Confluent Certified Administrator for Apache Kafka (CCAAK) - Complete Study Material

Executive Summary

The Confluent Certified Administrator for Apache Kafka (CCAAK) certification validates the critical skills required to configure, deploy, monitor, manage, and support Apache Kafka clusters in production environments. This comprehensive study guide provides in-depth coverage of all exam topics, hands-on exercises, best practices, and preparation strategies to help you successfully pass the certification exam.

Exam Overview:

- **Duration:** 90 minutes
- **Number of Questions:** 60 multiple-choice and multiple-select questions
- **Cost:** $150 USD
- **Passing Score:** Pass/Fail (no specific percentage disclosed)
- **Validity:** 2 years
- **Prerequisites:** 6-12 months of hands-on experience with Apache Kafka recommended

Table of Contents

Section 1: Apache Kafka Fundamentals (15%)

1.1 Core Kafka Architecture

1.1.1 Kafka Components

Brokers

Kafka brokers are the backbone of a Kafka cluster. Each broker is a server that handles records from producers, assigns offsets, stores data on disk, and responds to consumer requests. Key characteristics:

- Each broker is identified by a unique `broker.id`
- Brokers maintain metadata about topics, partitions, and replicas
- A broker can host multiple partition replicas (both leaders and followers)
- Brokers communicate with each other to replicate data and elect leaders

Topics and Partitions

Topics are logical channels for organizing messages, while partitions enable parallelism and scalability:

- Topics are divided into one or more partitions
- Each partition is an ordered, immutable sequence of records
- Partitions enable parallel processing by distributing data across multiple brokers
- The number of partitions determines the maximum parallelism for consumers
- Partition assignment follows a round-robin distribution by default

Producers

Producers publish data to Kafka topics:

- Producers choose which partition to send messages to (via partitioner)
- Can send messages synchronously or asynchronously
- Support batching for improved throughput
- Configure acknowledgment levels (acks) for durability guarantees

Consumers and Consumer Groups

Consumers read data from Kafka topics, organized into consumer groups for scalability:

- Each consumer in a group is assigned exclusive partitions
- Consumer groups enable horizontal scaling of message processing
- Kafka maintains consumer offsets to track processing progress
- Rebalancing occurs when consumers join or leave the group

1.1.2 Data Durability and Replication

Replication Mechanism

Kafka ensures data durability through replication:

- **Replication Factor:** Number of copies of each partition across brokers
- **Leader Replica:** Handles all read and write requests for a partition
- **Follower Replicas:** Replicate data from the leader asynchronously
- **In-Sync Replicas (ISR):** Replicas that are fully caught up with the leader

In-Sync Replicas (ISR)

ISR is critical for data consistency and availability:

```
Key ISR Concepts:
- A replica is in-sync if it's within replica.lag.time.max.ms of the leader
- Only ISR members can be elected as the new leader
- Producers with acks=all wait for all ISR to acknowledge writes
- min.insync.replicas defines minimum ISR count for write acceptance
```

Configuration Parameters:

```
# Replication settings
replication.factor=3
min.insync.replicas=2
replica.lag.time.max.ms=10000

# Producer acknowledgment
acks=all  # Wait for all in-sync replicas
```

Best Practices:

- Set replication factor ≥ 3 for production environments

- Configure min.insync.replicas = replication factor - 1
- Use acks=all for critical data requiring durability guarantees

1.1.3 KRaft vs ZooKeeper

ZooKeeper Mode (Legacy)

Historically, Kafka relied on ZooKeeper for metadata management:

- ZooKeeper stores cluster metadata, broker information, and topic configurations
- Controller election and leader election managed through ZooKeeper
- Adds operational complexity with separate service to manage

KRaft Mode (New Standard)

Apache Kafka Raft (KRaft) eliminates ZooKeeper dependency:

```
KRaft Advantages:
√ Simplified architecture - metadata managed within Kafka
√ Improved scalability - handles larger clusters with less latency
√ Enhanced resilience - Raft protocol for strong consistency
√ Faster metadata propagation and controller operations
√ Easier deployment and maintenance
```

Key Differences:

Aspect	ZooKeeper Mode	KRaft Mode
Metadata Storage	External ZooKeeper ensemble	Internal Kafka controllers
Controller Selection	ZooKeeper-based election	Raft-based quorum election
Scalability	Limited by ZooKeeper	Improved metadata handling
Operational Complexity	Higher (two systems)	Lower (single system)
Metadata Updates	Through ZooKeeper	Event-based Raft protocol

Migration Considerations:

- KRaft is GA in Kafka 3.3+ and recommended for new deployments
- Downgrade restrictions apply after metadata version updates
- Rolling upgrade process required for existing ZooKeeper clusters

Section 2: Apache Kafka Security (15%)

2.1 Authentication Mechanisms

2.1.1 SSL/TLS Authentication

SSL provides encryption and certificate-based authentication:

Configuration Steps:

1. **Generate Certificates:**

```
# Generate keystore for broker
keytool -keystore kafka.server.keystore.jks -alias localhost \
  -validity 365 -genkey -keyalg RSA

# Generate CA certificate
openssl req -new -x509 -keyout ca-key -out ca-cert -days 365

# Sign broker certificate
keytool -keystore kafka.server.keystore.jks -alias localhost \
  -certreq -file cert-file
openssl x509 -req -CA ca-cert -CAkey ca-key -in cert-file \
  -out cert-signed -days 365 -CAcreateserial
```

2. **Broker Configuration:**

```
# server.properties
listeners=SSL://hostname:9093
security.inter.broker.protocol=SSL
ssl.keystore.location=/var/private/ssl/kafka.server.keystore.jks
ssl.keystore.password=keystore_password
ssl.key.password=key_password
ssl.truststore.location=/var/private/ssl/kafka.server.truststore.jks
ssl.truststore.password=truststore_password
ssl.client.auth=required
```

3. **Client Configuration:**

```
# Producer/Consumer properties
security.protocol=SSL
ssl.truststore.location=/var/private/ssl/kafka.client.truststore.jks
ssl.truststore.password=truststore_password
ssl.keystore.location=/var/private/ssl/kafka.client.keystore.jks
ssl.keystore.password=keystore_password
ssl.key.password=key_password
```

2.1.2 SASL Authentication

SASL (Simple Authentication and Security Layer) supports multiple authentication mechanisms:

SASL/PLAIN (Simple Username/Password)

```
# server.properties
listeners=SASL_PLAINTEXT://hostname:9092
security.inter.broker.protocol=SASL_PLAINTEXT
sasl.mechanism.inter.broker.protocol=PLAIN
sasl.enabled.mechanisms=PLAIN

# JAAS configuration
KafkaServer {
  org.apache.kafka.common.security.plain.PlainLoginModule required
  username="admin"
  password="admin-secret"
  user_admin="admin-secret"
  user_alice="alice-secret";
};
```

SASL/SCRAM (Salted Challenge Response Authentication)

Most secure option for username/password authentication:

```
# Create SCRAM credentials
kafka-configs.sh --bootstrap-server localhost:9092 \
  --alter --add-config 'SCRAM-SHA-512=[password=alice-secret]' \
  --entity-type users --entity-name alice

# server.properties
listeners=SASL_SSL://hostname:9093
```

```
security.inter.broker.protocol=SASL_SSL
sasl.mechanism.inter.broker.protocol=SCRAM-SHA-512
sasl.enabled.mechanisms=SCRAM-SHA-512

# JAAS configuration
KafkaServer {
  org.apache.kafka.common.security.scram.ScramLoginModule required
  username="admin"
  password="admin-secret";
};
```

