about the rest

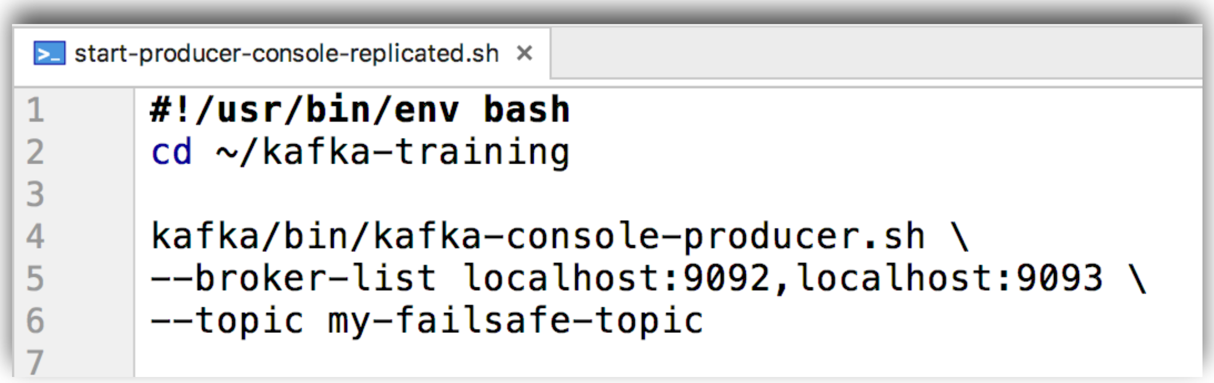
Start Kafka Consumer that uses the replicated topic. Next, create a script that starts the consumer and then start the consumer with the script (~/kafka-training/lab2/start-consumer-console-replicated.sh) as follows:

```
#!/usr/bin/env bash
cd ~/kafka-training
kafka/bin/kafka-console-consumer.sh \
    --bootstrap-server localhost:9094,localhost:9092 \
    --topic my-failsafe-topic \
    --from-beginning
```

Notice that a list of Kafka servers is passed to --bootstrap-server parameter. Only, two of the three servers get passed that we ran earlier. Even though only one broker is needed, the consumer client will learn about the other broker from just one server. Usually, you list multiple brokers in case there is an outage so that the client can connect.
Now we just run this start-consumer-console-replicated.sh to start the consumer as follows

```
$ ./start-consumer-console-replicated.sh
```

Start Kafka Producer


start-producer-console-replicated.sh

```
#!/usr/bin/env bash
cd ~/kafka-training

kafka/bin/kafka-console-producer.sh \
--broker-list localhost:9092,localhost:9093 \
--topic my-failsafe-topic

```

- Start producer
- Pass list of Kafka Brokers

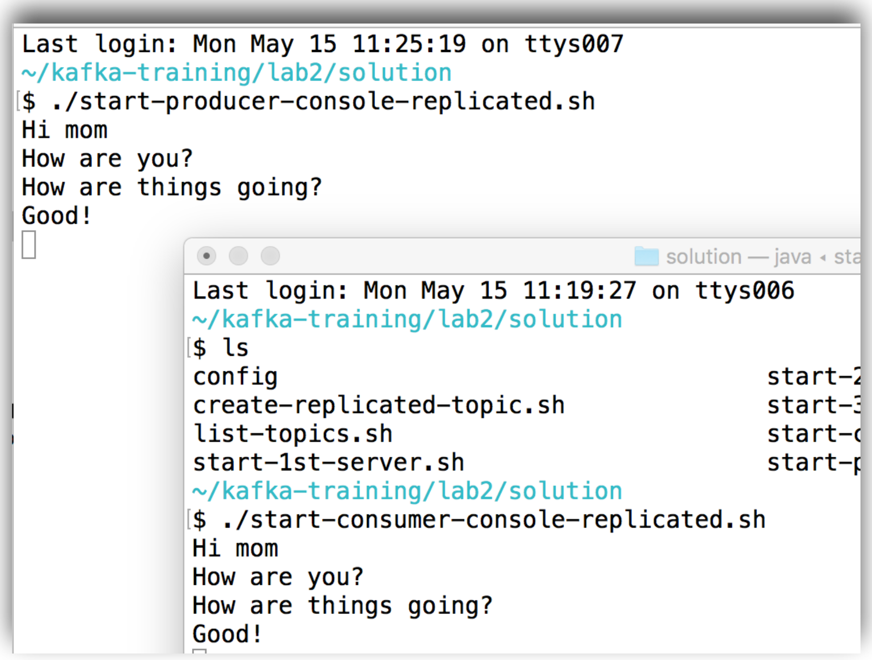
Start Kafka Producer that uses the replicated topic. Next, we create a script that starts the producer. Then launch the producer with the script you create. Create ~/kafka-training/lab2/start-consumer-producer-replicated.sh as follows:

```
#!/usr/bin/env bash
cd ~/kafka-training
kafka/bin/kafka-console-producer.sh \
--broker-list localhost:9092,localhost:9093 \
--topic my-failsafe-topic
```

Notice we start Kafka producer and pass it a list of Kafka Brokers to use via the parameter —broker-list. Now use the start-producer-console-replicated.sh to launch the producer as follows:

```
$ ./start-consumer-producer-replicated.sh
```

Kafka 1 consumer and 1 producer running



```
Last login: Mon May 15 11:25:19 on ttys007
~/kafka-training/lab2/solution
$ ./start-producer-console-replicated.sh
Hi mom
How are you?
How are things going?
Good!
```

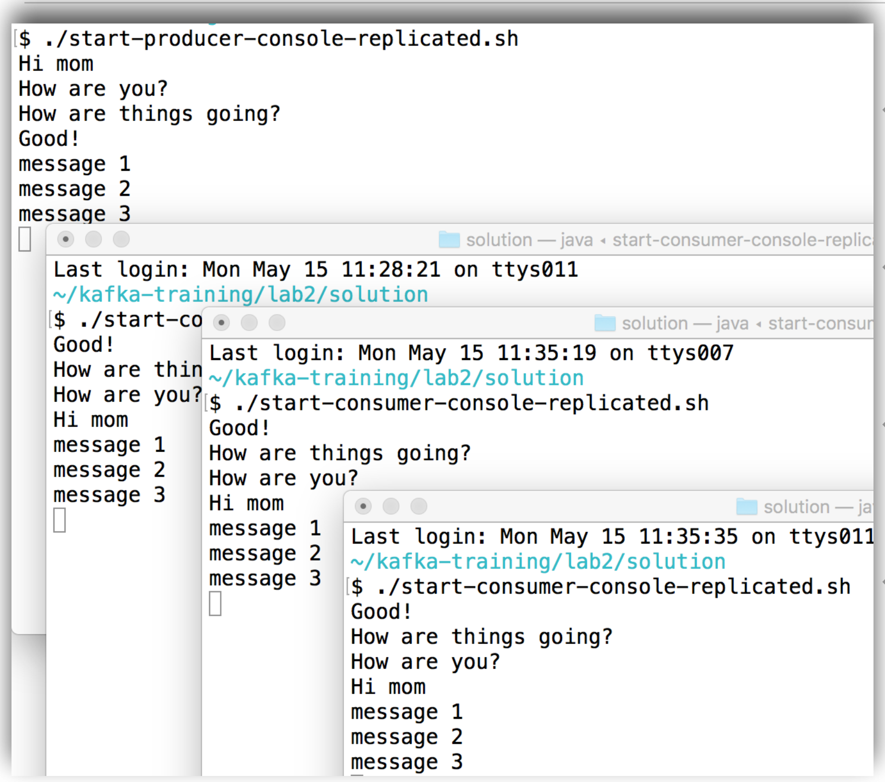
```
Last login: Mon May 15 11:19:27 on ttys006
~/kafka-training/lab2/solution
$ ls
config                                  start-2
create-replicated-topic.sh              start-3
list-topics.sh                          start-c
start-1st-server.sh                     start-p
~/kafka-training/lab2/solution
$ ./start-consumer-console-replicated.sh
Hi mom
How are you?
How are things going?
Good!
```

Now send some message from the producer to Kafka and see those messages consumed by the consumer.

