Build Your Own Kafka: Design Document

Overview

This document describes the design and implementation of a distributed message queue supporting partitioned topics, consumer groups, and replication. The key architectural challenge is maintaining ordered, durable message delivery across a scalable and fault-tolerant cluster of brokers, while navigating the inherent trade-offs between latency, throughput, and consistency.

> This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

> **Milestone(s):** All (foundational context)

1. Context and Problem Statement

At the heart of modern software architecture lies a fundamental tension: how do we reliably and efficiently move streams of events—user actions, sensor readings, database updates, payment transactions—between dozens or hundreds of independent services? This is the **scalable event streaming problem**. Traditional request/response communication creates a tightly coupled web of dependencies, where the failure or slowness of one service can cascade through the entire system. Message queues emerged to decouple services by introducing an asynchronous buffer, but early designs often traded off one critical property for another: throughput for ordering, durability for latency, or simplicity for scale.

Apache Kafka, introduced by LinkedIn in 2011, revolutionized this space by adopting a surprisingly simple core abstraction: the **immutable, partitioned, append-only log**. Instead of treating messages as transient envelopes to be delivered and deleted, Kafka treats them as permanent records in a ledger. This design shifts the focus from complex routing and delivery semantics to durable storage and high-speed sequential I/O. This document outlines the design and implementation of a simplified, educational version of such a system—"Build Your Own Kafka." We will explore how to construct a distributed message queue that provides strong ordering guarantees, horizontal scalability, and fault tolerance through replication, while navigating the inherent trade-offs between latency, throughput, and consistency.

1.1 The Centralized Ledger Analogy

To build intuition, imagine a traditional bank's **centralized ledger**. This ledger records every transaction—deposits, withdrawals, transfers—in strict chronological order. Each account's balance is not a stored number but a derived state computed by replaying all transactions for that account from the beginning of time. This model provides an **authoritative, immutable history** of all activity.

Now, scale this bank to serve millions of customers. A single ledger book becomes a bottleneck; clerks cannot write fast enough, and customers wait in long queues to record their transactions. The bank's solution is **partitioning**: they create multiple identical ledger books, each responsible for a specific subset of accounts (e.g., accounts A-L in Ledger 1, M-Z in Ledger 2). This is the core mental model for a Kafka **topic**:

- **Topic**: The overall stream of related events (e.g., `bank-transactions`). Analogous to the *concept* of the transaction log.
- **Partition**: A single, ordered, immutable sequence of records within a topic. Analogous to one physical ledger book. Ordering is guaranteed *only within a single partition*.
- **Record**: A single entry in the ledger (e.g., "Account #123: Deposit $100 at 14:30"). It consists of a key, a value, and a timestamp.
- **Offset**: The sequential, monotonically increasing index number assigned to each record as it is appended to its partition (e.g., record #0, #1, #2...). This is the page and line number in the ledger book.

The critical insight of the key is its role in **deterministic partitioning**. When a transaction (record) arrives, its account number (key) is hashed to consistently route it to the same ledger (partition). All transactions for account #123 always go to the same partition, preserving

their chronological order. If the key is null, the record is assigned to partitions in a round-robin fashion for load distribution.

> **Design Insight:** The shift from a "message queue" to a "log" is profound. A queue is about delivery and removal; a log is about durable recording and subscription. Consumers read from the log at their own pace, and the log persists independently of them. This enables replayability, auditing, and the ability to spawn new consumers for historical data analysis—patterns that are cumbersome or impossible with traditional queues.

This ledger analogy extends to the cluster:

- **Broker**: A bank branch office that safely stores a set of ledger books (partitions). A cluster consists of multiple brokers for redundancy and capacity.
- **Producer**: A teller or ATM that accepts new transactions and writes them into the correct ledger book.
- **Consumer**: An auditor or accountant who reads sequentially from one or more ledger books to compute balances or detect fraud.
- **Consumer Group**: A team of auditors, where each member is assigned a non-overlapping subset of the ledger books to parallelize the audit work.

1.2 The Scalable Event Streaming Problem

Building a system that realizes this ledger analogy at internet scale requires satisfying a interconnected set of core requirements. These requirements often exist in tension, and the design choices we make will prioritize certain qualities over others.

Core Requirements:

Requirement	Description	Implication for Design
High-Throughput & Low Latency	The system must ingest and deliver hundreds of thousands—or millions—of events per second with minimal delay.	Demands efficient serialization, batching, and heavy reliance on sequential disk I/O over random access. Network and disk overhead must be minimized.
Ordered Delivery	Events with a causal relationship (e.g., updates to the same entity) must be processed in the order they occurred.	Requires a strong ordering guarantee *within a partition*. The system must ensure all records for a given key go to the same partition and are appended in arrival order.
Durability & Fault Tolerance	Once acknowledged, messages must not be lost, even in the face of individual server failures, power outages, or network partitions.	Necessitates replication of data across multiple brokers (servers) and durable writes to disk (fsync). The system must define what "acknowledged" means (e.g., written to leader? written to all replicas?).
Scalability & Elasticity	The system's capacity should increase linearly by adding more brokers. It should also allow consumers and producers to scale independently.	Requires stateless producers/consumers, data partitioning, and dynamic reassignment of partition ownership. No single component (like a central router) should become a bottleneck.
Decoupling of Producers & Consumers	Producers should not need to know about consumers, and vice versa. Consumers should be able to process data at their own pace.	Implies a publish-subscribe model with persistent storage. The log acts as a buffer, allowing producers to write as fast as they can and consumers to read when they are ready.
Consumer Group Semantics	Multiple consumers can work together to process a topic, with each partition being consumed by exactly one group member at a time. This provides parallel processing and fault tolerance for the consumption side.	Requires a coordination service to manage group membership, assign partitions, and track progress (offsets).

The primary challenge is that these requirements conflict. For example:

- **Durability vs. Latency:** Ensuring a message is written to disk on multiple replicas (high durability) increases latency compared to acknowledging once the leader receives it.

- **Ordering vs. Throughput:** Maintaining strict global order requires a single sequencing point (a bottleneck). Partitioning sacrifices global order for parallelism and throughput.
- **Scalability vs. Simplicity:** A simple, single-broker design is easy to reason about but doesn't scale. A distributed design introduces complexity around consistency, failure detection, and recovery.

The Log-Based Solution: The partitioned log model directly addresses these tensions:

1. **Throughput & Scalability:** Partitioning allows parallel ingestion and consumption across many brokers and disks. Sequential appends maximize disk I/O efficiency.
2. **Ordered Delivery:** Order is guaranteed per-partition, which is sufficient for most use cases when keys are chosen wisely (e.g., `user_id`).
3. **Durability:** The log is an immutable, append-only file. Replication provides durability against node failure, while periodic fsync provides durability against process crashes.
4. **Decoupling:** The log is the persistent interface. Producers append; consumers subscribe to positions (offsets) in the log.

> **Architecture Decision Record (ADR): Log as the Primary Abstraction**
>
> - **Context:** We need a data structure that serves as the durable, ordered backbone for event streaming, balancing throughput, ordering, and simplicity.
> - **Options Considered:**
>   1. **Traditional Message Queue (e.g., AMQP model):** Messages are stored in in-memory or disk-backed queues, deleted upon acknowledgment. Supports complex routing (exchanges, bindings).
>   2. **Distributed Commit Log:** Messages are immutable records appended to partitioned, replicated log segments. Simple append/read semantics.
>   3. **Event Sourcing Database:** Treats the log as a system of record, with built-in query capabilities and state derivation.
> - **Decision:** Use a **Distributed Commit Log** as the primary abstraction.
> - **Rationale:** The log model provides the optimal balance for our core requirements. Its simplicity (append/read) enables extremely high throughput. Immutability simplifies replication and failure recovery. Partitioning provides horizontal scalability. The retention-based cleanup model is simpler than per-message deletion and enables valuable replay capabilities.
> - **Consequences:**
>   - **Enables:** High throughput, simple replication, replayability, and natural support for multiple consumers.
>   - **Trade-offs:** Requires consumers to manage their own position (offset). Storage management shifts from message TTL to log segment retention/compaction. No built-in complex routing; filtering must happen on the consumer side.

Option	Pros	Cons	Why Chosen?
Traditional Message Queue	Rich routing semantics, immediate deletion saves space, mature protocols (AMQP).	Deletion complicates replication and replay, often relies on slower random I/O, complex to scale horizontally.	Better for complex routing needs, but our primary goals are throughput and scalability.
Distributed Commit Log	Maximizes sequential I/O, simple replication, natural replay, excellent horizontal scalability.	Consumers must manage offset, no built-in message routing, storage grows until retention/compaction.	Directly aligns with high-throughput, ordered, durable event streaming. Simplicity is a feature.
Event Sourcing Database	Powerful query abilities, strong consistency models, integrated state derivation.	Higher overhead per message, more complex operational model, often lower write throughput.	Overkill for a messaging backbone; we want a transport layer, not a system of record.