SASL/GSSAPI (Kerberos)

Enterprise integration with existing Kerberos infrastructure:

```
# server.properties
listeners=SASL_PLAINTEXT://hostname:9092
security.inter.broker.protocol=SASL_PLAINTEXT
sasl.mechanism.inter.broker.protocol=GSSAPI
sasl.enabled.mechanisms=GSSAPI
sasl.kerberos.service.name=kafka
```

SASL_SSL vs SSL:

Protocol	Encryption	Authentication	Use Case
SSL	TLS	Certificate-based	Certificate infrastructure exists
SASL_PLAINTEXT	None	SASL mechanisms	Development only
SASL_SSL	TLS	SASL mechanisms	Integrate with Active Directory/LDAP

2.2 Authorization with ACLs

Access Control Lists (ACLs) provide fine-grained authorization:

ACL Structure:

```
Principal P is [Allowed|Denied] Operation O on Resource R from Host H
```

Common Operations:

- **Read:** Consume from topic
- **Write:** Produce to topic
- **Create:** Create topics
- **Delete:** Delete topics
- **Describe:** View topic metadata
- **Alter:** Modify topic configuration

Managing ACLs:

```
# Grant read access to topic
kafka-acls.sh --bootstrap-server localhost:9092 \
  --add --allow-principal User:alice \
  --operation Read --topic orders

# Grant write access to topic
kafka-acls.sh --bootstrap-server localhost:9092 \
  --add --allow-principal User:bob \
  --operation Write --topic orders

# Grant access to consumer group
kafka-acls.sh --bootstrap-server localhost:9092 \
  --add --allow-principal User:alice \
  --operation Read --group order-processors

# List ACLs
kafka-acls.sh --bootstrap-server localhost:9092 \
  --list --topic orders

# Remove ACL
kafka-acls.sh --bootstrap-server localhost:9092 \
  --remove --allow-principal User:alice \
  --operation Read --topic orders
```

Best Practices:

- Enable ACLs with `authorizer.class.name=kafka.security.authorizer.AclAuthorizer`
- Set `allow.everyone.if.no.acl.found=false` for security by default

- Use wildcards carefully (e.g., `Topic:*` for all topics)
- Apply principle of least privilege
- Regularly audit ACL configurations

2.3 Encryption

Encryption in Transit (TLS)

```
# Enable TLS for all listeners
listeners=SSL://hostname:9093
ssl.enabled.protocols=TLSv1.2,TLSv1.3
ssl.cipher.suites=TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
```

Encryption at Rest

- Kafka doesn't provide native encryption at rest
- Use disk-level encryption (LUKS, dm-crypt, cloud provider encryption)
- Application-level encryption before producing to Kafka
- Consider Confluent Enterprise for additional encryption features

Section 3: Deployment Architecture (12%)

3.1 Cluster Topology Design

3.1.1 High Availability (HA) Considerations

Multi-Broker Setup:

```
Minimum Production Setup:
- 3 brokers minimum for fault tolerance
- Odd number of controller nodes for KRaft (3 or 5)
- Distribute brokers across availability zones/racks
- Replication factor ≥ 3 for critical topics
```

Rack Awareness:

```
# Configure rack awareness
broker.rack=rack1

# Ensures replicas distributed across racks
# Protects against rack-level failures
```

Network Configuration:

```
# Multiple listeners for internal/external access
listeners=INTERNAL://0.0.0.0:9092,EXTERNAL://0.0.0.0:9093
advertised.listeners=INTERNAL://broker1.internal:9092,EXTERNAL://broker1.public:9093
listener.security.protocol.map=INTERNAL:PLAINTEXT,EXTERNAL:SASL_SSL
inter.broker.listener.name=INTERNAL
```

3.1.2 Capacity Planning

Resource Requirements:

Component	Minimum	Recommended	Notes
CPU	4 cores	8-16 cores	I/O threads scale with cores
RAM	8 GB	32-64 GB	Page cache critical for performance
Disk	100 GB SSD	1+ TB SSD/NVMe	RAID 10 for durability
Network	1 Gbps	10 Gbps	Network often bottleneck

Partition Sizing:

```
Guidelines:
- Target: 2000-4000 partitions per broker
- Consider: retention * throughput = storage needed
- Balance: More partitions = higher parallelism but more overhead
- Monitor: End-to-end latency increases with partition count
```

Storage Calculations:

```
Storage = Daily Throughput × Retention Days × Replication Factor
Example: 1 TB/day × 7 days × 3 replicas = 21 TB raw storage
```

```
Add 20% overhead for compaction/operations
```

3.2 Disaster Recovery Strategies

3.2.1 Cross-Cluster Replication

MirrorMaker 2 (Recommended):

```
# MM2 Configuration
clusters = primary, secondary
primary.bootstrap.servers = primary-kafka:9092
secondary.bootstrap.servers = secondary-kafka:9092

# Replication flows
primary->secondary.enabled = true
primary->secondary.topics = orders.*, payments.*

# Consumer group offset sync
sync.group.offsets.enabled = true
sync.group.offsets.interval.seconds = 60

# Topic configuration replication
sync.topic.configs.enabled = true
```

MirrorMaker 2 Features:

- Replicates topics, ACLs, configurations, and consumer offsets
- Supports multi-cluster, bidirectional replication
- Automatic recovery from network failures
- Built on Kafka Connect framework for scalability

Disaster Recovery Patterns:

1. **Active-Passive:**

```
Primary Cluster (Active) ---> Secondary Cluster (Standby)
                MM2
- All traffic to primary
- Secondary for DR only
```

```
- Failover when primary unavailable
```

2. **Active-Active:**

```
Cluster A <---> Cluster B
       MM2 (bidirectional)
- Both clusters serve traffic
- Geographic distribution
- Requires careful conflict resolution
```

3.2.2 Backup and Recovery

Topic Backup:

```
# Export topic data
kafka-console-consumer --bootstrap-server localhost:9092 \
  --topic orders --from-beginning \
  --property print.key=true \
  --property print.timestamp=true > backup.txt

# Restore topic data
kafka-console-producer --bootstrap-server localhost:9092 \
  --topic orders \
  --property "parse.key=true" \
  --property "key.separator=:" < backup.txt
```

Configuration Backup:

```
# Export topic configurations
kafka-configs.sh --bootstrap-server localhost:9092 \
  --describe --entity-type topics --entity-name orders

# Export ACLs
kafka-acls.sh --bootstrap-server localhost:9092 \
  --list > acls-backup.txt
```

3.3 Stretched Clusters

Stretched clusters span multiple data centers for enhanced availability:

Benefits:

- Synchronous replication across sites
- No data loss during site failure
- Single cluster management

Challenges:

- Requires low-latency network (<5ms)
- Higher operational complexity
- Potential split-brain scenarios

Configuration:

```
# Configure rack awareness for DC distribution
broker.rack=dc1-rack1

# Ensure replicas span data centers
# Use replica placement constraints
```

Section 4: Kafka Connect (12%)

4.1 Kafka Connect Architecture

Kafka Connect is a framework for streaming data between Kafka and external systems:

Key Components:

- **Connectors:** Define where data should be copied to/from
- **Tasks:** Implement the actual data copying
- **Workers:** Execute connectors and tasks
- **Converters:** Handle data serialization/deserialization

Connector Types:

- **Source Connectors:** Import data from external systems into Kafka

- **Sink Connectors:** Export data from Kafka to external systems

4.2 Standalone vs Distributed Mode

4.2.1 Standalone Mode

Suitable for development and testing:

```
# standalone.properties
bootstrap.servers=localhost:9092
key.converter=org.apache.kafka.connect.json.JsonConverter
value.converter=org.apache.kafka.connect.json.JsonConverter
offset.storage.file.filename=/tmp/connect.offsets
```

Launch Standalone:

```
connect-standalone.sh standalone.properties \
  file-source-connector.properties \
  file-sink-connector.properties
```

Characteristics:

- Single worker process
- All tasks run on single node
- No fault tolerance
- Configuration via files
- Simple for development/testing

4.2.2 Distributed Mode

Recommended for production:

```
# distributed.properties
bootstrap.servers=kafka1:9092,kafka2:9092,kafka3:9092
group.id=connect-cluster

# Converter configuration
key.converter=org.apache.kafka.connect.json.JsonConverter
```

```
value.converter=org.apache.kafka.connect.json.JsonConverter

# Internal topics for coordination
config.storage.topic=connect-configs
config.storage.replication.factor=3
offset.storage.topic=connect-offsets
offset.storage.replication.factor=3
status.storage.topic=connect-status
status.storage.replication.factor=3

# REST API configuration
rest.port=8083
```

Launch Distributed Worker:

```
connect-distributed.sh distributed.properties
```

Distributed Mode Features:

- Multiple worker nodes in cluster
- Automatic load balancing
- Fault tolerance through task redistribution
- Dynamic scaling
- REST API for management
- Configuration stored in Kafka topics

4.3 Managing Connectors

REST API Operations:

```
# List available connector plugins
curl http://localhost:8083/connector-plugins

# Create connector
curl -X POST http://localhost:8083/connectors \
  -H "Content-Type: application/json" \
  -d '{
    "name": "file-source",
```

```
    "config": {
      "connector.class": "FileStreamSource",
      "tasks.max": "1",
      "file": "/tmp/input.txt",
      "topic": "connect-test"
    }
  }'

# List connectors
curl http://localhost:8083/connectors

# Get connector status
curl http://localhost:8083/connectors/file-source/status

# Get connector configuration
curl http://localhost:8083/connectors/file-source

# Update connector configuration
curl -X PUT http://localhost:8083/connectors/file-source/config \
  -H "Content-Type: application/json" \
  -d '{"tasks.max": "2"}'

# Pause connector
curl -X PUT http://localhost:8083/connectors/file-source/pause

# Resume connector
curl -X PUT http://localhost:8083/connectors/file-source/resume

# Restart connector
curl -X POST http://localhost:8083/connectors/file-source/restart

# Delete connector
curl -X DELETE http://localhost:8083/connectors/file-source
```

4.4 Common Connector Configurations

JDBC Source Connector:

```
{
  "name": "jdbc-source",
  "config": {
    "connector.class": "io.confluent.connect.jdbc.JdbcSourceConnector",
    "connection.url": "jdbc:postgresql://localhost:5432/mydb",
    "connection.user": "postgres",
    "connection.password": "password",
    "topic.prefix": "db-",
    "mode": "incrementing",
    "incrementing.column.name": "id",
    "poll.interval.ms": "5000",
    "tasks.max": "1"
  }
}
```

Elasticsearch Sink Connector:

```
{
  "name": "elasticsearch-sink",
  "config": {
    "connector.class": "io.confluent.connect.elasticsearch.ElasticsearchSinkConnector",
    "connection.url": "http://localhost:9200",
    "topics": "orders,payments",
    "key.ignore": "true",
    "schema.ignore": "true",
    "tasks.max": "2"
  }
}
```

4.5 Monitoring and Troubleshooting Connectors

Key Metrics:

```
Connector Metrics:
- connector-status: Health status of connector
- task-status: Status of individual tasks
- connector-total-task-count: Number of tasks
- source-record-poll-rate: Records polled from source (source connectors)
```

```
- sink-record-send-rate: Records sent to sink (sink connectors)
```

Common Issues:

1. **Task Failures:**

```
# Check task status
curl http://localhost:8083/connectors/jdbc-source/tasks/0/status

# Restart failed task
curl -X POST http://localhost:8083/connectors/jdbc-source/tasks/0/restart
```

2. **Offset Issues:**

```
# Reset connector offsets (distributed mode)
# Delete connector first, then reset offsets
kafka-consumer-groups.sh --bootstrap-server localhost:9092 \
  --group connect-jdbc-source --reset-offsets --to-earliest \
  --topic connect-offsets --execute
```

3. **Performance Tuning:**

```
{
  "tasks.max": "4",
  "batch.size": "1000",
  "max.poll.records": "500",
  "poll.interval.ms": "5000"
}
```

Section 5: Apache Kafka Cluster Configuration (22%)

5.1 Broker Configuration

5.1.1 Essential Broker Settings

Core Configuration (server.properties):

```
# Broker Identity
broker.id=1
node.id=1  # For KRaft mode

# Network Settings
listeners=PLAINTEXT://0.0.0.0:9092
advertised.listeners=PLAINTEXT://broker1.example.com:9092
listener.security.protocol.map=PLAINTEXT:PLAINTEXT
inter.broker.listener.name=PLAINTEXT

# Log Directories
log.dirs=/var/kafka-logs
num.network.threads=8
num.io.threads=16

# Replication Settings
default.replication.factor=3
min.insync.replicas=2
replica.fetch.min.bytes=1
replica.fetch.wait.max.ms=500
replica.lag.time.max.ms=10000

# Log Retention
log.retention.hours=168  # 7 days
log.retention.bytes=-1  # Unlimited
log.segment.bytes=1073741824  # 1 GB
log.cleanup.policy=delete
```

5.1.2 Performance Tuning Parameters

I/O and Network Threads:

```
# Adjust based on CPU cores and workload
num.network.threads=8  # Handles network requests
num.io.threads=16  # Handles disk I/O

# Socket buffer sizes
socket.send.buffer.bytes=102400  # 100 KB
socket.receive.buffer.bytes=102400
```

```
socket.request.max.bytes=104857600  # 100 MB
```

Message Batching:

```
# Producer batch settings
batch.size=16384  # 16 KB
linger.ms=10
compression.type=lz4

# Log flush settings (careful tuning needed)
log.flush.interval.messages=10000
log.flush.interval.ms=1000
```

Buffer and Memory:

```
# Replica fetcher settings
replica.socket.receive.buffer.bytes=65536

# Controller settings
controller.socket.timeout.ms=30000

# Network request memory
queued.max.requests=500
```

5.1.3 Durability vs Performance Tradeoffs

Maximum Durability:

```
# Producer
acks=all
enable.idempotence=true
max.in.flight.requests.per.connection=5

# Broker
min.insync.replicas=2
unclean.leader.election.enable=false
log.flush.interval.messages=1  # Sync every message (expensive!)
```

Maximum Throughput:

```
# Producer
acks=1
compression.type=lz4
batch.size=32768
linger.ms=100

# Broker
min.insync.replicas=1
num.replica.fetchers=4
replica.fetch.min.bytes=1048576
```