Use producer console to send messages as follows:
~/kafka-training/lab2
$ ./start-consumer-producer-replicated.sh
Hi Mom
How are you?
How are things going?
Good!

See the messages in the Kafka consumer console
~/kafka-training/lab2
$ ./start-consumer-console-replicated.sh
Hi Mom
How are you?
How are things going?
Good!

Start a second and third consumer



```
$ ./start-producer-console-replicated.sh
Hi mom
How are you?
How are things going?
Good!
message 1
message 2
message 3
```

```
Last login: Mon May 15 11:28:21 on ttys011
~/kafka-training/lab2/solution
$ ./start-co
Good!
How are thin
How are you?
Hi mom
message 1
message 2
message 3
```

```
Last login: Mon May 15 11:35:19 on ttys007
~/kafka-training/lab2/solution
$ ./start-consumer-console-replicated.sh
Good!
How are things going?
How are you?
Hi mom
message 1
message 2
message 3
```

```
Last login: Mon May 15 11:35:35 on ttys011
~/kafka-training/lab2/solution
$ ./start-consumer-console-replicated.sh
Good!
How are things going?
How are you?
Hi mom
message 1
message 2
message 3
```

- Acts like pub / sub
- Each consumer in its own group
- Message goes to each
- How do we load share?

Now start two more consumers in their own terminal window and then send more messages from the producer. Start Producer as follows:
$ ./start-consumer-producer-replicated.sh
…
message 1
message 2
message 3

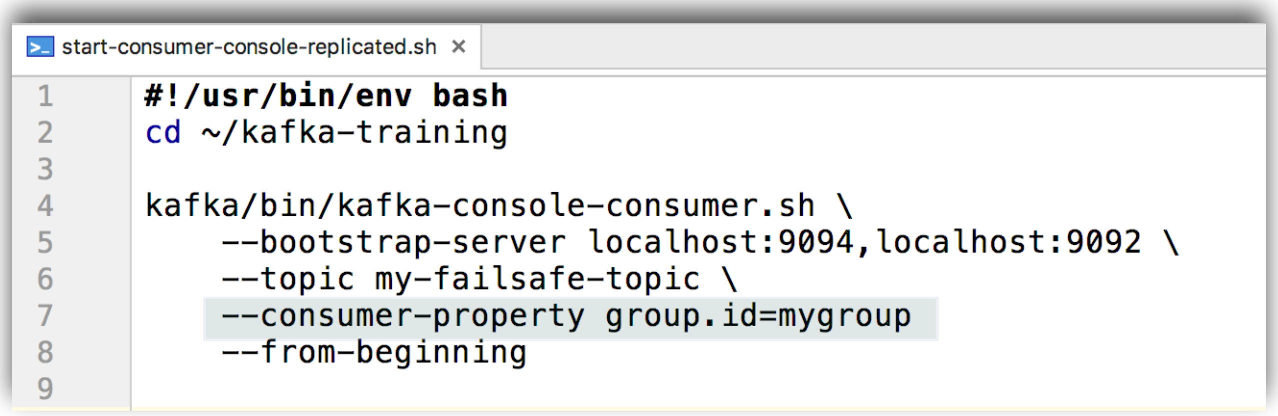
Then start first consumer.
$ ./start-consumer-console-replicated.sh
Hi Mom
…
message 1
message 2
message 3

..

$ ./start-consumer-console-replicated.sh
Hi Mom
…
message 2
message 3

Notice that the messages are sent to all of the consumers because each consumer is in a different consumer group.

Running consumers in same group


start-consumer-console-replicated.sh

```
#!/usr/bin/env bash
cd ~/kafka-training

kafka/bin/kafka-console-consumer.sh \
    --bootstrap-server localhost:9094,localhost:9092 \
    --topic my-failsafe-topic \
    --consumer-property group.id=mygroup
    --from-beginning

```

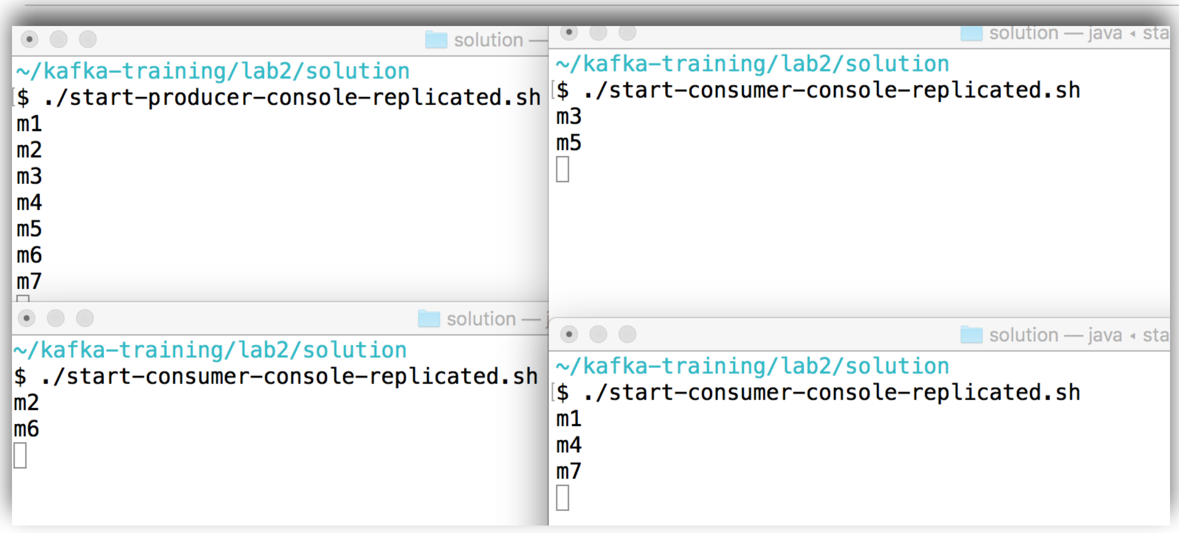
- Modify start consumer script
- Add the consumers to a group called mygroup
- Now they will share load

Now let's change the consumer to each be in their own consumer group. Stop the producers and the consumers from before, but leave Kafka and ZooKeeper running. Now let's modify the start-consumer-console-replicated.sh script to add a Kafka consumer group. We want to put all of the consumers in same consumer group. This way the consumers will share the messages as each consumer in the consumer group will get its share of partitions. Notice that the script is the same as before except we added --consumer-property group.id=mygroup which will put every consumer that runs with this script into the mygroup consumer group. Now we just run the producer and three consumers.