1.3 Existing Approaches & Trade-offs

Kafka's log-centric design did not emerge in a vacuum. It sits within a landscape of messaging and streaming systems, each making different architectural trade-offs. Understanding these alternatives clarifies *why* Kafka's choices are particularly well-suited for high-

volume event streaming.

1. Traditional Enterprise Message Brokers (e.g., RabbitMQ, ActiveMQ): These systems are built around the **smart broker/dumb consumer** model. The broker is responsible for complex message routing, delivery guarantees, and transaction management.

- **Typical Architecture:** Centralized or clustered brokers hold messages in queues. They support advanced patterns like publish/subscribe, request/reply, and point-to-point via exchanges, bindings, and queues.
- **Trade-offs:**
  - **Pros:** Rich feature set (priority, TTL, dead-letter queues), strong delivery guarantees with transactions, flexible routing.
  - **Cons:** The broker is a complex, stateful monolith that can become a performance bottleneck. Scaling often involves partitioning (federation/shovel) which is complex. Persistence models (often B-trees) can be slower than sequential logs. Consumers are generally stateful connections to the broker.
- **Comparison to Our Design:** Our system adopts a **dumb broker/smart consumer** model. The broker's job is simple: durably store logs and serve reads/writes. All intelligence (partition assignment, offset management, consumer state) is pushed to the client libraries. This shifts complexity but allows the broker to be simpler, more robust, and easier to scale horizontally.

2. Distributed Log Systems (e.g., Apache BookKeeper, NATS Streaming): These share Kafka's log-oriented philosophy but differ in implementation details.

- **Apache BookKeeper:** Provides a reliable storage service for log segments. Kafka historically used BookKeeper for tiered storage. It offers strong durability guarantees via a quorum-write protocol.
- **NATS Streaming:** Provides a similar log abstraction but often with simpler clustering models and a focus on the Go ecosystem.
- **Trade-offs:** BookKeeper separates storage and serving, which can offer flexibility. NATS Streaming offers simplicity and integration with the NATS core. Kafka's integrated design (managing its own storage) provides end-to-end control over performance and semantics.

3. Streaming Platforms & Databases (e.g., Apache Pulsar, AWS Kinesis, Apache Flink): These systems extend the core log model with additional capabilities.

- **Apache Pulsar:** Uses a similar log abstraction but with a clearer separation of compute (brokers) and storage (Apache BookKeeper). Supports both queuing and streaming semantics natively.
- **AWS Kinesis:** A managed service offering a Kafka-like shard (partition) model, with tight integration into the AWS ecosystem. Often has stricter limits on retention and throughput per shard.
- **Apache Flink:** A true stream processing engine that can also act as a persistent event store with its Stateful Functions API, blurring the line between transport and processing.
- **Trade-offs:** Pulsar's separation can aid in independent scaling and operational flexibility. Kinesis offers simplicity as a service. Flink integrates processing and storage. Our educational design follows Kafka's more monolithic broker model for simplicity of understanding, acknowledging that separation of concerns is a valid advanced architectural pattern.

4. General-Purpose Distributed Databases (e.g., etcd, CockroachDB): These can be (ab)used as message queues by treating a table as a queue.

- **Trade-offs:**
  - **Pros:** Strong consistency, transactions, and rich querying.
  - **Cons:** Write amplification, lower throughput for streaming workloads, and inefficient tail reads. They are optimized for random access and point queries, not sequential appends and reads.
- **Key Distinction:** Databases are optimized for **point-in-time queries** over a dataset. Logs are optimized for **continuous ingestion and sequential reads** of an unbounded stream. The access patterns are fundamentally different.

The following table summarizes the key architectural trade-offs:

System Category	Primary Data Model	Scaling Model	Durability Mechanism	Typical Use Case
Traditional Broker (RabbitMQ)	Queue/Exchange	Vertical scaling or complex federation	Persistent messages in broker (often B-tree)	Enterprise application integration, complex routing.
Our Design / Kafka	Partitioned, Append-Only Log	Horizontal partitioning across brokers	Replicated log segments, sequential disk I/O	High-throughput event streaming, log aggregation, activity tracking.
Pulsar	Segmented Log (with separate storage)	Independent scaling of brokers & storage	BookKeeper ledger replication	Streaming and queuing with cloud-native operations.
Database-as-Queue	Table/Key-Value	Horizontal sharding (for distributed DBs)	Replicated write-ahead log (WAL)	When strong consistency and queueing are both required (generally an anti-pattern for high-volume streams).

> **The Central Trade-Off: Simplicity vs. Control.** Our educational design, following Kafka's early architecture, chooses to keep storage management within the broker. This creates a more monolithic but simpler-to-understand component. In production systems at scale, the trend is toward disaggregating storage (as seen in Pulsar/Kafka Tiered Storage) to improve elasticity and cost efficiency. However, for learning the fundamentals of replication, partitioning, and consensus, the integrated model provides a more coherent mental model.

By grounding our design in the immutable log analogy and understanding the landscape of alternatives, we establish a clear foundation. The subsequent sections will delve into how to concretely build a system that embodies these principles, starting with the fundamental unit of storage: the topic and its partitions.

Implementation Guidance

This section is purely conceptual and foundational; there is no code to implement. However, it is critical to internalize the mental models and trade-offs discussed here before writing any code. A strong conceptual understanding will prevent fundamental architectural missteps.

Next Steps for the Learner:

1. **Internalize the Ledger Analogy:** Sketch out how a topic named `page-views` with 3 partitions would store records for `user_id` keys 1, 2, and 3. Determine which partition each record would land on using a simple hash modulo partition count strategy.
2. **Contrast with a Queue:** Write down two key differences in behavior between a traditional queue (message deleted after acknowledgment) and a log (message retained until retention expires) from the perspective of a consumer that crashes and restarts.
3. **Anticipate Trade-offs:** Before starting Milestone 1, consider: If you prioritize very low latency for producers, what durability setting (`acks`) might you choose, and what is the potential risk?

Proceed to **Section 5: Component Design: Broker and Topic Partitions** to begin the implementation of the ledger (log) itself.

2. Goals and Non-Goals

> **Milestone(s):** All (foundational requirements)

This section establishes the bounded scope of our educational system. In any engineering endeavor, particularly when learning complex distributed systems, explicitly defining what we **will** and **will not** build is crucial. It prevents scope creep, focuses effort on core learning objectives, and sets realistic expectations about the system's capabilities and limitations.

The design philosophy for "Build Your Own Kafka" prioritizes **educational clarity over feature completeness** and **conceptual implementation over production robustness**. We aim to build a *functional prototype* that illustrates the architectural patterns and trade-offs of a distributed log, not a production-ready message broker.

2.1 Functional Goals

Our system must implement the core abstractions and protocols that make a Kafka-like message queue work. Think of these as the **minimum viable pillars** that, when combined, deliver the fundamental value proposition: scalable, ordered, and durable event streaming.

Functional Area	Required Capability	Description & Learning Objective
Topic & Partition Management	Create topics with configurable partitions	Learn how logical data streams (topics) are physically sharded across partitions for horizontal scaling. Each partition is an independent, ordered sequence.
	Consistent key-based routing	Understand how message keys determine partition assignment, enabling per-key ordering guarantees while distributing load.
	Sequential offset assignment	Implement the immutable, append-only log abstraction where each message gets a unique, monotonic offset within its partition.
Producer	Configurable acknowledgment levels (0, 1, all)	Experience the durability vs. latency trade-off firsthand by implementing fire-and-forget, leader-acknowledged, and fully-replicated write semantics.
	Client-side batching	Learn how accumulating messages into batches drastically improves network and disk I/O efficiency, a key technique for high-throughput systems.
	Retry with backoff	Handle transient failures (network timeouts, leader elections) gracefully without requiring application intervention.
Consumer Groups	Dynamic group membership	Implement the "team reading a book" metaphor where consumers in a group coordinate to share the work of consuming a topic's partitions.
	Partition assignment strategies (Range, RoundRobin)	Explore different algorithms for distributing partitions among group members, balancing fairness and locality.
	Offset commit & fetch	Provide at-least-once delivery semantics by allowing consumers to periodically save their read position, enabling resume-after-restart.
	Group rebalancing	Handle the complex distributed coordination problem of redistributing partitions when group membership changes (joins, leaves, failures).
Replication	Leader-follower log replication	Build fault tolerance by having followers asynchronously copy data from the partition leader, ensuring data survives single-broker failures.
	In-Sync Replica (ISR) management	Maintain a dynamic set of "caught-up" followers and understand how this set defines durability guarantees (`acks=all`).
	High Watermark advancement	Implement the safety mechanism that prevents consumers from reading data that could be lost on leader failure.
	Leader election (from ISR)	Recover from leader failure by promoting a fully caught-up follower, ensuring no data loss during failover (when configured).

These goals map directly to the four project milestones. Successfully implementing them yields a system where:

1. A **producer** can send a message with a key to a topic.
2. The system **hashes the key** to select a specific partition for that topic.
3. The **partition leader** appends the message to its local log, assigns an offset, and (based on `acks`) waits for replication to followers.
4. A **consumer group** with multiple members subscribes to the topic.