Balanced (Recommended):

```
# Producer
acks=all
compression.type=lz4
batch.size=16384
linger.ms=10
enable.idempotence=true

# Broker
min.insync.replicas=2
default.replication.factor=3
unclean.leader.election.enable=false
```

5.2 Topic Configuration

5.2.1 Creating and Managing Topics

Create Topic:

```
# Create topic with specific configuration
kafka-topics.sh --bootstrap-server localhost:9092 \
  --create --topic orders \
  --partitions 6 \
  --replication-factor 3 \
  --config retention.ms=604800000 \
  --config segment.bytes=536870912 \
  --config min.insync.replicas=2
```

```
# List topics
kafka-topics.sh --bootstrap-server localhost:9092 --list

# Describe topic
kafka-topics.sh --bootstrap-server localhost:9092 \
  --describe --topic orders
```

Modify Topic Configuration:

```
# Add partitions (cannot reduce)
kafka-topics.sh --bootstrap-server localhost:9092 \
  --alter --topic orders --partitions 12

# Change topic configuration
kafka-configs.sh --bootstrap-server localhost:9092 \
  --alter --entity-type topics --entity-name orders \
  --add-config retention.ms=1209600000

# Delete configuration override
kafka-configs.sh --bootstrap-server localhost:9092 \
  --alter --entity-type topics --entity-name orders \
  --delete-config retention.ms
```

5.2.2 Retention Policies

Time-Based Retention:

```
# Topic-level configuration
retention.ms=604800000  # 7 days in milliseconds
# Alternatives: retention.minutes, retention.hours

# Delete vs Compact
cleanup.policy=delete  # Delete old segments
# OR
cleanup.policy=compact  # Keep latest value per key
# OR
cleanup.policy=compact,delete  # Both
```

Size-Based Retention:

```
# Maximum size per partition
retention.bytes=1073741824  # 1 GB per partition
# Total topic size = retention.bytes × partition_count × replication_factor
```

Segment Configuration:

```
segment.bytes=1073741824  # 1 GB segment files
segment.ms=604800000  # Roll segment after 7 days
```

5.2.3 Log Compaction

Log compaction retains latest value for each key:

```
# Enable compaction
cleanup.policy=compact

# Compaction settings
min.cleanable.dirty.ratio=0.5  # 50% dirty before compaction
segment.ms=3600000  # 1 hour
delete.retention.ms=86400000  # 24 hours for tombstones
min.compaction.lag.ms=0
max.compaction.lag.ms=9223372036854775807
```

Use Cases:

- Database change capture (CDC)
- User profile updates
- Configuration management
- State stores for stream processing

Example:

```
Original Log:
Key: user1, Value: {name: "John", age: 25}
Key: user2, Value: {name: "Jane", age: 30}
Key: user1, Value: {name: "John", age: 26}
Key: user2, Value: null  # Tombstone
```

```
After Compaction:
Key: user1, Value: {name: "John", age: 26}
# user2 deleted after tombstone retention
```

5.3 Scaling Operations

5.3.1 Adding Brokers

```
# Step 1: Configure new broker
# Edit server.properties with new broker.id

# Step 2: Start new broker
kafka-server-start.sh config/server.properties

# Step 3: Verify broker joined cluster
kafka-broker-api-versions.sh --bootstrap-server localhost:9092

# Step 4: Reassign partitions to new broker (optional)
# Generate reassignment plan
kafka-reassign-partitions.sh --bootstrap-server localhost:9092 \
  --broker-list "1,2,3,4" --topics-to-move-json-file topics.json \
  --generate

# Execute reassignment
kafka-reassign-partitions.sh --bootstrap-server localhost:9092 \
  --reassignment-json-file reassignment.json --execute

# Verify reassignment
kafka-reassign-partitions.sh --bootstrap-server localhost:9092 \
  --reassignment-json-file reassignment.json --verify
```

5.3.2 Decommissioning Brokers

```
# Step 1: Move partitions off broker
# Create reassignment plan excluding broker

# Step 2: Execute reassignment and wait for completion
```

```
# Step 3: Controlled shutdown
kafka-server-stop.sh

# Alternative: Graceful shutdown via signal
kill -TERM <kafka-pid>
```

5.3.3 Partition Rebalancing

Automatic Rebalancing:

```
# Enable automatic leader rebalancing
auto.leader.rebalance.enable=true
leader.imbalance.check.interval.seconds=300
leader.imbalance.per.broker.percentage=10
```

Manual Rebalancing:

```
# Trigger preferred leader election
kafka-leader-election.sh --bootstrap-server localhost:9092 \
  --election-type PREFERRED --all-topic-partitions

# Use Cruise Control for advanced rebalancing (Confluent)
```

5.4 Cluster Upgrades

5.4.1 Rolling Upgrade Process

Best Practices:

1. **Prepare:**

```
# Backup configurations
cp -r config/ config-backup/

# Update inter.broker.protocol.version
inter.broker.protocol.version=3.6
log.message.format.version=3.6
```

2. **Upgrade Sequence:**

```
a. Upgrade controller last (identify with kafka-metadata.sh)
b. One broker at a time
c. Wait for ISR sync before proceeding
d. Monitor cluster health continuously
```

3. **Upgrade Steps per Broker:**

```
# 1. Controlled shutdown
kafka-server-stop.sh

# 2. Upgrade Kafka binaries
# Install new version

# 3. Start broker with new version
kafka-server-start.sh config/server.properties

# 4. Verify broker rejoined and caught up
kafka-broker-api-versions.sh --bootstrap-server localhost:9092
```

4. **Finalize Upgrade:**

```
# Update protocol versions (after all brokers upgraded)
kafka-configs.sh --bootstrap-server localhost:9092 \
  --alter --entity-type brokers --entity-default \
  --add-config inter.broker.protocol.version=3.7

# Restart brokers again with updated versions
```

5.4.2 Version Compatibility

Compatibility Matrix:

```
Client Version    Compatible Broker Versions
3.7               3.3 - 3.7
3.6               3.3 - 3.6
3.5               3.0 - 3.5
```

Downgrade Considerations:

- Possible before finalizing metadata version
- Not possible after metadata version update in KRaft
- Test thoroughly in staging environment[29]

5.5 Configuration Management

Dynamic Configuration:

```
# Broker-level configuration (no restart required)
kafka-configs.sh --bootstrap-server localhost:9092 \
  --alter --entity-type brokers --entity-name 1 \
  --add-config 'max.connections=2000'

# Cluster-wide defaults
kafka-configs.sh --bootstrap-server localhost:9092 \
  --alter --entity-type brokers --entity-default \
  --add-config 'log.retention.hours=336'

# List configurations
kafka-configs.sh --bootstrap-server localhost:9092 \
  --describe --entity-type brokers --entity-name 1
```

Section 6: Observability (10%)

6.1 JMX Metrics

Java Management Extensions (JMX) is the primary interface for Kafka metrics:

6.1.1 Enabling JMX

```
# Set JMX port when starting Kafka
export JMX_PORT=9999
kafka-server-start.sh config/server.properties

# Enable remote JMX (use cautiously)
export KAFKA_JMX_OPTS="-Dcom.sun.management.jmxremote \
  -Dcom.sun.management.jmxremote.port=9999 \
```

```
  -Dcom.sun.management.jmxremote.authenticate=false \
  -Dcom.sun.management.jmxremote.ssl=false"
```