Run this three times - start-consumer-console-replicated.sh as follows:
$ ./start-consumer-console-replicated.sh

Then run producer console as follows:
$ ./start-consumer-producer-replicated.sh

Start up three consumers again



```
~/kafka-training/lab2/solution
$ ./start-producer-console-replicated.sh
m1
m2
m3
m4
m5
m6
m7
```

```
~/kafka-training/lab2/solution
$ ./start-consumer-console-replicated.sh
m3
m5
```

```
~/kafka-training/lab2/solution
$ ./start-consumer-console-replicated.sh
m2
m6
```

```
~/kafka-training/lab2/solution
$ ./start-consumer-console-replicated.sh
m1
m4
m7
```

- Start up producer and three consumers
- Send 7 messages
- Notice how messages are spread among 3 consumers

Now send seven messages from the Kafka producer console as follows:

```
$ ./start-consumer-producer-replicated.sh
m1
m2
m3
m4
m5
m6
m7
```

Notice that the messages are spread evenly among the consumers.

1st Kafka Consumer gets m3, m5

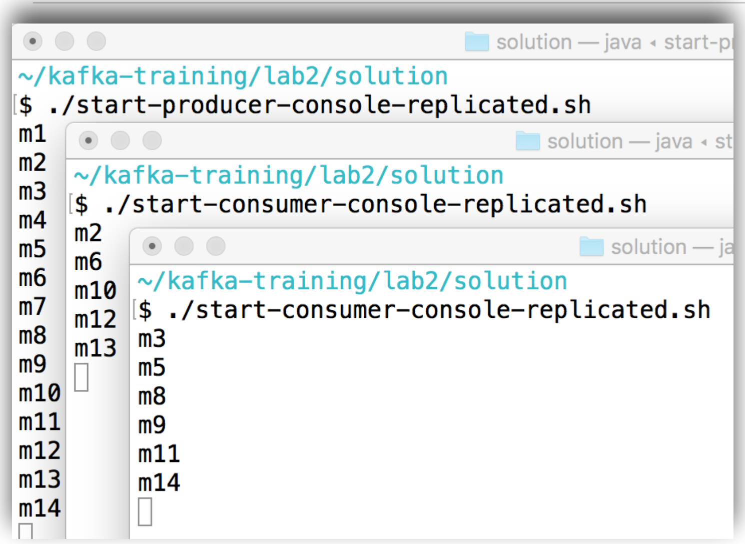
```
$ ./start-consumer-console-replicated.sh
m3
m5
```

Notice the first consumer gets messages m3 and m5.

2nd Kafka Consumer gets m2, m6

```
$ ./start-consumer-console-replicated.sh
m2
m6
```

Consumer Failover



```
~/kafka-training/lab2/solution
$ ./start-producer-console-replicated.sh
m1
m2
m3
m4
m5
m6
m7
m8
m9
m10
m11
m12
m13
m14
```

```
~/kafka-training/lab2/solution
$ ./start-consumer-console-replicated.sh
m2
m6
m10
m12
m13
```

```
~/kafka-training/lab2/solution
$ ./start-consumer-console-replicated.sh
m3
m5
m8
m9
m11
m14
```

- Kill one consumer
- Send seven more messages
- Load is spread to remaining consumers
- Failover WORK!

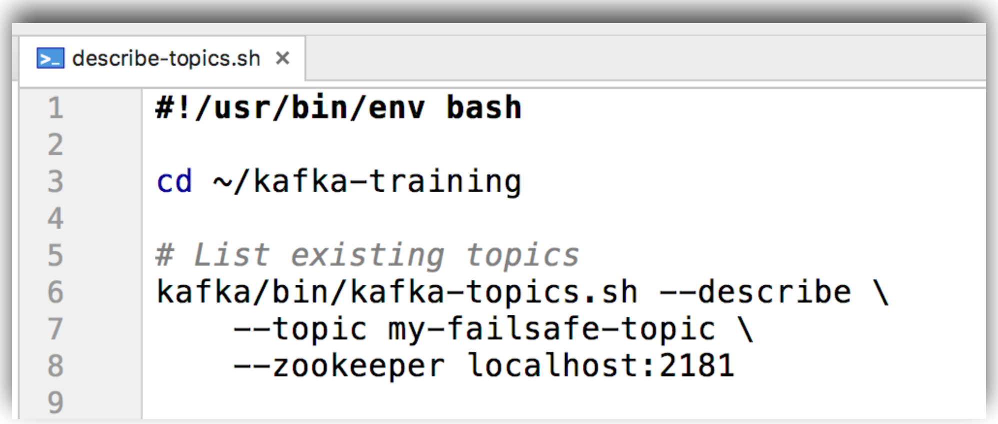
Next, let’s demonstrate consumer failover by killing one of the consumers and sending seven more messages. Kafka should divide up the work to the consumers that are running.
First, kill the third consumer (CTRL-C in the consumer terminal does the trick).
Now send seven more messages with the Kafka console-producer. Notice that the messages are spread evenly among the remaining consumers.

The first consumer got m8, m9, m11 and m14.

The second consumer got m10, m12, and m13.

We killed one consumer, sent seven more messages, and saw Kafka spread the load to remaining consumers. Kafka consumer failover works!

Create Kafka Describe Topic


describe-topics.sh

```
#!/usr/bin/env bash

cd ~/kafka-training

# List existing topics
kafka/bin/kafka-topics.sh --describe \
    --topic my-failsafe-topic \
    --zookeeper localhost:2181

```

- —describe will show list partitions, ISRs, and partition leadership

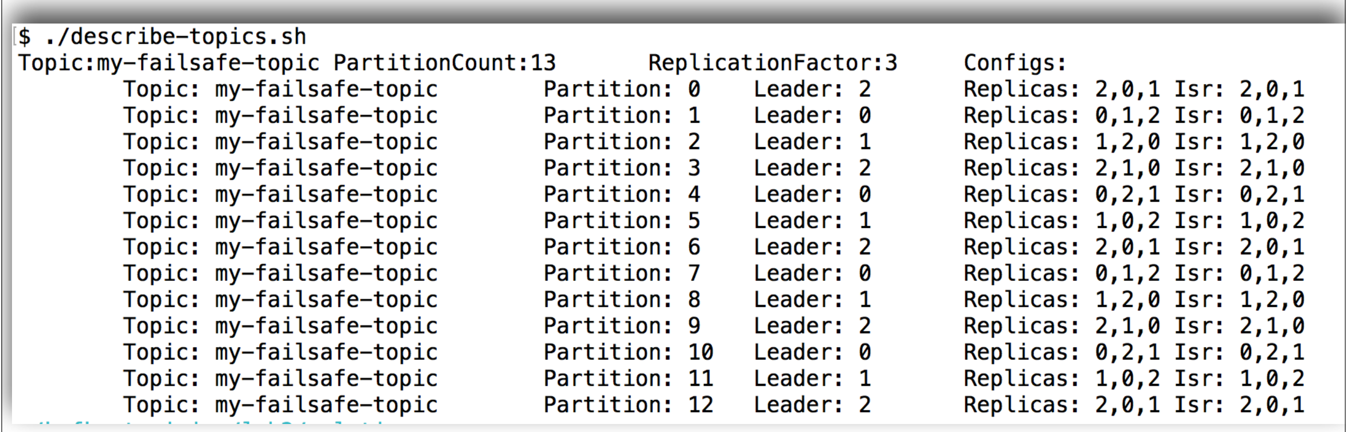
You can use kafka-topics.sh to see how the Kafka topic is laid out among the Kafka brokers. The ---describe will show partitions, ISRs, and broker partition leadership.

Create script ~/kafka-training/lab2/describe-topics.sh as follows:

```
#!/usr/bin/env bash
cd ~/kafka-training
# List existing topics
kafka/bin/kafka-topics.sh --describe \
    --topic my-failsafe-topic \
    --zookeeper localhost:2181
```

Let’s run kafka-topics.sh --describe and see the topology of our my-failsafe-topic.