5. The **group coordinator** assigns each partition to exactly one group member.
6. Each **consumer** fetches messages from its assigned partitions, processes them, and commits its offsets.
7. If a **broker fails**, the system elects a new leader from the in-sync replicas for its hosted partitions, and consumer groups rebalance to reassign the affected partitions.

2.2 Non-Functional Goals

Beyond the feature checklist, our system should exhibit certain qualitative properties. These are not explicit user-facing features but essential characteristics of a well-designed educational prototype.

Property	Goal & Rationale
Durability (within reason)	Messages acknowledged with `acks=all` should survive a single broker failure without loss. This is achieved via on-disk storage (fsync configurable) and replication. We trade off absolute durability (e.g., synchronous disk writes for every message) for implementation simplicity and performance that illustrates the concept.
Scalability (demonstrable)	The architecture should allow horizontal scaling: adding more partitions to a topic should increase write throughput; adding more consumers to a group should increase read throughput. The implementation should not have central bottlenecks (like a global lock) that would prevent this in a multi-broker setup, even if our test cluster is small.
Fault Tolerance (basic)	The system must handle predictable failures gracefully: broker crashes trigger leader election and consumer rebalancing; network timeouts trigger producer retries. We aim for *clean failure modes* (clear error messages, no silent data corruption) over complex recovery from arbitrary faults (e.g., Byzantine failures).
Understandability (paramount)	Code and design must prioritize clarity for learners. This means: • Choosing simpler, more explicit algorithms over optimized but obscure ones. • Using clear naming and modular separation of concerns. • Providing extensive logging to visualize internal state (e.g., "Leader for partition `orders-0` changed from broker 1 to broker 2"). • Avoiding premature optimization that obfuscates the core logic.
Operability (for learning)	The system should be easy to run and observe locally. This includes simple startup/shutdown procedures, exposed metrics/logs for debugging, and the ability to simulate failures (e.g., killing a broker process) to see recovery in action.

> **Design Insight:** The primary non-functional goal is **pedagogical value**. Every design decision should be evaluated against the question: "Does this make the core concepts easier or harder to understand?" This often means choosing the more explicit, verbose, or modular approach over the most performant or idiomatic one.

2.3 Explicit Non-Goals

To maintain focus, we deliberately exclude several features that are important in production Kafka but are considered advanced or tangential to our core learning objectives. Stating these upfront prevents "feature creep" and clarifies the system's boundaries.

Area	What We Are NOT Building	Rationale & Learning Impact
Protocol Compatibility	Full Apache Kafka wire protocol (binary) compatibility.	Implementing the official protocol is complex and adds significant serialization/deserialization overhead. We will define a **simplified, explicit protocol** (e.g., JSON over HTTP or a trivial binary format) to keep the focus on system logic, not wire-format intricacies.
Transactional Messaging	Exactly-once semantics (EOS), idempotent producers, or cross-partition transactions.	Transactions introduce significant complexity (transaction coordinator, write-ahead logs for transaction state, two-phase commit). They are a valuable *advanced topic* but would double the project scope. We focus on at-least-once delivery as the foundational model.
Storage Efficiency	Tiered storage (offloading to S3), log compaction, or sophisticated compression (Zstandard, LZ4).	While critical for production cost and performance, these are optimizations built *on top of* the core log abstraction. Implementing them would distract from understanding the log itself. Our log segments will be simple on-disk files.
Production-Grade Security	Authentication (SASL), Authorization (ACLs), or Encryption (TLS).	Security is vital but largely orthogonal to the distributed systems algorithms we're studying. Adding it would complicate every network call and configuration without deepening understanding of replication, consensus, or partitioning.
Advanced Cluster Management	Dynamic configuration, partition reassignment tools, rack-aware replica placement, or multi-region replication.	These are operations and reliability features for managing large clusters. Our educational cluster will be static-configured and likely run on a single machine.
Monitoring & Metrics	A comprehensive metrics subsystem (JMX, Prometheus endpoints) or detailed health checks.	While we value observability for learning, we'll achieve it through structured logging rather than a full metrics pipeline to keep the codebase lean.
High-Performance Networking	Custom RPC layer with zero-copy, kernel-bypass, or sophisticated connection pooling.	We'll use the standard library's networking (`net/http` or `net`) to keep I/O models simple and understandable. Performance tuning is a separate, deep discipline.
Client Libraries	Idiomatic, feature-complete client libraries for multiple languages.	We'll build a minimal client (producer/consumer) in Go for testing and demonstration. The focus is on the server-side broker logic.

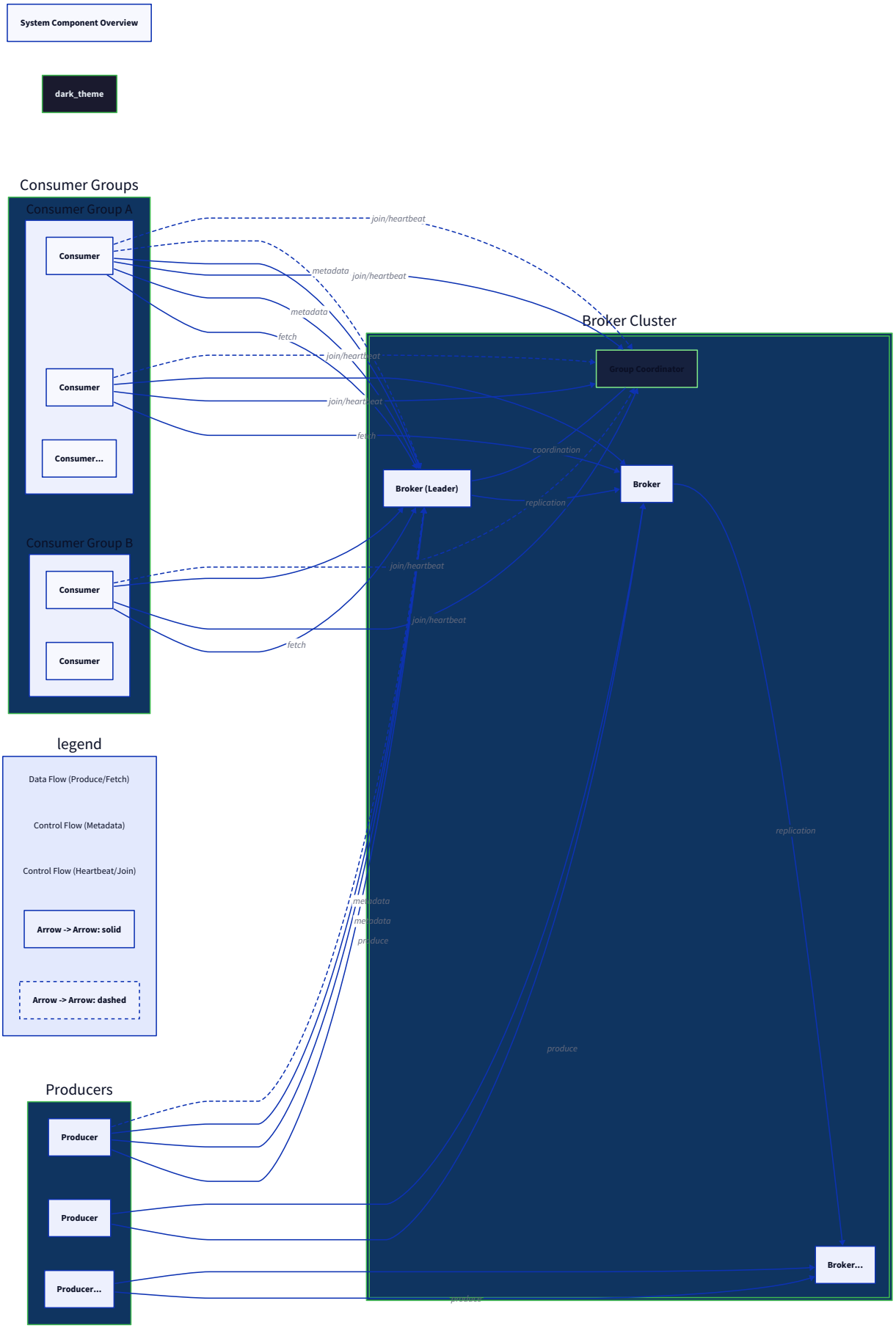
> **ADR Summary: Scope Limitation for Educational Focus Context:** The project could easily expand to encompass many production features of Apache Kafka. **Options Considered:**
>
> 1. **Minimal Core:** Build only the four milestone features with simplified protocols and storage.
> 2. **Extended Feature Set:** Include 1-2 advanced features like idempotent producers or log compaction.
> 3. **Protocol-First:** Focus on implementing the full Kafka binary protocol with a minimal feature set. **Decision:** Adopt Option 1 (Minimal Core). **Rationale:** The primary objective is to learn distributed systems concepts—replication, partitioning, consensus, and fault tolerance—not to build a production system. Every additional feature dilutes focus, increases complexity, and extends the timeline, risking that learners never complete the core educational journey. A working minimal system that demonstrates the key architectural patterns is more valuable than a half-finished system with many partially implemented features.
>
>   **Consequences:** The built system will be unsuitable for production use. Learners will need to understand its limitations. However, they will gain a crystal-clear understanding of the foundational mechanics, which is the best preparation for subsequently studying advanced features or contributing to real systems.