6.1.2 Critical Broker Metrics

Under-Replicated Partitions:

```
MBean: kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions
Alert Threshold: > 0
Meaning: Partitions where follower replicas are behind leader
Action: Investigate network, disk, or broker issues
```

Offline Partitions:

```
MBean: kafka.controller:type=KafkaController,name=OfflinePartitionsCount
Alert Threshold: > 0
Meaning: Partitions without an active leader
Action: Critical - immediate investigation required
```

Active Controller Count:

```
MBean: kafka.controller:type=KafkaController,name=ActiveControllerCount
Alert Threshold: != 1
Meaning: Should always be exactly 1 in cluster
Action: Controller election issue if 0 or split-brain if > 1
```

Request Metrics:

```
# Request rate
kafka.network:type=RequestMetrics,name=RequestsPerSec,request={Produce|Fetch}

# Request latency
kafka.network:type=RequestMetrics,name=TotalTimeMs,request={Produce|Fetch}

# Queue time
kafka.network:type=RequestMetrics,name=RequestQueueTimeMs
```

Log Metrics:

```
# Log flush rate
kafka.log:type=LogFlushStats,name=LogFlushRateAndTimeMs

# Log size
kafka.log:type=Log,name=Size,topic={topic},partition={partition}
```

6.1.3 Producer/Consumer Metrics

Producer Metrics:

```
# Record send rate
kafka.producer:type=producer-metrics,client-id={client-id},attribute=record-send-rate

# Batch size average
kafka.producer:type=producer-metrics,client-id={client-id},attribute=batch-size-avg

# Compression ratio
kafka.producer:type=producer-metrics,client-id={client-id},attribute=compression-rate-
avg

# Request latency
kafka.producer:type=producer-metrics,client-id={client-id},attribute=request-latency-avg
```

Consumer Metrics:

```
# Lag (most critical consumer metric)
kafka.consumer:type=consumer-fetch-manager-metrics,client-id={client-
id},attribute=records-lag-max

# Fetch rate
kafka.consumer:type=consumer-fetch-manager-metrics,client-id={client-
id},attribute=fetch-rate

# Records consumed rate
kafka.consumer:type=consumer-fetch-manager-metrics,client-id={client-
id},attribute=records-consumed-rate
```

6.2 Monitoring Tools

6.2.1 Command-Line Tools

Check Consumer Lag:

```
kafka-consumer-groups.sh --bootstrap-server localhost:9092 \
  --describe --group order-processors

# Output shows:
# GROUP           TOPIC    PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG
# order-processors orders  0          1000            1050            50
```

Monitor Topic Metrics:

```
# Topic details
kafka-topics.sh --bootstrap-server localhost:9092 \
  --describe --topic orders

# Log segment information
kafka-log-dirs.sh --bootstrap-server localhost:9092 \
  --describe --broker-list 1,2,3 --topic-list orders
```

6.2.2 Monitoring Stack Integration

Prometheus + Grafana:

1. **JMX Exporter Configuration:**

```
# jmx_exporter_config.yml
lowercaseOutputName: true
rules:
  - pattern: kafka.server<type=(.+), name=(.+)><>Value
    name: kafka_server_$1_$2
  - pattern: kafka.controller<type=(.+), name=(.+)><>Value
    name: kafka_controller_$1_$2
```

2. **Start Kafka with JMX Exporter:**

```
export KAFKA_OPTS="-
javaagent:/opt/jmx_exporter/jmx_prometheus_javaagent.jar=7071:jmx_exporter_config.yml"
kafka-server-start.sh config/server.properties
```

3. **Prometheus Scrape Configuration:**

```
scrape_configs:
  - job_name: 'kafka'
    static_configs:
      - targets: ['kafka1:7071', 'kafka2:7071', 'kafka3:7071']
```

Confluent Control Center:

Confluent's enterprise tool for comprehensive cluster management:

- Real-time cluster monitoring dashboard
- Topic inspection and message browsing
- Consumer group monitoring with lag visualization
- Connector management interface
- ksqlDB query development
- Alerts and notifications
- Schema Registry integration

6.3 Key Performance Indicators (KPIs)

Availability KPIs:

```
√ Offline Partitions: 0
√ Under-Replicated Partitions: 0
√ Active Controller Count: 1
√ Broker Uptime: > 99.9%
```

Performance KPIs:

```
√ Producer Request Latency (p99): < 100ms
√ Consumer Lag: < 1000 messages per partition
√ Network Utilization: < 80%
√ Disk I/O Wait: < 20%
```

Resource KPIs:

```
✓ CPU Utilization: < 70% average
✓ Memory Usage: Sufficient for page cache
✓ Disk Usage: < 80% capacity
✓ Network Bandwidth: Headroom for spikes
```

6.4 Log Analysis

Broker Logs:

```
# Check server logs
tail -f /var/log/kafka/server.log

# Common log patterns to monitor:
# ISR shrinkage
grep "Shrinking ISR" server.log

# Leader elections
grep "Completed leader election" server.log

# Replication errors
grep "UnderReplicatedPartitions" server.log
```

Log Levels:

```
# Adjust logging in log4j.properties
log4j.logger.kafka=INFO
log4j.logger.kafka.controller=DEBUG
log4j.logger.kafka.network.RequestChannel=WARN
```

Section 7: Troubleshooting (15%)

7.1 Common Issues and Resolutions

7.1.1 Under-Replicated Partitions

Symptoms:

- `UnderReplicatedPartitions` metric > 0
- ISR list smaller than replication factor

Causes and Solutions:

1. **Slow Broker:**

```
# Check broker metrics
# Look for high CPU, disk I/O, or network usage

# Check replica lag
kafka-topics.sh --bootstrap-server localhost:9092 \
  --describe --under-replicated-partitions

# Solutions:
- Increase replica.fetch.min.bytes
- Add more disk I/O capacity
- Upgrade network infrastructure
- Reduce load on broker
```

2. **Network Issues:**

```
# Check network connectivity between brokers
ping broker2.example.com
traceroute broker2.example.com

# Solutions:
- Fix network configuration
- Increase replica.socket.timeout.ms
- Check firewall rules
```

3. **Disk Problems:**

```
# Check disk health
df -h
iostat -x 1

# Solutions:
- Free up disk space
- Replace failing disks
```

```
- Adjust log retention
```

7.1.2 Broker Failures

Detection:

```
# Check broker status
kafka-broker-api-versions.sh --bootstrap-server localhost:9092

# Check controller logs
grep "Broker.*shutdown" /var/log/kafka/controller.log
```

Recovery Steps:

1. **Identify Failure Cause:**

```
# Check broker logs
tail -100 /var/log/kafka/server.log

# Common causes:
- OutOfMemoryError
- Disk full
- Network partition
- Process killed
```

2. **Restart Broker:**

```
# Clean restart
kafka-server-start.sh config/server.properties

# Check if broker rejoined cluster
kafka-broker-api-versions.sh --bootstrap-server localhost:9092
```

3. **Verify Partition Recovery:**

```
# Check for under-replicated partitions
kafka-topics.sh --bootstrap-server localhost:9092 \
  --describe --under-replicated-partitions

# Wait for ISR to sync
```

```
watch -n 5 'kafka-topics.sh --bootstrap-server localhost:9092 \
  --describe --topic orders'
```