Use Describe Topics

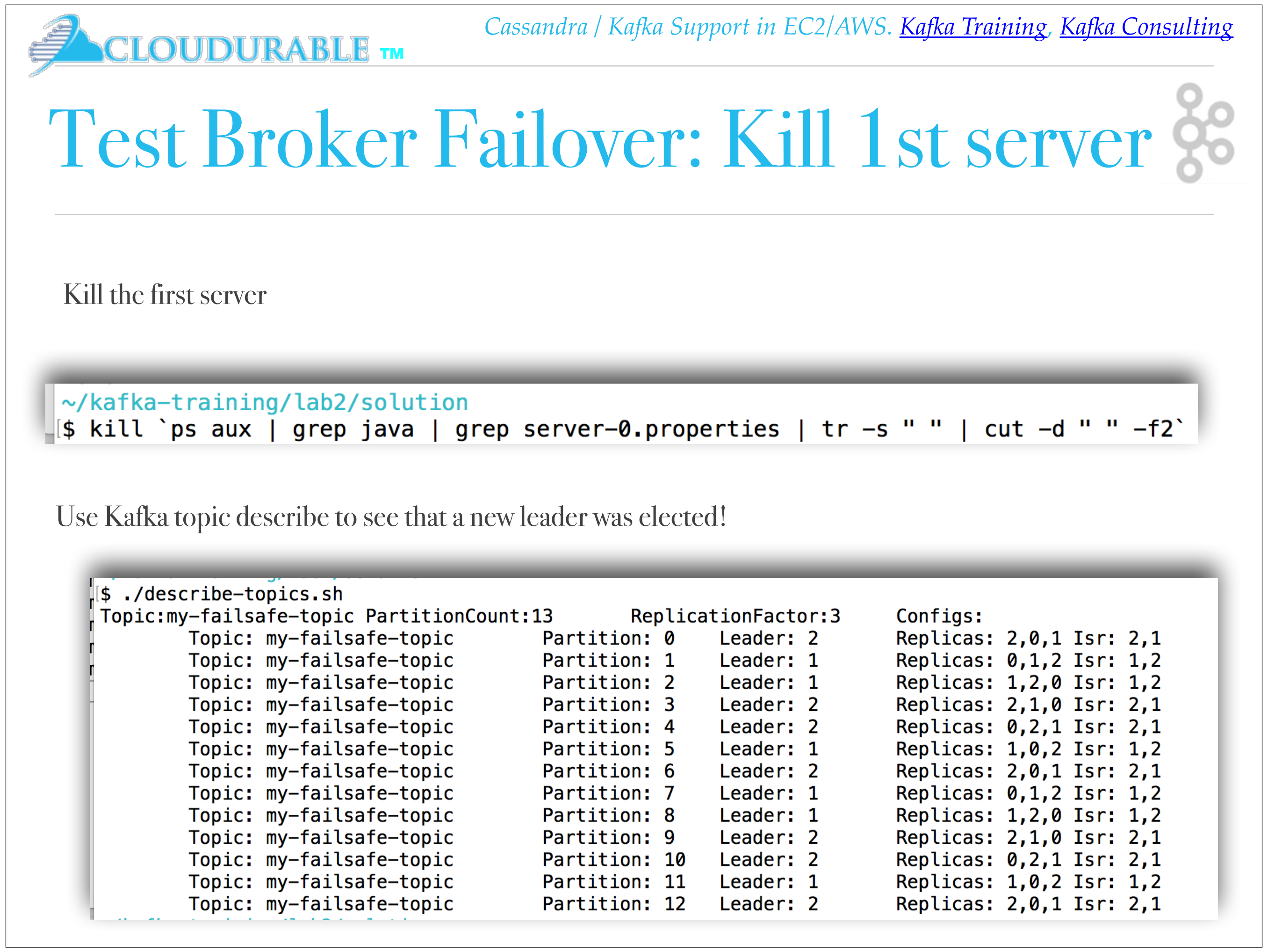


```
$ ./describe-topics.sh
Topic:my-failsafe-topic PartitionCount:13       ReplicationFactor:3     Configs:
        Topic: my-failsafe-topic        Partition: 0    Leader: 2       Replicas: 2,0,1 Isr: 2,0,1
        Topic: my-failsafe-topic        Partition: 1    Leader: 0       Replicas: 0,1,2 Isr: 0,1,2
        Topic: my-failsafe-topic        Partition: 2    Leader: 1       Replicas: 1,2,0 Isr: 1,2,0
        Topic: my-failsafe-topic        Partition: 3    Leader: 2       Replicas: 2,1,0 Isr: 2,1,0
        Topic: my-failsafe-topic        Partition: 4    Leader: 0       Replicas: 0,2,1 Isr: 0,2,1
        Topic: my-failsafe-topic        Partition: 5    Leader: 1       Replicas: 1,0,2 Isr: 1,0,2
        Topic: my-failsafe-topic        Partition: 6    Leader: 2       Replicas: 2,0,1 Isr: 2,0,1
        Topic: my-failsafe-topic        Partition: 7    Leader: 0       Replicas: 0,1,2 Isr: 0,1,2
        Topic: my-failsafe-topic        Partition: 8    Leader: 1       Replicas: 1,2,0 Isr: 1,2,0
        Topic: my-failsafe-topic        Partition: 9    Leader: 2       Replicas: 2,1,0 Isr: 2,1,0
        Topic: my-failsafe-topic        Partition: 10   Leader: 0       Replicas: 0,2,1 Isr: 0,2,1
        Topic: my-failsafe-topic        Partition: 11   Leader: 1       Replicas: 1,0,2 Isr: 1,0,2
        Topic: my-failsafe-topic        Partition: 12   Leader: 2       Replicas: 2,0,1 Isr: 2,0,1
```

- Lists which broker owns (leader of) which partition
- Lists Replicas and ISR (replicas that are up to date)
- Notice there are 13 topics

We are going to lists which broker owns (leader of) which partition, and list replicas and ISRs of each partition. ISRs are replicas that are up to date. Remember there are 13 topics.

Notice how each broker gets a share of the partitions as leaders and followers. Also, see how Kafka replicates the partitions on each broker.



Test Broker Failover: Kill 1st server

Kill the first server

```
~/kafka-training/lab2/solution
$ kill `ps aux | grep java | grep server-0.properties | tr -s " " | cut -d " " -f2`
```