Understanding these non-goals is as important as the goals themselves. It defines the "finished line" for the project and allows learners to celebrate a complete, coherent system that, while limited, fundamentally works and teaches the intended lessons.

> **Milestone(s):** All (foundational architecture)

3. High-Level Architecture

This section presents the architectural blueprint of our distributed message queue system. Before diving into individual components, we establish a **mental model** of the entire system as a **global library system**. Imagine a vast library (the cluster) with multiple branches (brokers). Authors (producers) submit new books (messages) to the library catalog (topics), where each book is placed on a specific shelf (partition). Reading clubs (consumer groups) visit the library, with each club member (consumer) assigned specific shelves to read from. A head librarian (group coordinator) tracks which club members are present and which shelves they're reading. This analogy helps ground the abstract distributed components in tangible, real-world concepts before we formalize their technical specifications.


System Component Overview
dark_theme
Consumer Groups
Consumer Group A
Consumer
Consumer
Consumer...
Consumer Group B
Consumer
Consumer
Broker Cluster
Group Coordinator
Broker (Leader)
Broker
Broker...
join/heartbeat
metadata
join/heartbeat
metadata
fetch
join/heartbeat
join/heartbeat
fetch
coordination
replication
join/heartbeat
join/heartbeat
fetch
replication
metadata
metadata
produce
produce
produce
legend
Data Flow (Produce/Fetch)
Control Flow (Metadata)
Control Flow (Heartbeat/Join)
Arrow -> Arrow: solid
Arrow -> Arrow: dashed
Producers
Producer
Producer
Producer...

3.1 Component Overview & Responsibilities

The system comprises four primary component types that work in concert to deliver durable, ordered message streaming. Each component has a distinct responsibility domain and maintains specific state.

3.1.1 Core Components

Component	Primary Responsibility	Key Data Held	Key Operations Performed
Broker	Stores partition replicas, handles client read/write requests, and replicates data to other brokers.	- Local partition logs (`Partition.Log`) - Topic metadata - Replica state (leader/follower) - Consumer offsets (cached)	- Append records to partition logs - Serve fetch requests to consumers - Replicate data to follower brokers - Elect partition leaders - Track In-Sync Replicas (ISR)
Producer	Publishes records to topics, handling batching, reliability guarantees, and error recovery.	- Unsent message batches per partition - Metadata about cluster (topic/partition leaders) - Sequence numbers (for idempotency)	- Route messages to correct partitions - Accumulate messages into batches - Send batches to broker leaders - Handle acknowledgments and retries
Consumer	Reads records from topics, maintains consumption position, and participates in consumer groups.	- Assigned partitions and their current offsets - Fetched but unprocessed records - Group membership metadata	- Subscribe to topics - Fetch records from broker leaders - Process records and advance offset - Commit offsets to broker - Participate in group rebalancing
Group Coordinator	Manages consumer group membership, partition assignment, and offset persistence. Typically co-located with a broker.	- Group membership lists - Partition assignments per group - Committed offsets (durable storage) - Generation IDs (for rebalancing)	- Process join/leave requests - Trigger and manage rebalancing - Distribute partition assignments - Store and serve committed offsets

3.1.2 Component Interactions

The components interact through well-defined request/response patterns over network connections:

1. **Producer → Broker (Data Plane)**: Producers send `Produce` requests to broker leaders for specific partitions. These requests contain batched messages that the broker appends to its local log.
2. **Consumer → Broker (Data Plane)**: Consumers send `Fetch` requests to broker leaders to retrieve records from partitions, starting from a specific offset.

3. **Consumer ↔ Coordinator (Control Plane)**: Consumers communicate with the group coordinator for membership management:
    - `JoinGroup`: Register with a consumer group
    - `SyncGroup`: Receive partition assignments
    - `Heartbeat`: Maintain active membership
    - `OffsetCommit`: Persist consumption progress
    - `OffsetFetch`: Retrieve last committed offset
4. **Broker ↔ Broker (Replication Plane)**: Brokers replicate data among themselves:
    - Followers fetch new records from partition leaders
    - Leaders track follower replication progress (ISR)
    - Brokers elect new leaders when current leaders fail

> **Key Insight**: The separation between data plane (message flow) and control plane (coordination) is fundamental. The data plane must be optimized for high throughput and low latency, while the control plane prioritizes consistency and correctness, even at the cost of some latency.

3.1.3 Deployment Topology

In a typical deployment:

- Multiple **brokers** form a cluster, with each broker running on a separate physical or virtual machine.
- **Producers** and **consumers** run as client libraries within application processes.
- The **group coordinator** is typically implemented as a special role assumed by one of the brokers (often the first broker or one elected via consensus).
- All components communicate over TCP/IP networks, with the system designed to tolerate network partitions and broker failures.

3.2 Recommended File/Module Structure

For the Go implementation, we recommend a modular package structure that separates concerns, facilitates testing, and mirrors the architectural components. This structure balances simplicity for learners with good software engineering practices.

```
build-your-own-kafka/
├── cmd/                          # Application entry points
│   ├── broker/                   # Broker server executable
│   │   └── main.go               # Broker startup and configuration
│   ├── producer-cli/             # CLI tool for testing producers
│   │   └── main.go
│   └── consumer-cli/             # CLI tool for testing consumers
│       └── main.go
├── internal/                     # Private application code (not for external use)
│   ├── broker/                   # Broker core logic
│   │   ├── server.go             # Main broker server struct and lifecycle
│   │   ├── api_handler.go        # Request/response handling (Produce, Fetch, etc.)
│   │   ├── log_manager.go        # Partition log management
│   │   ├── replica_manager.go    # Leader/follower replication
│   │   └── coordinator.go        # Group coordination (if embedded)
│   ├── client/                   # Client libraries
│   │   ├── producer/             # Producer client implementation
│   │   │   ├── producer.go       # Public Producer interface
│   │   │   ├── accumulator.go    # Batch accumulation logic
│   │   │   ├── partitioner.go    # Partition selection strategies
│   │   │   └── sender.go         # Network sender with retries
│   │   └── consumer/             # Consumer client implementation
│   │       ├── consumer.go       # Public Consumer interface
│   │       ├── fetcher.go        # Record fetching logic
│   │       ├── group_member.go   # Consumer group membership
│   │       └── offset_tracker.go # Offset commit/fetch management
│   ├── protocol/                 # Wire protocol definitions
│   │   ├── messages.go           # Request/response structs (ProduceRequest, etc.)
│   │   ├── serialization.go      # Binary encoding/decoding
│   │   └── api_keys.go           # API key constants
│   ├── storage/                  # Persistent storage layer
│   │   ├── log_segment.go        # Log segment with index file
│   │   ├── wal.go                # Write-ahead log abstraction
│   │   └── offset_store.go       # Consumer offset persistence
│   ├── types/                    # Core domain types
│   │   ├── topic.go              # Topic and Partition types
│   │   ├── record.go             # Record and RecordBatch types
│   │   └── metadata.go           # Cluster metadata types
│   └── network/                  # Network abstraction
│       ├── connection.go         # TCP connection management
│       ├── request_handler.go    # Request routing
│       └── server.go             # Generic TCP server
├── pkg/                          # Public libraries (if any)
│   └── publicapi/                # Stable public APIs for external use
└── test/                         # Integration tests and utilities
    ├── integration/              # End-to-end integration tests
    └── helpers.go                # Test utilities
```

Package Rationale:

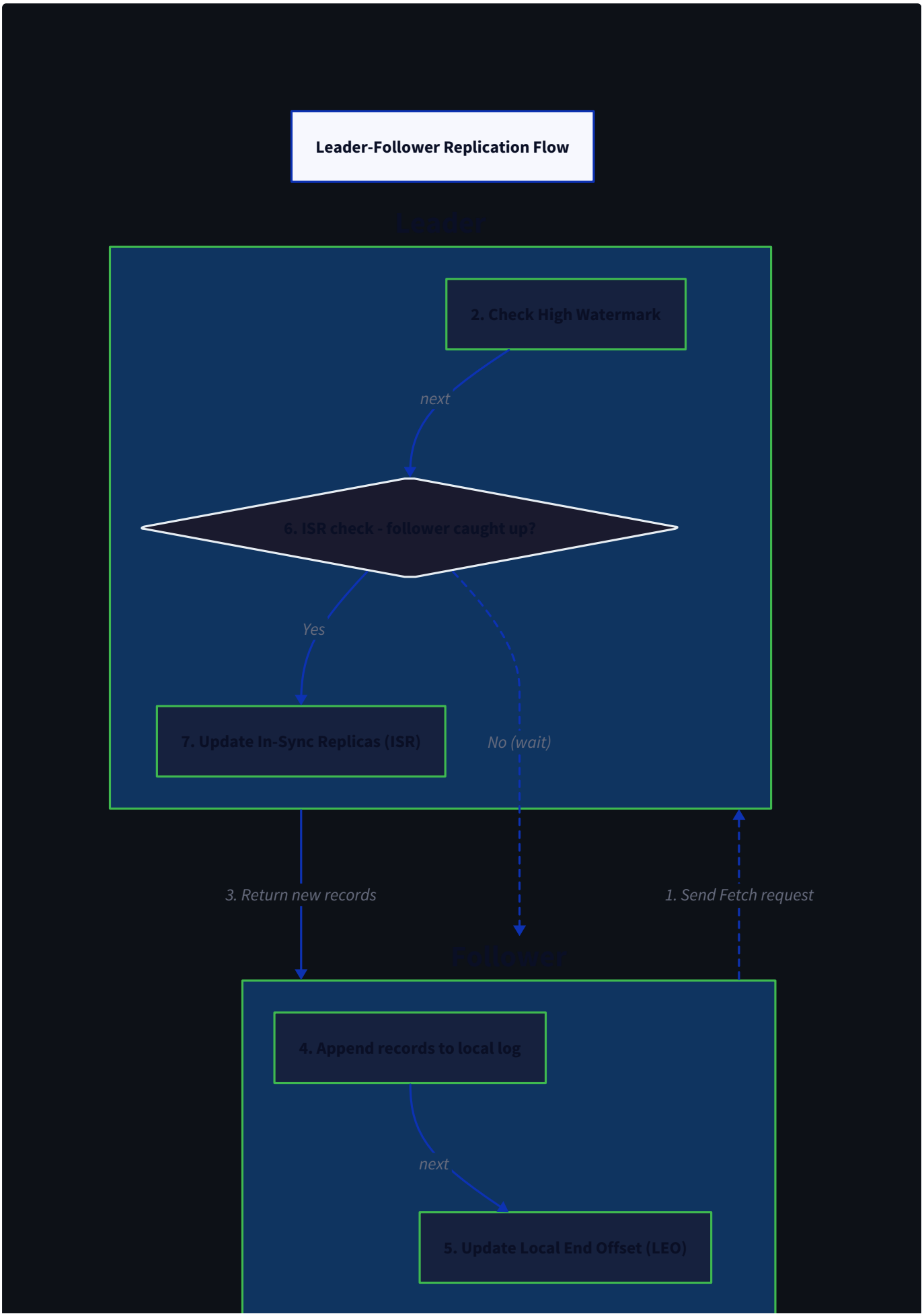
- **`cmd/`**: Follows Go conventions for executable applications. Separating broker, producer-cli, and consumer-cli allows independent development and testing.
- **`internal/`**: Contains implementation details that should not be imported by external code. This prevents dependency entanglement and maintains clean API boundaries.
- **`internal/broker/`**: Houses the server-side logic. Separating `log_manager`, `replica_manager`, and `coordinator` allows each to evolve independently and be tested in isolation.
- **`internal/client/`**: Contains client libraries. Producers and consumers are separate packages as they have distinct concerns, though they share some protocol code.
- **`internal/protocol/`**: Centralizes wire format definitions. This is critical because brokers and clients must agree exactly on binary formats.

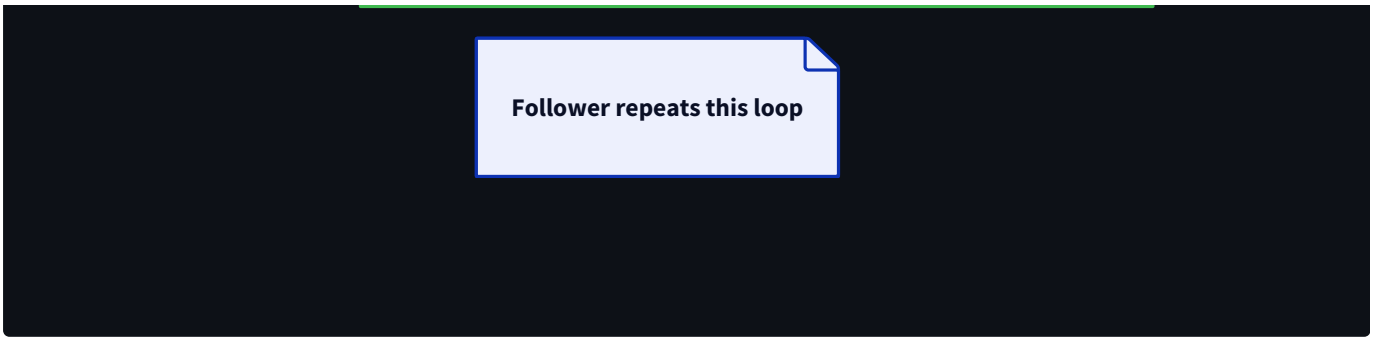
- `internal/storage/` : Abstracts persistence concerns. This separation allows swapping storage backends (e.g., file system vs. embedded database) without affecting higher layers.
- `internal/types/` : Defines core domain objects shared across packages. Keeping these in a central location prevents circular dependencies.

> **Design Principle**: Each package should have a single, clear responsibility. Dependencies should flow downward: higher-level packages (`broker` , `client`) depend on lower-level ones (`protocol` , `storage` , `types`), but not vice versa.

3.3 End-to-End Message Flow

To understand how components interact, let's trace the journey of a single message through the system. We'll follow a concrete example: a weather service (producer) publishing a temperature reading to a `weather-updates` topic, which is then consumed by a dashboard application (consumer).


Leader-Follower Replication Flow
Leader
2. Check High Watermark
next
6. ISR check - follower caught up?
Yes
7. Update In-Sync Replicas (ISR)
No (wait)
3. Return new records
1. Send Fetch request
Follower
4. Append records to local log
next
5. Update Local End Offset (LEO)



Scenario Setup:

- **Topic**: `weather-updates` with 3 partitions
- **Message Key**: `station-42` (identifies the weather station)
- **Message Value**: `{"temp": 22.5, "humidity": 65}`
- **Producer**: Configured with `acks=all` (wait for all ISR replicas)
- **Consumer Group**: `dashboard-group` with 2 consumer instances

Step-by-Step Flow:

1. **Producer Initialization and Metadata Fetch**
   - The producer starts with a bootstrap broker address (e.g., `broker1:9092`).
   - It connects to the bootstrap broker and sends a `MetadataRequest` for the `weather-updates` topic.
   - The broker responds with metadata including:
     - Partition count (3)
     - Leader broker for each partition (e.g., partition 0 → broker2, partition 1 → broker1, partition 2 → broker3)
     - Replica brokers for each partition
   - The producer caches this metadata for future requests (refreshing periodically or on errors).
2. **Message Routing to Partition**
   - The producer receives the message with key `station-42`.
   - It applies the partitioner logic: hash the key, modulo partition count.
   - `hash("station-42") % 3 = 1` → message routes to partition 1.
   - The producer knows broker1 leads partition 1 (from cached metadata).
3. **Batch Accumulation**
   - The producer maintains a per-partition batch accumulator.
   - The message is added to the batch for partition 1.
   - If the batch is not yet full and the linger time hasn't expired, the producer waits for more messages to batch.
   - Once batch is full or timer fires, the batch is marked ready for sending.
4. **Produce Request to Leader**
   - The producer sends a `ProduceRequest` to broker1 (leader for partition 1).
   - The request contains:
     - Topic: `weather-updates`
     - Partition: 1
     - Record batch with our message (and potentially other batched messages)
     - Required acks: `all` (wait for all ISR replicas)
   - The producer starts a retry timer and stores the batch in an "in-flight" buffer.
5. **Leader Log Append**

- Broker1 receives the `ProduceRequest` and validates permissions, topic existence, etc.
- It appends the record batch to its local log for partition 1.
- The log manager assigns the next sequential offset (e.g., offset 142) to the first record in the batch.
- The leader writes the batch to its write-ahead log and calls `fsync()` based on durability configuration.
- The leader updates its **log end offset** (LEO) for partition 1 to 143 (142 + 1 record).

6. **Replication to Followers**
   - Partition 1 has replicas on broker2 and broker3 (followers).
   - Followers periodically send `FetchRequest` to the leader (or the leader pushes, depending on design).
   - The leader sends the new record batch to followers in response to their fetch requests.
   - Each follower appends the batch to its own log and acknowledges to the leader.
   - The leader tracks which replicas have caught up. Once all ISR replicas have the record, the leader advances the **high watermark** to offset 143.
7. **Acknowledgment to Producer**
   - With `acks=all`, the leader waits until all ISR replicas have replicated the record.
   - Once the high watermark advances past our record's offset, the leader sends a successful `ProduceResponse` to the producer.
   - The response includes the base offset assigned (142) for the batch.
   - The producer removes the batch from its in-flight buffer and can consider the message durably stored.
8. **Consumer Subscription and Assignment**
   - Meanwhile, two consumers (C1 and C2) in group `dashboard-group` subscribe to topic `weather-updates`.
   - Each consumer sends `JoinGroup` request to the group coordinator (which may be broker1).
   - The coordinator designates one consumer as the group leader and collects subscription preferences.
   - The group leader runs the partition assignment strategy (e.g., range assignment):
     - Partitions 0 and 1 → Consumer C1
     - Partition 2 → Consumer C2
   - The coordinator sends assignments via `SyncGroup` responses.
9. **Consumer Fetch and Delivery**
   - Consumer C1, now assigned partition 1, fetches records starting from its last committed offset (initially 0 or from previous session).
   - It sends a `FetchRequest` to broker1 (leader for partition 1), requesting records from offset 142 onward.
   - Broker1 returns records starting at offset 142, which includes our temperature reading.
   - Consumer C1 delivers the record to the application callback for processing.
   - The consumer periodically commits its offset (e.g., offset 143) via `OffsetCommit` requests.
10. **Offset Commitment**
    - Consumer C1 sends an `OffsetCommit` request to the group coordinator.
    - The coordinator durably stores `{group: dashboard-group, topic: weather-updates, partition: 1, offset: 143}`.
    - On restart, consumer C1 can fetch this committed offset and resume from where it left off.