7.1.3 Consumer Lag Issues

Diagnosis:

```
# Check consumer group lag
kafka-consumer-groups.sh --bootstrap-server localhost:9092 \
  --describe --group order-processors

# Metrics to analyze:
- LAG: How far behind consumer is
- LOG-END-OFFSET: Latest message offset
- CURRENT-OFFSET: Consumer's current position
```

Solutions:

1. **Scale Consumers:**

```
# Add more consumer instances
# Ensure consumers ≤ partitions

# If at partition limit, increase partitions:
kafka-topics.sh --bootstrap-server localhost:9092 \
  --alter --topic orders --partitions 12
```

2. **Tune Consumer Configuration:**

```
# Increase fetch size
fetch.min.bytes=1048576  # 1 MB
max.partition.fetch.bytes=2097152  # 2 MB

# Reduce commit frequency
enable.auto.commit=true
auto.commit.interval.ms=5000

# Increase session timeout
session.timeout.ms=30000
max.poll.interval.ms=300000
```

```
max.poll.records=1000
```

3. **Optimize Processing:**

```
// Batch processing
List<ConsumerRecord> batch = new ArrayList<>();
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
    for (ConsumerRecord<String, String> record : records) {
        batch.add(record);
        if (batch.size() >= BATCH_SIZE) {
            processBatch(batch);
            consumer.commitSync();
            batch.clear();
        }
    }
}
```

7.1.4 Message Loss

Prevention:

1. **Producer Configuration:**

```
# Enable idempotence
enable.idempotence=true
# This sets:
# acks=all
# retries=Integer.MAX_VALUE
# max.in.flight.requests.per.connection=5

# Additional settings
compression.type=lz4
```

2. **Broker Configuration:**

```
# Require minimum in-sync replicas
min.insync.replicas=2
default.replication.factor=3
```

```
unclean.leader.election.enable=false
```

3. **Consumer Configuration:**

```
# Manual offset management
enable.auto.commit=false

# Commit after processing
isolation.level=read_committed  # For transactional producers
```

7.1.5 Rebalancing Issues

Symptoms:

- Frequent consumer group rebalances
- Processing pauses
- Consumer timeouts

Solutions:

1. **Extend Timeouts:**

```
# Consumer configuration
session.timeout.ms=30000  # Heartbeat timeout
heartbeat.interval.ms=3000  # Heartbeat frequency
max.poll.interval.ms=600000  # Processing time limit
```

2. **Reduce Rebalance Delay:**

```
# Broker configuration
group.initial.rebalance.delay.ms=3000  # Wait for members to join
```

3. **Use Cooperative Rebalancing:**

```
# Consumer configuration (Kafka 2.4+)
partition.assignment.strategy=org.apache.kafka.clients.consumer.CooperativeStickyAssignor
```

7.1.6 Performance Issues

Bottleneck Identification:

1. **Producer Bottleneck:**

```
# Symptoms:
- High producer request latency
- Low throughput

# Check metrics:
kafka.producer:type=producer-metrics,attribute=request-latency-avg

# Solutions:
- Increase batch.size
- Increase linger.ms
- Enable compression
- Add more partitions
```

2. **Broker Bottleneck:**

```
# Symptoms:
- High CPU/disk usage
- Slow request processing

# Check metrics:
kafka.network:type=RequestMetrics,name=RequestQueueTimeMs

# Solutions:
- Increase num.io.threads
- Upgrade hardware (SSD, more CPU)
- Scale cluster (add brokers)
- Optimize topic configurations
```

3. **Consumer Bottleneck:**

```
# Symptoms:
- High consumer lag
- Low records-consumed-rate

# Check metrics:
kafka.consumer:type=consumer-fetch-manager-metrics,attribute=records-lag-max
```

```
# Solutions:
- Scale consumers
- Optimize processing logic
- Increase fetch size
- Use parallel processing
```

7.2 Troubleshooting Workflows

7.2.1 Cluster Health Check

```
#!/bin/bash
# cluster-health-check.sh

echo "=== Broker Status ==="
kafka-broker-api-versions.sh --bootstrap-server localhost:9092

echo "=== Under-Replicated Partitions ==="
kafka-topics.sh --bootstrap-server localhost:9092 \
  --describe --under-replicated-partitions

echo "=== Offline Partitions ==="
kafka-topics.sh --bootstrap-server localhost:9092 \
  --describe --unavailable-partitions

echo "=== Consumer Group Lag ==="
kafka-consumer-groups.sh --bootstrap-server localhost:9092 \
  --list | while read group; do
    echo "Group: $group"
    kafka-consumer-groups.sh --bootstrap-server localhost:9092 \
      --describe --group $group
done

echo "=== Disk Usage ==="
df -h | grep kafka

echo "=== Recent Errors in Logs ==="
grep -i "error\|exception" /var/log/kafka/server.log | tail -20
```

7.2.2 Performance Troubleshooting

```
# 1. Check system resources
top
iostat -x 1 10
vmstat 1 10
netstat -s

# 2. Analyze Kafka metrics
# Use JConsole or export JMX metrics

# 3. Profile producers/consumers
# Enable metrics in client applications

# 4. Test end-to-end latency
kafka-run-class kafka.tools.EndToEndLatency \
  localhost:9092 test-topic 10000 100 1024

# 5. Benchmark throughput
kafka-producer-perf-test.sh --topic test-topic \
  --num-records 1000000 --record-size 1024 \
  --throughput -1 --producer-props bootstrap.servers=localhost:9092

kafka-consumer-perf-test.sh --bootstrap-server localhost:9092 \
  --topic test-topic --messages 1000000
```

7.3 Recovery Procedures

7.3.1 Disaster Recovery

Scenario: Complete Cluster Loss

1. **Restore from Backup:**

```
# If using MirrorMaker 2
# Failover to DR cluster
# Update client configurations to point to DR cluster

# If using snapshots
# Restore Kafka data directories
```

```
# Restart cluster
```

2. **Reset Consumer Offsets:**

```
# If offsets lost, reset to beginning or specific point
kafka-consumer-groups.sh --bootstrap-server localhost:9092 \
  --group order-processors \
  --reset-offsets --to-earliest --all-topics --execute
```

7.3.2 Data Corruption

Scenario: Corrupted Log Segments

```
# 1. Stop affected broker
kafka-server-stop.sh

# 2. Identify corrupted segments
kafka-run-class kafka.tools.DumpLogSegments \
  --files /var/kafka-logs/orders-0/00000000000000000000.log \
  --print-data-log

# 3. Delete corrupted segment (only if replicated)
# Kafka will recover from other replicas
rm /var/kafka-logs/orders-0/00000000000000000000.log
rm /var/kafka-logs/orders-0/00000000000000000000.index

# 4. Restart broker
kafka-server-start.sh config/server.properties

# 5. Verify recovery
kafka-topics.sh --bootstrap-server localhost:9092 \
  --describe --topic orders
```

Section 8: Advanced Topics and Ecosystem

8.1 Schema Registry

Schema Registry provides centralized schema management:

Supported Formats:

- Avro (most common)
- Protobuf
- JSON Schema

Key Features:

- Schema versioning
- Backward/forward/full compatibility checking
- Schema evolution
- Centralized schema storage

Producer Configuration:

```
# Avro producer
key.serializer=io.confluent.kafka.serializers.KafkaAvroSerializer
value.serializer=io.confluent.kafka.serializers.KafkaAvroSerializer
schema.registry.url=http://schema-registry:8081
```