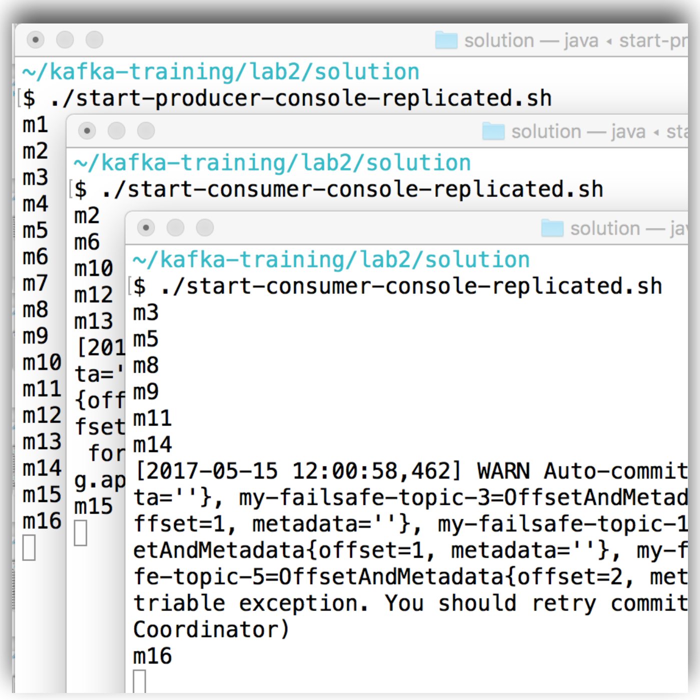
Use Kafka topic describe to see that a new leader was elected!

```
$ ./describe-topics.sh
Topic:my-failsafe-topic PartitionCount:13       ReplicationFactor:3     Configs:
        Topic: my-failsafe-topic        Partition: 0    Leader: 2       Replicas: 2,0,1 Isr: 2,1
        Topic: my-failsafe-topic        Partition: 1    Leader: 1       Replicas: 0,1,2 Isr: 1,2
        Topic: my-failsafe-topic        Partition: 2    Leader: 1       Replicas: 1,2,0 Isr: 1,2
        Topic: my-failsafe-topic        Partition: 3    Leader: 2       Replicas: 2,1,0 Isr: 2,1
        Topic: my-failsafe-topic        Partition: 4    Leader: 2       Replicas: 0,2,1 Isr: 2,1
        Topic: my-failsafe-topic        Partition: 5    Leader: 1       Replicas: 1,0,2 Isr: 1,2
        Topic: my-failsafe-topic        Partition: 6    Leader: 2       Replicas: 2,0,1 Isr: 2,1
        Topic: my-failsafe-topic        Partition: 7    Leader: 1       Replicas: 0,1,2 Isr: 1,2
        Topic: my-failsafe-topic        Partition: 8    Leader: 1       Replicas: 1,2,0 Isr: 1,2
        Topic: my-failsafe-topic        Partition: 9    Leader: 2       Replicas: 2,1,0 Isr: 2,1
        Topic: my-failsafe-topic        Partition: 10   Leader: 2       Replicas: 0,2,1 Isr: 2,1
        Topic: my-failsafe-topic        Partition: 11   Leader: 1       Replicas: 1,0,2 Isr: 1,2
        Topic: my-failsafe-topic        Partition: 12   Leader: 2       Replicas: 2,0,1 Isr: 2,1
```

You can stop the first broker by hitting CTRL-C in the broker terminal or by running the above command. Now that the first Kafka broker has stopped, let's use Kafka topics describe to see that new leaders were elected!

Notice how Kafka spreads the leadership over the 2nd and 3rd Kafka brokers.

Show Broker Failover Worked



```
~/kafka-training/lab2/solution
$ ./start-producer-console-replicated.sh
m1
m2
m3
m4
m5
m6
m7
m8
m9
m10
m11
m12
m13
m14
m15
m16
```

```
~/kafka-training/lab2/solution
$ ./start-consumer-console-replicated.sh
m2
m6
m10
m12
m13
[201
ta='
{off
fset
 for
g.ap
m15
```

```
~/kafka-training/lab2/solution
$ ./start-consumer-console-replicated.sh
m3
m5
m8
m9
m11
m14
[2017-05-15 12:00:58,462] WARN Auto-commit
ta=''}, my-failsafe-topic-3=OffsetAndMetad
ffset=1, metadata=''}, my-failsafe-topic-1
etAndMetadata{offset=1, metadata=''}, my-f
fe-topic-5=OffsetAndMetadata{offset=2, met
triable exception. You should retry commit
Coordinator)
m16
```

- Send two more messages from the producer
- Notice that the consumer gets the messages
- Broker Failover WORKS!

Let's prove that failover worked by sending two more messages from the producer console.
Then notice if the consumers still get the messages. Send the message m15 and m16.
Notice that the messages are spread evenly among the remaining live consumers.
The first Kafka broker gets m16. The 1st Kafka broker gets m15.

? Kafka Cluster Review

- Why did the three consumers not load share the messages at first?
- How did we demonstrate failover for consumers?
- How did we demonstrate failover for producers?
- What tool and option did we use to show ownership of partitions and the ISRs?

Why did the three consumers not load share the messages at first?
They did not load share at first because they were each in a different consumer group. Consumer groups each subscribe to a topic and maintain their own offsets per partition in that topic.

How did we demonstrate failover for consumers?
We shut a consumer down. Then we sent more messages. We observed Kafka spreading messages to the remaining cluster.

How did we show failover for producers?
We didn't. We showed failover for Kafka brokers by shutting one down, then using the producer console to send two more messages. Then we saw that the producer used the remaining Kafka brokers. Those Kafka brokers then delivered the messages to the live consumers.

What tool and option did we use to show ownership of partitions and the ISRs?
We used kafka-topics.sh using the --describe option.