Failure Scenario Handling

If broker1 fails **after** appending the record but **before** replicating to followers:

1. The producer's retry timer expires (no response received).
2. The producer refreshes metadata and discovers broker1 is down.
3. A new leader election occurs for partition 1 (from ISR set).

4. The producer resends the batch to the new leader.
5. Because the original write may or may not have persisted, we rely on **idempotent sequence numbers** (optional) or accept **at-least-once** delivery semantics.

This end-to-end flow illustrates the coordinated dance between components to deliver durable, ordered messaging while handling failures transparently.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option (Recommended for Learning)	Advanced Option (For Extension)
Network Transport	Plain TCP with custom binary protocol using Go's `net` package	gRPC with Protocol Buffers for type-safe RPCs
Serialization	Manual binary encoding using `encoding/binary`	Apache Avro with schema registry
Storage Backend	Direct filesystem I/O with `os.File`	Embedded key-value store (BadgerDB, BoltDB)
Coordination	In-memory coordinator on designated broker with periodic persistence	External coordination service (etcd, ZooKeeper)
Concurrency	Goroutines + channels + `sync.Mutex` / `sync.RWMutex`	Actor model with third-party library (protoactor-go)

B. Recommended Starter Code Structure

Here's a complete, working foundation for the network and storage layers that learners can build upon:

File: `internal/network/server.go`

```go
package network

import (
    "context"
    "fmt"
    "net"
    "sync"
)

// TCPServer is a generic TCP server that accepts connections and delegates
// handling to a RequestHandler.
type TCPServer struct {
    addr     string
    listener net.Listener
    handler  RequestHandler
    wg       sync.WaitGroup
    ctx      context.Context
    cancel   context.CancelFunc
}

// RequestHandler processes incoming requests from a connection.
type RequestHandler interface {
    Handle(conn net.Conn) error
}

// NewTCPServer creates a new TCP server.
func NewTCPServer(addr string, handler RequestHandler) *TCPServer {
    return &TCPServer{
        addr:    addr,
        handler: handler,
    }
}

// Start begins accepting connections.
func (s *TCPServer) Start() error {
    listener, err := net.Listen("tcp", s.addr)
```

```
    if err != nil {
        return fmt.Errorf("failed to listen on %s: %w", s.addr, err)
    }

    s.listener = listener
    s.ctx, s.cancel = context.WithCancel(context.Background())

    s.wg.Add(1)
    go s.acceptLoop()

    return nil
}

// acceptLoop accepts incoming connections and spawns goroutines to handle them.
func (s *TCPServer) acceptLoop() {
    defer s.wg.Done()

    for {
        select {
        case <-s.ctx.Done():
            return
        default:
            conn, err := s.listener.Accept()
            if err != nil {
                // Log error but continue
                continue
            }

            s.wg.Add(1)
            go func(c net.Conn) {
                defer s.wg.Done()
                defer c.Close()
                s.handler.Handle(c)
            }(conn)
```

```
        }
    }
}

// Stop gracefully shuts down the server.
func (s *TCPServer) Stop() error {
    if s.cancel != nil {
        s.cancel()
    }

    if s.listener != nil {
        s.listener.Close()
    }

    s.wg.Wait()
    return nil
}
```

File: `internal/storage/wal.go`

```go
package storage

import (
    "encoding/binary"
    "fmt"
    "io"
    "os"
    "path/filepath"
    "sync"
)

// WAL implements a simple write-ahead log for durable writes.
type WAL struct {
    file     *os.File
    filePath string
    mu       sync.Mutex
    offset   int64 // Current write offset
}

// OpenWAL opens or creates a WAL file at the given path.
func OpenWAL(filePath string) (*WAL, error) {
    // Ensure directory exists
    dir := filepath.Dir(filePath)
    if err := os.MkdirAll(dir, 0755); err != nil {
        return nil, fmt.Errorf("failed to create directory %s: %w", dir, err)
    }

    // Open file in append mode, create if doesn't exist
    file, err := os.OpenFile(filePath, os.O_RDWR|os.O_CREATE|os.O_APPEND, 0644)
    if err != nil {
        return nil, fmt.Errorf("failed to open WAL file %s: %w", filePath, err)
    }

    // Get current file size
    stat, err := file.Stat()
```

```
    if err != nil {
        file.Close()
        return nil, fmt.Errorf("failed to stat WAL file: %w", err)
    }

    return &WAL{
        file:     file,
        filePath: filePath,
        offset:   stat.Size(),
    }, nil
}

// Append writes data to the WAL and optionally fsyncs.
func (w *WAL) Append(data []byte, sync bool) (int64, error) {
    w.mu.Lock()
    defer w.mu.Unlock()

    // Write length prefix (8 bytes)
    length := uint64(len(data))
    header := make([]byte, 8)
    binary.BigEndian.PutUint64(header, length)

    // Write header
    if _, err := w.file.Write(header); err != nil {
        return 0, fmt.Errorf("failed to write header: %w", err)
    }

    // Write data
    if _, err := w.file.Write(data); err != nil {
        return 0, fmt.Errorf("failed to write data: %w", err)
    }

    // Sync if requested
    if sync {
```

```
        if err := w.file.Sync(); err != nil {
            return 0, fmt.Errorf("failed to sync: %w", err)
        }
    }

    startOffset := w.offset
    w.offset += int64(8 + len(data)) // Update offset for next write

    return startOffset, nil
}

// ReadAll reads all entries from the WAL.
func (w *WAL) ReadAll() ([][]byte, error) {
    w.mu.Lock()
    defer w.mu.Unlock()

    // Seek to beginning
    if _, err := w.file.Seek(0, io.SeekStart); err != nil {
        return nil, fmt.Errorf("failed to seek: %w", err)
    }

    var entries [][]byte

    for {
        // Read length prefix
        var length uint64
        if err := binary.Read(w.file, binary.BigEndian, &length); err != nil {
            if err == io.EOF {
                break
            }
            return nil, fmt.Errorf("failed to read length: %w", err)
        }

        // Read data
```

```
        data := make([]byte, length)
        if _, err := io.ReadFull(w.file, data); err != nil {
            return nil, fmt.Errorf("failed to read data: %w", err)
        }

        entries = append(entries, data)
    }

    return entries, nil
}

// Close closes the WAL file.
func (w *WAL) Close() error {
    w.mu.Lock()
    defer w.mu.Unlock()

    return w.file.Close()
}

// CurrentOffset returns the current write offset.
func (w *WAL) CurrentOffset() int64 {
    w.mu.Lock()
    defer w.mu.Unlock()
    return w.offset
}
```

File: `internal/types/topic.go`

```go
package types

// Topic represents a named stream of records divided into partitions.
type Topic struct {
    Name       string
    Partitions []*Partition
    Config     map[string]string
}

// Partition represents a single ordered, immutable sequence of records.
type Partition struct {
    Topic           string
    ID              int
    LeaderBrokerID  int
    ReplicaBrokerIDs []int
    Log             Log
    // Additional fields for replication
    HighWatermark   int64
    ISR             []int // Broker IDs in the in-sync replica set
}

// Log represents the storage for a partition.
type Log struct {
    Segments      []*LogSegment
    CurrentOffset int64 // Next offset to assign
}

// LogSegment represents a physical file segment of the log.
type LogSegment struct {
    BaseOffset int64  // First offset in this segment
    DataFile   string // Path to data file
    IndexFile  string // Path to index file (offset → position)
}
```