Consumer Configuration:

```
# Avro consumer
key.deserializer=io.confluent.kafka.serializers.KafkaAvroDeserializer
value.deserializer=io.confluent.kafka.serializers.KafkaAvroDeserializer
schema.registry.url=http://schema-registry:8081
specific.avro.reader=true
```

8.2 ksqlDB

ksqlDB is a streaming SQL engine for Kafka:

Use Cases:

- Materialized views
- Stream processing with SQL

- Real-time analytics
- ETL pipelines

Example Queries:

```
-- Create stream from topic
CREATE STREAM orders_stream (
  order_id VARCHAR KEY,
  customer_id VARCHAR,
  product VARCHAR,
  amount DECIMAL(10,2)
) WITH (
  KAFKA_TOPIC='orders',
  VALUE_FORMAT='JSON'
);

-- Create aggregated table
CREATE TABLE orders_by_customer AS
  SELECT customer_id,
         COUNT(*) as order_count,
         SUM(amount) as total_amount
  FROM orders_stream
  GROUP BY customer_id
  EMIT CHANGES;

-- Query materialized view
SELECT * FROM orders_by_customer WHERE customer_id='12345';
```

8.3 Message Compression

Compression reduces network and storage requirements:

Compression Types:

Type	Compression Ratio	CPU Usage	Speed	Use Case
lz4	Low	Lowest	Fastest	Default choice, balanced
snappy	Medium	Moderate	Fast	Good balance
gzip	Highest	Highest	Slowest	Bandwidth-constrained

zstd	Medium-High	Moderate	Moderate	Modern alternative to gzip

Configuration:

```
# Producer
compression.type=lz4

# Broker (preserve producer compression)
compression.type=producer
```

8.4 Quotas and Throttling

Quotas prevent clients from overwhelming the cluster:

Types of Quotas:

1. **Network Bandwidth Quotas:** Bytes per second
2. **Request Rate Quotas:** Percentage of thread utilization

Configure Quotas:

```
# Set producer quota (10 MB/s)
kafka-configs.sh --bootstrap-server localhost:9092 \
  --alter --add-config 'producer_byte_rate=10485760' \
  --entity-type users --entity-name alice

# Set consumer quota (20 MB/s)
kafka-configs.sh --bootstrap-server localhost:9092 \
  --alter --add-config 'consumer_byte_rate=20971520' \
  --entity-type users --entity-name alice

# Set request quota (50% of broker threads)
kafka-configs.sh --bootstrap-server localhost:9092 \
  --alter --add-config 'request_percentage=50' \
  --entity-type users --entity-name alice

# Client-ID based quotas
kafka-configs.sh --bootstrap-server localhost:9092 \
  --alter --add-config 'producer_byte_rate=10485760' \
```

```
  --entity-type clients --entity-name my-producer
```

8.5 Exactly-Once Semantics (EOS)

Kafka provides exactly-once processing guarantees:

Idempotent Producer:

```
enable.idempotence=true
# Automatically sets:
# acks=all
# retries=Integer.MAX_VALUE
# max.in.flight.requests.per.connection=5
```

Transactional Producer:

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("transactional.id", "my-transactional-id");
props.put("enable.idempotence", "true");

KafkaProducer<String, String> producer = new KafkaProducer<>(props);

// Initialize transactions
producer.initTransactions();

try {
    producer.beginTransaction();
    producer.send(new ProducerRecord<>("topic1", "key", "value"));
    producer.send(new ProducerRecord<>("topic2", "key", "value"));
    producer.commitTransaction();
} catch (Exception e) {
    producer.abortTransaction();
}
```

Transactional Consumer:

```
isolation.level=read_committed
enable.auto.commit=false
```

Section 9: Hands-On Labs and Exercises

Lab 1: Kafka Cluster Setup

Objective: Set up a 3-broker Kafka cluster with proper configuration

Steps:

1. **Configure Broker 1:**

```
# server-1.properties
broker.id=1
listeners=PLAINTEXT://localhost:9092
log.dirs=/tmp/kafka-logs-1
num.partitions=3
default.replication.factor=3
min.insync.replicas=2
log.retention.hours=168
```

2. **Configure Brokers 2 and 3:**

```
# server-2.properties (change broker.id, listeners, log.dirs)
# server-3.properties (change broker.id, listeners, log.dirs)
```

3. **Start Brokers:**

```
kafka-server-start.sh config/server-1.properties &
kafka-server-start.sh config/server-2.properties &
kafka-server-start.sh config/server-3.properties &
```

4. **Verify Cluster:**

```
kafka-broker-api-versions.sh --bootstrap-server localhost:9092
```

Lab 2: Security Configuration

Objective: Enable SSL authentication and ACLs

Tasks:

1. Generate SSL certificates
2. Configure brokers for SSL
3. Create ACLs for users
4. Test producer/consumer with SSL and ACLs

Lab 3: Monitoring Setup

Objective: Set up monitoring with JMX and Prometheus

Tasks:

1. Enable JMX on Kafka brokers
2. Deploy JMX Exporter
3. Configure Prometheus to scrape metrics
4. Create Grafana dashboards
5. Set up alerts for critical metrics

Lab 4: Performance Tuning

Objective: Optimize cluster for high throughput

Tasks:

1. Run baseline performance tests
2. Tune producer/consumer configurations
3. Adjust broker parameters
4. Re-test and measure improvements
5. Document configuration changes

Lab 5: Disaster Recovery

Objective: Implement and test DR procedures

Tasks:

1. Set up MirrorMaker 2 between clusters
2. Configure bidirectional replication
3. Simulate primary cluster failure
4. Failover to secondary cluster
5. Verify data integrity and consumer offsets

Lab 6: Troubleshooting Scenarios

Scenario 1: Under-replicated partitions appear

- Diagnose the issue
- Implement fix
- Verify resolution

Scenario 2: Consumer lag increasing

- Analyze consumer group
- Identify bottleneck
- Scale consumers appropriately

Scenario 3: Broker crashes and won't restart

- Check logs
- Identify root cause
- Recover broker to healthy state

Section 10: Exam Preparation Strategy

10.1 Study Plan (8-Week Program)

Weeks 1-2: Fundamentals

- Kafka architecture deep dive

- Replication and durability concepts
- KRaft vs ZooKeeper
- Practice: Set up local cluster

Weeks 3-4: Security and Configuration

- SSL/TLS and SASL authentication
- ACL management
- Broker and topic configuration
- Practice: Secure cluster setup

Weeks 5-6: Operations and Monitoring

- Kafka Connect deployment
- JMX metrics and monitoring
- Performance tuning
- Practice: Monitoring stack setup

Weeks 7-8: Troubleshooting and Review

- Common issues and resolutions
- Disaster recovery procedures
- Practice exams
- Review weak areas

10.2 Key Study Resources

Official Documentation:

- Apache Kafka Documentation: https://kafka.apache.org/documentation/
- Confluent Documentation: https://docs.confluent.io/
- Confluent Training Courses

Practice Platforms:

- VMExam CCAAK Practice Tests

- Confluent Community Slack
- Kafka hands-on labs in cloud environments

Additional Resources:

- Kafka: The Definitive Guide (O'Reilly)
- Confluent blog posts and webinars
- GitHub repositories with practice questions

10.3 Exam Tips

Time Management:

- 90 minutes for 60 questions = 1.5 minutes per question
- Flag difficult questions and return later
- Don't spend more than 2 minutes on any single question

Question Strategies:

- Read questions carefully, noting keywords: "best," "most," "primary"
- Eliminate obviously wrong answers first
- Watch for multiple-select questions (may have 2+ correct answers)
- Consider practical scenarios over theoretical perfection

Common Question Topics:

- Controller election process
- ISR mechanics and min.insync.replicas
- Consumer group coordination
- Log compaction behavior
- Security configurations
- Monitoring metrics interpretation

During the Exam:

- Start with questions you're confident about

- Mark uncertain questions for review
- Verify answers before submitting
- Use process of elimination for challenging questions

10.4 Sample Practice Questions

Question 1: When a broker running the controller fails, how is the new controller selected?

- A) The broker with highest broker.id
- B) The next broker to recreate the ZooKeeper ephemeral node / win Raft election
- C) The broker with most free disk space
- D) Manually designated by administrator

Answer: B

Question 2: What does the High Water Mark (HWM) represent?

- A) Maximum partition size
- B) Producer offset position
- C) Offset of last message replicated to all ISR members
- D) Consumer group commit offset

Answer: C

Question 3: Which configuration ensures a producer waits for acknowledgment from all in-sync replicas?

- A) acks=0
- B) acks=1
- C) acks=all
- D) acks=quorum

Answer: C

Question 4: What is the recommended minimum replication factor for production topics?

- A) 1
- B) 2

- C) 3
- D) 5

Answer: C

Question 5: Which metric indicates partitions without an active leader?

- A) UnderReplicatedPartitions
- B) OfflinePartitionsCount
- C) LeaderElectionRate
- D) PartitionLag

Answer: B

Section 11: Quick Reference Guide

11.1 Essential Commands Cheat Sheet

Cluster Management:

```
# Start Kafka
kafka-server-start.sh config/server.properties

# Stop Kafka gracefully
kafka-server-stop.sh

# Check broker versions
kafka-broker-api-versions.sh --bootstrap-server localhost:9092
```

Topic Management:

```
# Create topic
kafka-topics.sh --bootstrap-server localhost:9092 \
  --create --topic mytopic --partitions 3 --replication-factor 3

# List topics
kafka-topics.sh --bootstrap-server localhost:9092 --list
```

```
# Describe topic
kafka-topics.sh --bootstrap-server localhost:9092 --describe --topic mytopic

# Delete topic
kafka-topics.sh --bootstrap-server localhost:9092 --delete --topic mytopic

# Alter partitions
kafka-topics.sh --bootstrap-server localhost:9092 --alter --topic mytopic --partitions 6
```

Consumer Groups:

```
# List groups
kafka-consumer-groups.sh --bootstrap-server localhost:9092 --list

# Describe group
kafka-consumer-groups.sh --bootstrap-server localhost:9092 --describe --group mygroup

# Reset offsets
kafka-consumer-groups.sh --bootstrap-server localhost:9092 \
  --group mygroup --reset-offsets --to-earliest --all-topics --execute
```

Configuration Management:

```
# Alter topic config
kafka-configs.sh --bootstrap-server localhost:9092 \
  --alter --entity-type topics --entity-name mytopic \
  --add-config retention.ms=86400000

# Alter broker config
kafka-configs.sh --bootstrap-server localhost:9092 \
  --alter --entity-type brokers --entity-name 1 \
  --add-config max.connections=2000

# Describe configs
kafka-configs.sh --bootstrap-server localhost:9092 \
  --describe --entity-type topics --entity-name mytopic
```

ACL Management:

```
# Add ACL
kafka-acls.sh --bootstrap-server localhost:9092 \
  --add --allow-principal User:alice \
  --operation Read --topic mytopic

# List ACLs
kafka-acls.sh --bootstrap-server localhost:9092 --list

# Remove ACL
kafka-acls.sh --bootstrap-server localhost:9092 \
  --remove --allow-principal User:alice \
  --operation Read --topic mytopic
```

11.2 Configuration Quick Reference

Critical Broker Configs:

```
# Identity
broker.id=1
node.id=1

# Networking
listeners=PLAINTEXT://:9092
advertised.listeners=PLAINTEXT://broker1:9092

# Replication
default.replication.factor=3
min.insync.replicas=2
replica.lag.time.max.ms=10000

# Retention
log.retention.hours=168
log.segment.bytes=1073741824

# Performance
num.network.threads=8
num.io.threads=16
```

Critical Producer Configs:

```
bootstrap.servers=localhost:9092
acks=all
retries=Integer.MAX_VALUE
enable.idempotence=true
compression.type=lz4
batch.size=16384
linger.ms=10
```

Critical Consumer Configs:

```
bootstrap.servers=localhost:9092
group.id=mygroup
enable.auto.commit=false
session.timeout.ms=30000
max.poll.interval.ms=300000
max.poll.records=500
```

11.3 Metric Alert Thresholds

Metric	Warning	Critical	Action
OfflinePartitionsCount	> 0	> 5	Immediate investigation
UnderReplicatedPartitions	> 10	> 50	Check broker health
ActiveControllerCount	!= 1	!= 1	Controller election issue
Consumer Lag	> 10000	> 100000	Scale consumers
Disk Usage	> 75%	> 90%	Free space or add storage
Request Latency (p99)	> 200ms	> 500ms	Performance tuning needed

11.4 Troubleshooting Decision Tree

```
Problem: High Consumer Lag
├── Check: Number of consumers vs partitions
│   ├── If consumers < partitions → Add consumers
│   └── If consumers >= partitions → Check processing time
│       ├── Slow processing → Optimize application code
│       └── Fast processing → Increase fetch size
│
```

```
Problem: Under-Replicated Partitions
├── Check: Which broker hosting affected replicas
│   ├── Broker online → Check disk/network
│   │   ├── Disk full → Free space
│   │   └── Network issues → Fix connectivity
│   └── Broker offline → Restart broker
│
Problem: Message Loss
├── Check: Producer acks configuration
│   ├── If acks=0 or 1 → Change to acks=all
│   └── If acks=all → Check min.insync.replicas
│      └── Ensure min.insync.replicas >= 2
│
Problem: Poor Throughput
├── Check: Bottleneck location
│   ├── Producer → Increase batch size, compression
│   ├── Broker → Add brokers, optimize config
│   └── Consumer → Scale consumers, increase fetch size
```

Conclusion

This comprehensive study guide covers all seven domains of the CCAAK certification exam. Success requires:

1. **Theoretical Knowledge:** Understanding Kafka architecture, concepts, and best practices
2. **Hands-On Experience:** Practical work with Kafka clusters in real scenarios
3. **Troubleshooting Skills:** Ability to diagnose and resolve common issues
4. **Exam Preparation:** Practice questions and time management strategies

Key Success Factors:

- Minimum 6-12 months hands-on Kafka experience[5][2]
- Deep understanding of all seven exam domains
- Regular practice with command-line tools
- Familiarity with monitoring and troubleshooting workflows

- Multiple practice exam attempts scoring >85%

Final Recommendations:

1. Set up a local multi-broker Kafka cluster for practice
2. Work through all hands-on labs in this guide
3. Join the Confluent Community Slack for support
4. Take multiple practice exams before scheduling
5. Review official Confluent documentation thoroughly
6. Focus extra attention on Cluster Configuration (22%) and Troubleshooting (15%) sections

Good luck with your CCAAK certification journey!