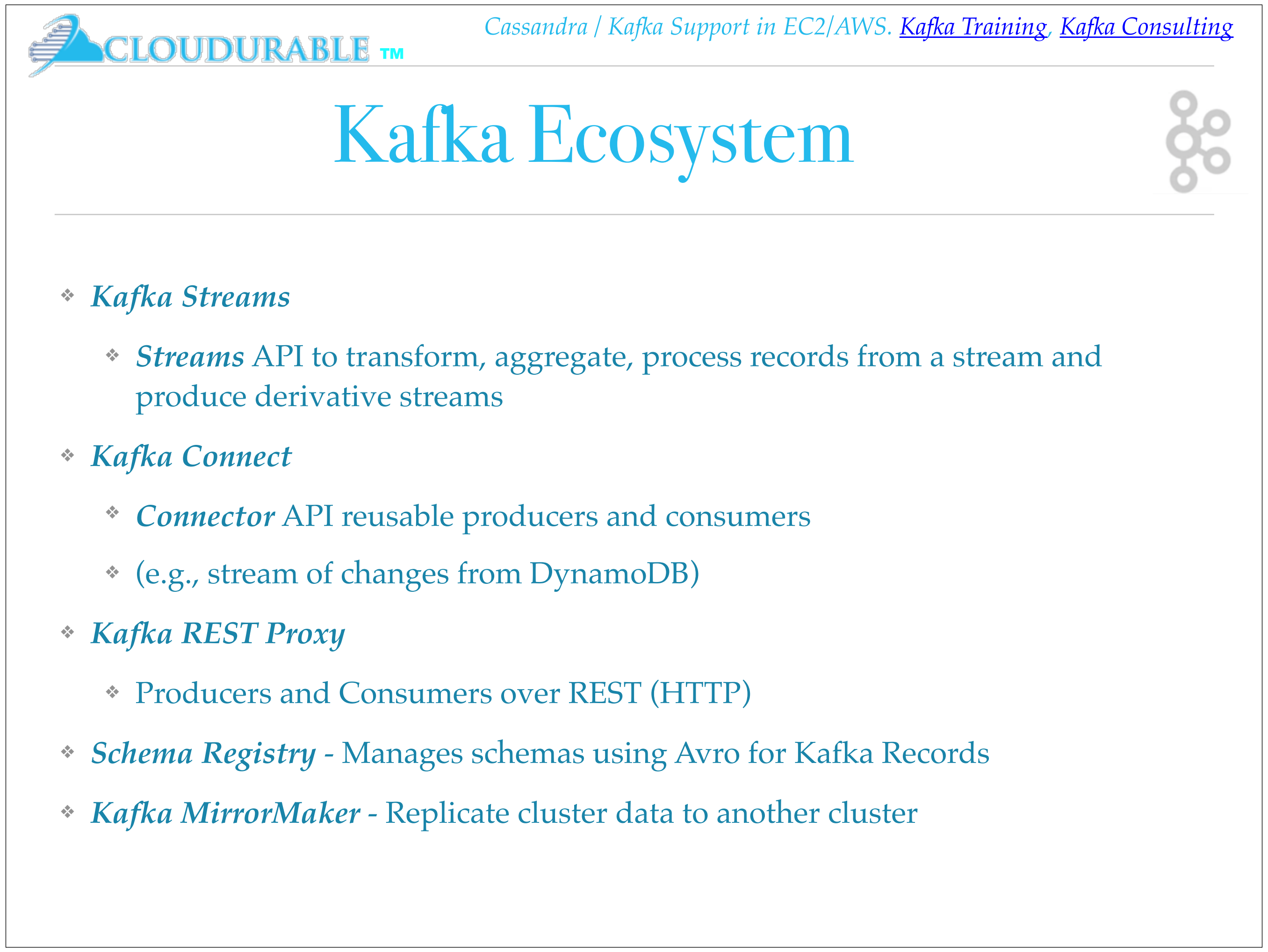
Use Kafka to send and receive messages

Lab 2 Use Kafka multiple nodes

Use a Kafka Cluster to replicate a Kafka topic log

Kafka Ecosystem

Kafka Connect
Kafka Streaming
Kafka Schema Registry
Kafka REST



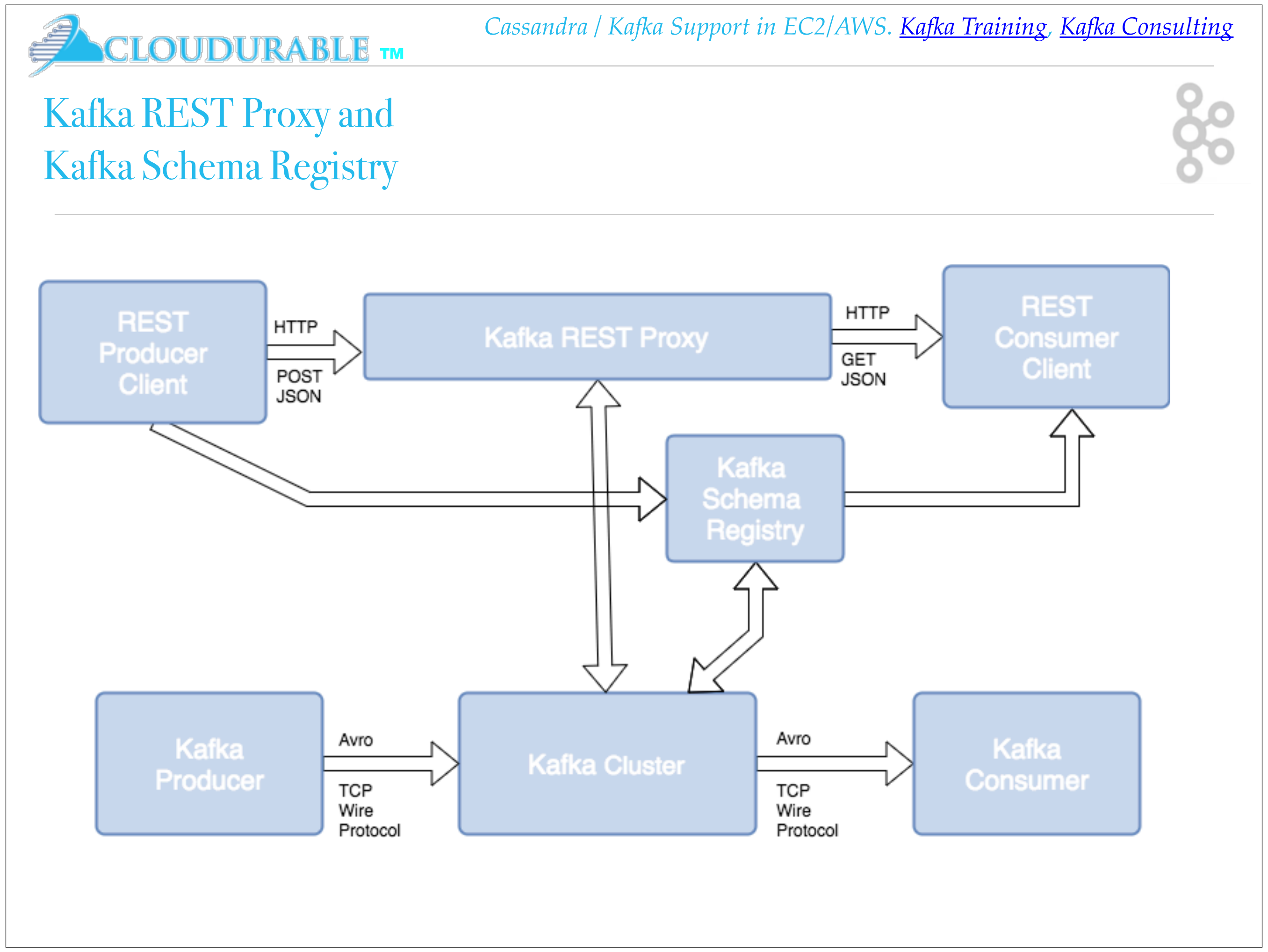
Kafka Ecosystem

- ***Kafka Streams***
  - ***Streams*** API to transform, aggregate, process records from a stream and produce derivative streams
- ***Kafka Connect***
  - ***Connector*** API reusable producers and consumers
  - (e.g., stream of changes from DynamoDB)
- ***Kafka REST Proxy***
  - Producers and Consumers over REST (HTTP)
- ***Schema Registry*** - Manages schemas using Avro for Kafka Records
- ***Kafka MirrorMaker*** - Replicate cluster data to another cluster

The core of Kafka is the brokers, topics, logs, partitions, and cluster. The core also consists of related tools like MirrorMaker. The aforementioned is Kafka as it exists in Apache.

The Kafka ecosystem consists of Kafka Core, Kafka Streams, Kafka Connect, Kafka REST Proxy, and the Schema Registry. Most of the additional pieces of the Kafka ecosystem comes from Confluent and is not part of Apache.