C. Core Logic Skeleton

File: `internal/broker/server.go`

```go
package broker

import (
    "context"
    "sync"

    "github.com/your-org/build-your-own-kafka/internal/network"
    "github.com/your-org/build-your-own-kafka/internal/storage"
    "github.com/your-org/build-your-own-kafka/internal/types"
)

// Server is the main broker server.
type Server struct {
    config    Config
    server    *network.TCPServer
    logDir    string

    // State protected by mutex
    mu          sync.RWMutex
    topics      map[string]*types.Topic
    // TODO: Add other state fields: consumer groups, replica state, etc.

    // Dependencies
    logManager  *LogManager
    coordinator *Coordinator
}

// Config holds broker configuration.
type Config struct {
    Host string
    Port int
    DataDir string
}

// NewServer creates a new broker server.
```

```go
func NewServer(config Config) *Server {
    return &Server{
        config: config,
        topics: make(map[string]*types.Topic),
        logDir: filepath.Join(config.DataDir, "logs"),
    }
}

// Start starts the broker server.
func (s *Server) Start() error {
    // TODO 1: Create log directory if it doesn't exist
    // TODO 2: Initialize log manager to handle partition logs
    // TODO 3: Load existing topics and partitions from disk
    // TODO 4: Initialize coordinator for consumer groups
    // TODO 5: Create and start TCP server with request handler
    // TODO 6: Start background goroutines: replica fetcher, leader election, etc.
    return nil
}

// CreateTopic creates a new topic with the given partition count.
func (s *Server) CreateTopic(name string, partitionCount int) error {
    s.mu.Lock()
    defer s.mu.Unlock()

    // TODO 1: Validate topic name doesn't already exist
    // TODO 2: Validate partition count is positive
    // TODO 3: Create partition objects with empty logs
    // TODO 4: Assign partition leaders (initially this broker)
    // TODO 5: Initialize log segments for each partition
    // TODO 6: Store topic in topics map
    // TODO 7: Persist topic metadata to disk

    return nil
}
```

```go
// HandleProduce handles a produce request from a client.
func (s *Server) HandleProduce(req *protocol.ProduceRequest) (*protocol.ProduceResponse, error) {
    // TODO 1: Validate request: topic exists, partition exists, acks valid
    // TODO 2: For each partition in request:
    //   - Check if this broker is leader for the partition
    //   - Append records to partition log
    //   - Update log end offset
    // TODO 3: Based on acks level:
    //   - acks=0: Return success immediately
    //   - acks=1: Wait for leader write, then return
    //   - acks=all: Wait for all ISR replicas, update high watermark, then return
    // TODO 4: Build response with offsets for each partition
    return nil, nil
}

// HandleFetch handles a fetch request from a consumer.
func (s *Server) HandleFetch(req *protocol.FetchRequest) (*protocol.FetchResponse, error) {
    // TODO 1: Validate request: topic exists, partition exists
    // TODO 2: For each requested partition:
    //   - Check if this broker is leader for the partition
    //   - Read records starting from requested offset up to high watermark
    //   - If offset is beyond high watermark, wait up to max wait time
    // TODO 3: Build response with records for each partition
    return nil, nil
}
```

D. Language-Specific Hints

1. **Concurrency**: Use `sync.RWMutex` for protecting shared broker state (topics, partitions). Readers can acquire read locks concurrently, while writers need exclusive locks.
2. **File I/O**: Always use `file.Sync()` after critical writes to ensure durability. For performance, batch sync operations when possible.
3. **Network Connections**: Use `SetDeadline()` on connections to prevent hung connections from consuming resources.
4. **Error Handling**: In Go, return errors rather than panicking. Use `fmt.Errorf("%w", err)` to wrap errors with context.
5. **Memory Management**: For high-throughput scenarios, reuse byte buffers with `sync.Pool` to reduce garbage collection pressure.
6. **Testing**: Use `net.Listen("tcp", "localhost:0")` to get random ports for integration tests, avoiding port conflicts.

E. Milestone Checkpoint

After implementing the high-level architecture, you should be able to:

1. **Start a broker**: Run `go run cmd/broker/main.go` and see it listen on the configured port (e.g., `:9092`).
2. **Create a topic via API**: Use a simple curl command or test program to call `CreateTopic` and verify the topic appears in the broker's topic list.
3. **Verify directory structure**: Check that the data directory contains subdirectories for each topic partition.
4. **Basic connectivity**: Write a simple test that connects to the broker, sends a ping, and receives a response.

Expected directory structure after creating a topic:

```
data/
└── logs/
    └── weather-updates-0/
        ├── 00000000000000000000.log
        └── 00000000000000000000.index
```

Signs something is wrong:

- Broker fails to start: Check port availability and directory permissions.
- Topic creation fails: Verify partition count is positive and topic name follows naming rules.
- No log files created: Ensure the broker has write permissions to the data directory.

4. Data Model

> **Milestone(s):** 1, 2, 3, 4 (foundational structures) This section defines the foundational data structures that give our message queue its shape, both in memory and on persistent storage. Before diving into component interactions and algorithms, we must establish a precise vocabulary of types and their relationships. These structures embody the core concepts of topics, partitions, logs, and offsets, forming the skeleton upon which all system behavior is built.

4.1 Core Types and Relationships

Think of the data model as a **library system for events**. A `Topic` is like a book series (e.g., "Financial Transactions"). Each book in the series is a `Partition` (e.g., "Volume 1: NYSE", "Volume 2: NASDAQ"). Inside each partition, `Record`s are individual pages, numbered sequentially by `Offset` (page number). `LogSegment`s are the physical bindings that group a contiguous set of pages for storage efficiency. Finally, `ConsumerGroup`s are reading clubs; each member (`Consumer`) checks out specific volumes and keeps a bookmark (`OffsetMetadata`) to remember where they left off.

This mental model clarifies the hierarchy and ownership: a topic *has* partitions, a partition *is* a log, a log *contains* segments, and segments *hold* records. Consumer groups *track* progress per partition. The following tables define each type's fields and purpose.

In-Memory Core Types

These structures are primarily held in memory on brokers and clients to manage runtime state.

Type	Field Name	Type	Description
`Topic`	`Name`	`string`	Unique identifier for the topic (e.g., `"user-events"`).
	`Partitions`	`[]Partition`	The list of partitions that constitute this topic. The length defines the partition count.
	`Config`	`map[string]string`	Key-value settings for the topic (e.g., `"retention.ms": "604800000"`).
`Partition`	`Topic`	`string`	The name of the parent topic.
	`ID`	`int`	The zero-based identifier of this partition within the topic.
	`LeaderBrokerID`	`int`	The broker ID currently acting as the leader for this partition, handling all reads and writes.
	`ReplicaBrokerIDs`	`[]int`	The ordered list of broker IDs hosting replicas of this partition. The first element is typically the leader.
	`Log`	`Log`	The append-only log instance managing the physical storage of records for this partition.
	`HighWatermark`	`int64`	The offset of the last record that is known to be replicated to all In-Sync Replicas (ISR). Consumers cannot read beyond this point.
	`LogEndOffset`	`int64`	The offset that will be assigned to the next record appended to the log (also called LEO).
	`ISR`	`[]int`	The list of broker IDs currently considered "in-sync" (replicas caught up within a configurable lag threshold).
`Record`	`Key`	`[]byte`	Optional key used for partitioning and log compaction. `nil` keys are allowed and handled by a default partitioner.
	`Value`	`[]byte`	The message payload. Can be `nil` for tombstones in log compaction.
	`Timestamp`	`int64`	The time the record was created, in milliseconds since the Unix epoch. Used for retention and client-side ordering.
	`Headers`	`map[string][]byte`	Optional application-specific key-value pairs attached to the record (e.g., tracing IDs, content-type).
`Log`	`Segments`	`[]LogSegment`	The ordered list of segment files comprising the log. The last segment is the active one for appends.
	`CurrentOffset`	`int64`	The next offset to be assigned (identical to `Partition.LogEndOffset`, maintained

Type	Field Name	Type	Description
			for convenience).
	`BaseDir`	`string`	The directory path where segment files for this partition are stored (e.g., `"/data/topic-0"`).
`LogSegment`	`BaseOffset`	`int64`	The offset of the first record contained in this segment. Used for filename generation and binary search.
	`DataFile`	`string`	Path to the primary data file (e.g., `"00000000000000000000.log"`) holding the record batches.
	`IndexFile`	`string`	Path to the sparse index file (e.g., `"00000000000000000000.index"`) mapping record offsets to byte positions in the data file.
	`SizeBytes`	`int64`	Current size of the data file in bytes. Used to trigger rolling to a new segment.
`ConsumerGroup`	`GroupID`	`string`	Unique identifier for the consumer group (e.g., `"email-processors"`).
	`Members`	`map[string]ConsumerMetadata`	Map of consumer member IDs to their metadata (subscriptions, assigned partitions).
	`State`	`GroupState`	Current group state: `Stable`, `PreparingRebalance`, `Dead`.
	`GenerationID`	`int32`	Monotonically increasing integer incremented on each successful rebalancing. Used to identify obsolete requests.
`ConsumerMetadata`	`MemberID`	`string`	Unique ID assigned by the coordinator for this group member session.
	`ClientID`	`string`	Client-provided identifier for debugging.
	`SubscribedTopics`	`[]string`	List of topics this consumer has subscribed to.
	`AssignedPartitions`	`map[string][]int32`	Map from topic to list of partition IDs assigned to this member after rebalancing.
	`LastHeartbeat`	`int64`	Unix timestamp (ms) of the last received heartbeat. Used for failure detection.
`OffsetMetadata`	`Partition`	`PartitionID`	Composite identifier (Topic, PartitionID) for the partition.
	`GroupID`	`string`	The consumer group this offset belongs to.
	`Offset`	`int64`	The last successfully processed record offset. The consumer will resume from `Offset + 1`.