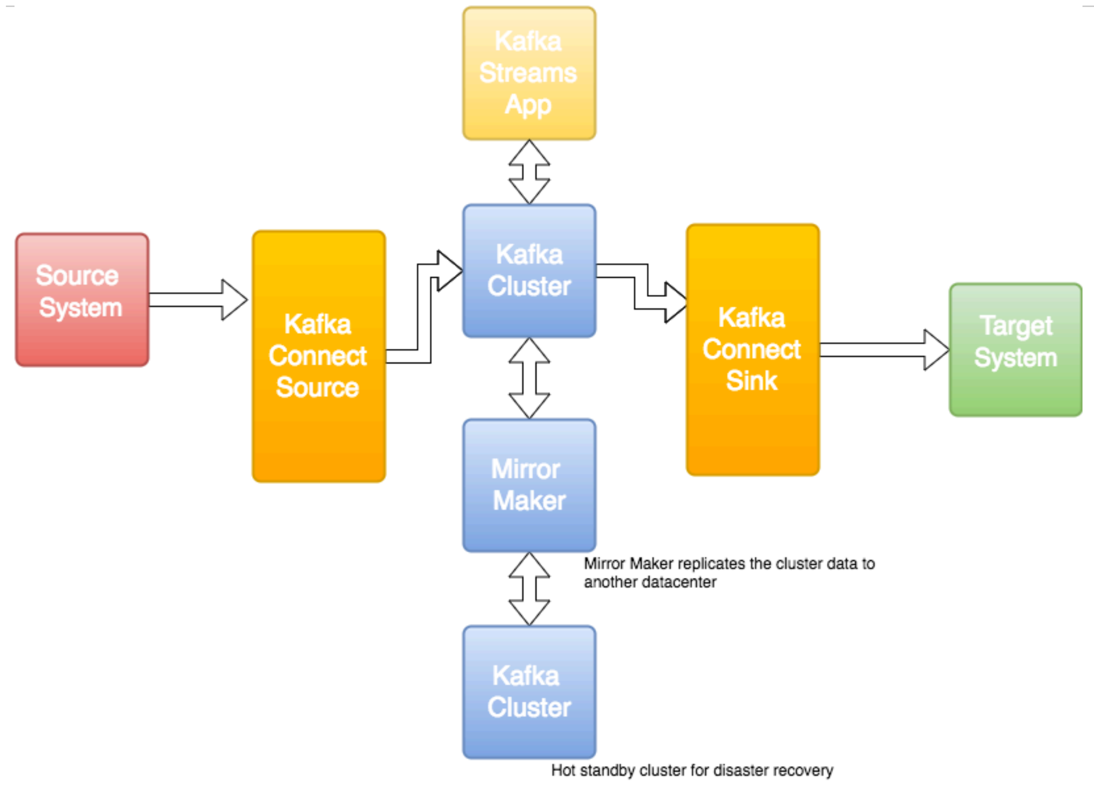
Kafka Stream is the Streams API to transform, aggregate, and process records from a stream and produces derivative streams. Kafka Connect is the connector API to create reusable producers and consumers (e.g., stream of changes from DynamoDB). The Kafka REST Proxy is used to producers and consumer over REST (HTTP). The Schema Registry manages schemas using Avro for Kafka records. The Kafka MirrorMaker is used to replicate cluster data to another cluster.

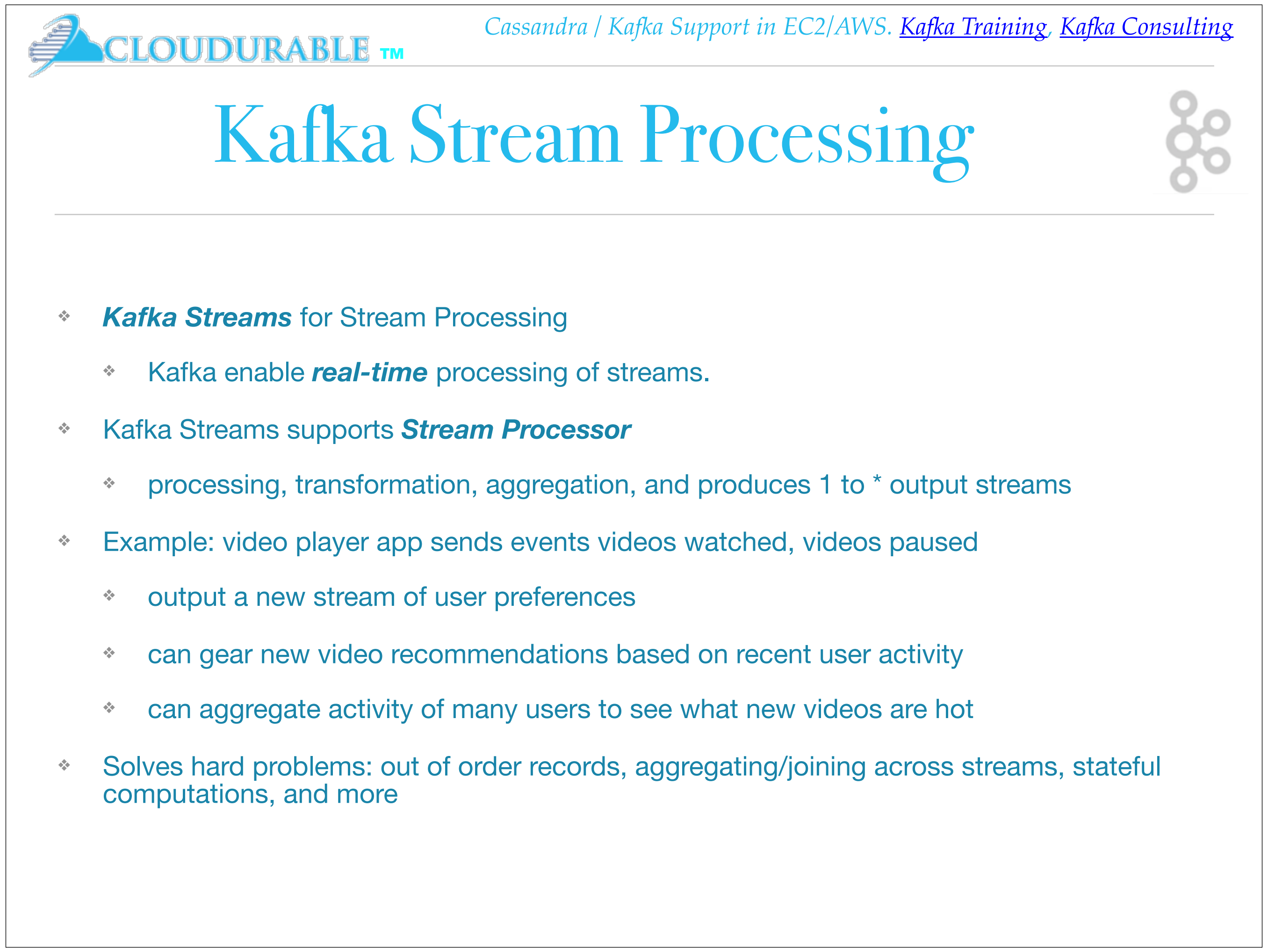


Kafka REST Proxy and Kafka Schema Registry

Kafka REST Proxy allows easy integration.
Schema validation of Kafka records is handled by Kafka schema registry.

Kafka Ecosystem





Kafka Stream Processing

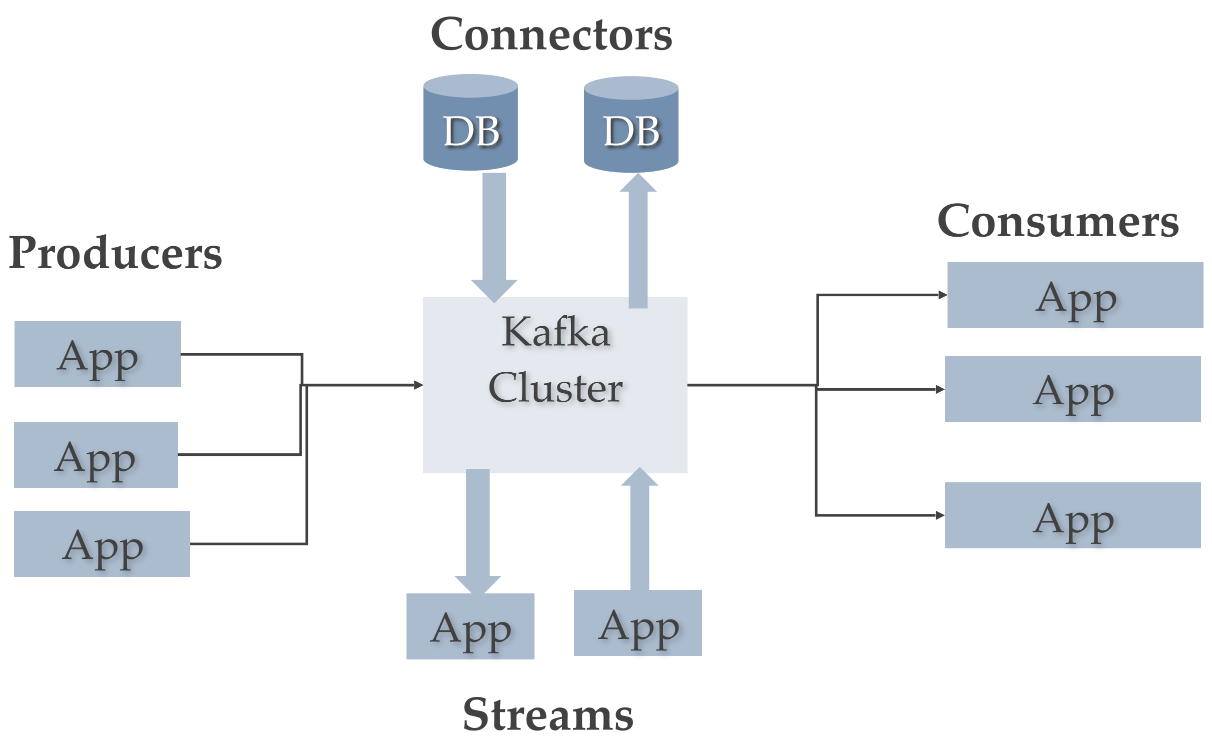
- ***Kafka Streams*** for Stream Processing
    - Kafka enable ***real-time*** processing of streams.
- Kafka Streams supports ***Stream Processor***
    - processing, transformation, aggregation, and produces 1 to * output streams
- Example: video player app sends events videos watched, videos paused
    - output a new stream of user preferences
    - can gear new video recommendations based on recent user activity
    - can aggregate activity of many users to see what new videos are hot
- Solves hard problems: out of order records, aggregating/joining across streams, stateful computations, and more

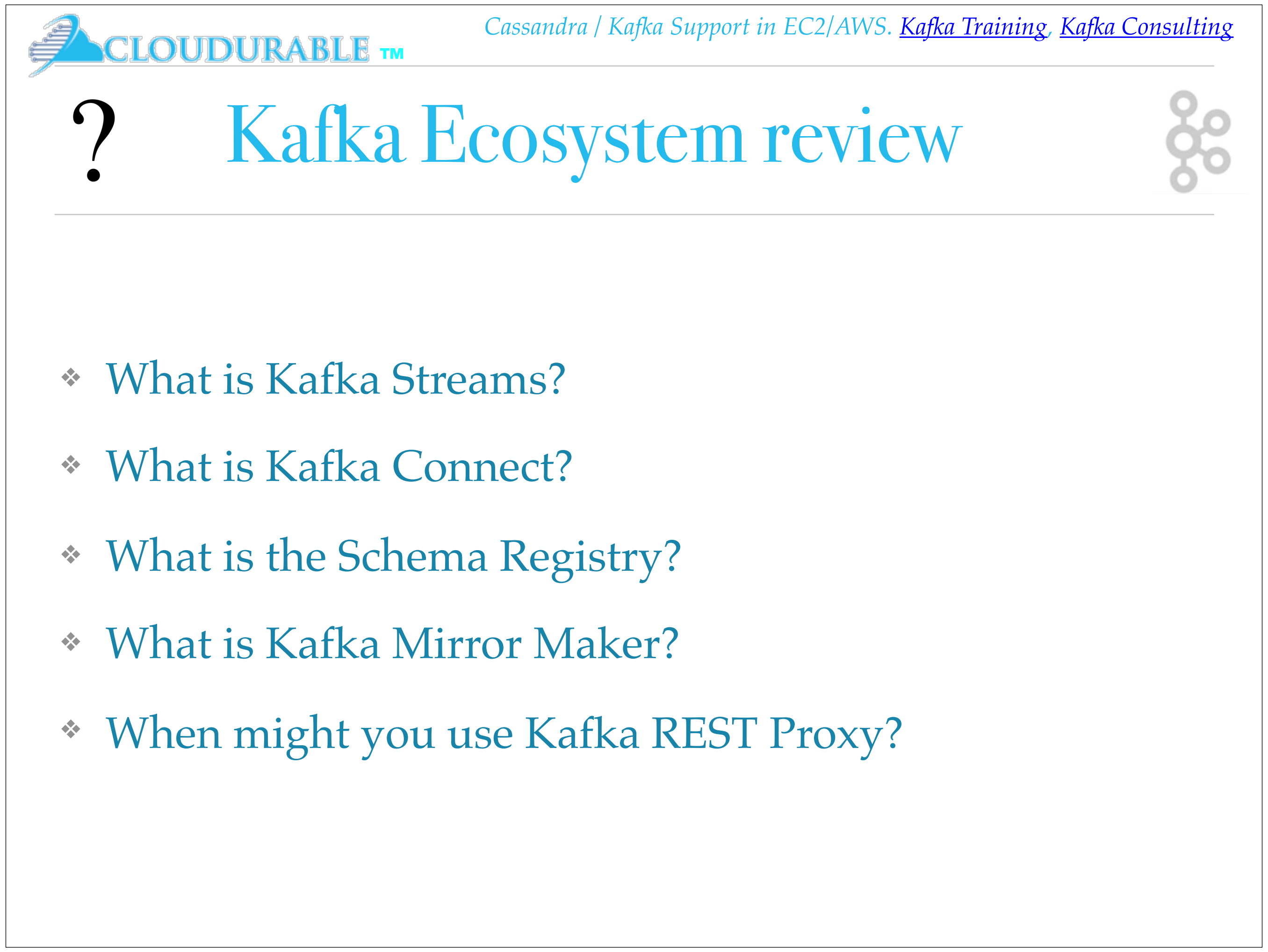
Kafka Streams for Stream Processing

The Kafka Stream API builds on core Kafka primitives and has a life of its own.
Kafka Streams enable real-time processing of streams.
Kafka Streams supports stream processor.
A stream processor takes continual streams of records from input topics, performs some processing, transformation, aggregation on input, and produces one or more output streams
For example, a video player application might take an input stream of events of videos watched, and videos paused, and output a stream of user preferences and then gear new video recommendations based on recent user activity or aggregate activity of many users to see what new videos are hot.

Kafka Stream API solves hard problems with out of order records, aggregating across multiple streams, joining data from multiple streams, allowing for stateful computations, and more.

Kafka Connectors and Streams





? Kafka Ecosystem review

- What is Kafka Streams?
- What is Kafka Connect?
- What is the Schema Registry?
- What is Kafka Mirror Maker?
- When might you use Kafka REST Proxy?

What is Kafka Streams?
Kafka Streams enable real-time processing of streams. It can aggregate across multiple streams, joining data from multiple streams, allowing for stateful computations, and more.

What is Kafka Connect?
Kafka Connect is the connector API to create reusable producers and consumers (e.g., stream of changes from DynamoDB).
Kafka Connect Sources are sources of records. Kafka Connect Sinks are a destination for records.

What is the Schema Registry?
The Schema Registry manages schemas using Avro for Kafka records.

What is Kafka Mirror Maker?
The Kafka MirrorMaker is used to replicate cluster data to another cluster.

When might you use Kafka REST Proxy?
The Kafka REST Proxy is used to producers and consumer over REST (HTTP). You could use it for easy integration of existing code bases.

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)