Type	Field Name	Type	Description
	`Metadata`	`string`	Optional client-provided string (e.g., a commit message).
	`CommitTimestamp`	`int64`	When this offset was committed (ms since epoch).

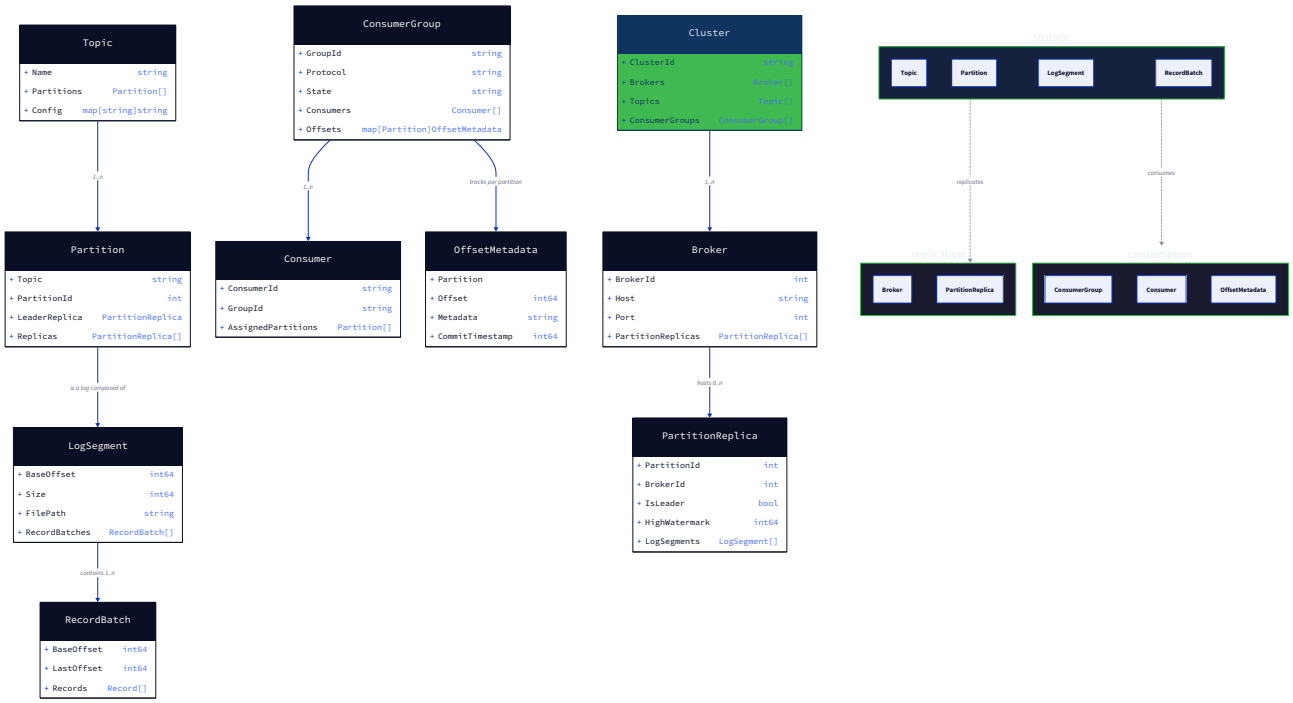
Type Relationships and Lifetime

The relationships between these types form a directed graph. A `Topic` aggregates `Partition` objects. Each `Partition` owns a single `Log` instance. The `Log` manages a list of `LogSegment` objects, which are created, rolled, and deleted over time. `Record` objects are ephemeral; they are created by producers, serialized into batches, and written to segments. Once written, they are primarily accessed via byte offsets, not as in-memory objects.

`ConsumerGroup` and `OffsetMetadata` have a separate lifecycle. Groups are created when the first consumer joins and persist until all members leave or a session timeout expires. Offsets are committed by consumers and stored durably, separate from the log data, to survive broker restarts.

> **Design Insight:** The separation of *message data* (in partition logs) from *consumption progress* (in offset storage) is critical. It allows multiple independent consumer groups to read the same topic at their own pace, and enables replay by resetting offsets without modifying the underlying log.

The class diagram below visualizes these relationships:



4.2 On-Disk Storage Format

Our system's durability guarantee hinges on reliably persisting two types of data: **log records** (the messages themselves) and **consumer offsets** (progress bookmarks). The design of the on-disk format must balance read efficiency, write performance, crash consistency, and operational simplicity.

> **Mental Model: The Ship's Logbook and Its Index.** Imagine a captain's logbook (`*.log` file) where entries are written sequentially, day by day. A separate index (`*.index` file) records that "Day 50 starts on page 120." The logbook is append-only; we never modify past entries. The index is sparse—we don't index every entry, only every few pages—to keep it small and fast to search.

Log Segment Files

Each partition's log is split into multiple **segment files** for manageability. Segments are rolled when the active data file reaches a configured size (e.g., 1 GB) or time threshold. The naming convention uses the `BaseOffset` (the offset of the first record in the segment) padded to 20 digits: `{base_offset}.log` and `{base_offset}.index`.

Record Batch Format (`.log` file) Messages are grouped into **batches** for efficiency. The batch is the unit of writing and reading from disk. The binary format is as follows:

Field	Type	Value/Description
`base_offset`	int64	The offset of the first record in this batch. Used for recovery and validation.
`batch_length`	int32	Length in bytes of the batch data (from `partition_leader_epoch` to the end of `records`).
`partition_leader_epoch`	int32	For leader fencing; can be set to 0 in initial implementation.
`magic`	int8	Protocol version (set to `2` for our system, indicating v2 record format).
`crc`	uint32	CRC32C checksum of the batch data (from `magic` to end of `records`). Validated on read.
`attributes`	int16	Bit flags for compression (`0x00` = none, `0x01` = gzip, `0x02` = snappy), timestamp type, and transactional control.
`last_offset_delta`	int32	The difference between the last and first offset in the batch. Allows quick calculation of the last offset without parsing all records.
`first_timestamp`	int64	The timestamp of the first record in the batch.
`max_timestamp`	int64	The maximum timestamp among records in the batch.
`producer_id`	int64	For idempotent producers; set to `-1` if not used.
`producer_epoch`	int16	For idempotent producers; set to `0` if not used.
`base_sequence`	int32	For idempotent producers; set to `0` if not used.
`record_count`	int32	Number of records in the batch.
`records`	[]Record	Array of individual records, each encoded as below.

Individual Record Format (inside batch) Records are encoded within a batch to avoid repeating common metadata.

Field	Type	Description
`length`	varint	Length of the record data in bytes (following this field).
`attributes`	int8	Currently unused (set to `0`).
`timestamp_delta`	varint	Difference from `first_timestamp` in milliseconds.
`offset_delta`	varint	Difference from `base_offset`.
`key_length`	varint	Length of the key bytes, or `-1` for null key.
`key`	bytes	The key bytes (if `key_length >= 0`).
`value_length`	varint	Length of the value bytes, or `-1` for null value.
`value`	bytes	The value bytes (if `value_length >= 0`).
`headers_count`	varint	Number of headers.
`headers`	[]Header	Array of headers, each with a `header_key` (string varint) and `header_value` (bytes varint).

> **Why Batches?** Writing individual records would incur massive overhead per I/O operation. Batching amortizes the cost of the disk seek and filesystem sync across many messages, dramatically increasing throughput. The batch header also allows readers to skip batches (e.g., by offset or timestamp) without parsing individual records.

Sparse Index Format (`.index` file) The index file provides a mapping from **offset** to **byte position** in the corresponding `.log` file. It is sparse to remain small; we index only every `index.interval.bytes` (e.g., 4KB) of log data. Each entry is two 8-byte values:

Field	Type	Description
`relative_offset`	int32	The offset relative to the segment's `base_offset` (`actual_offset - base_offset`). This allows using a 4-byte integer, saving space.
`position`	int32	The physical byte position of the start of the log batch containing this offset, within the `.log` file.

The index is memory-mapped for fast binary search. To find the position for a target offset `O`: 1) binary search the index for the largest `relative_offset <= (O - base_offset)`, 2) read the `.log` file from the corresponding `position`, and 3) scan forward until reaching offset `O`.

Consumer Offset Storage Format

Consumer offsets must be persisted durably and support fast updates and queries. We have two primary design options:

Decision: Store Offsets in a Special Internal Topic

- **Context**: We need a durable, replicated, and scalable storage for consumer group offsets that aligns with the system's existing replication and fault-tolerance mechanisms.
- **Options Considered**:
  1. **Special internal topic** (`__consumer_offsets`): A compacted topic where each message key is `(GroupID, Topic, Partition)` and the value is the offset metadata.
  2. **Dedicated file per broker**: A local file (e.g., `offsets.json`) storing all offsets for groups managed by that broker's coordinator.
  3. **External key-value store**: Using an embedded DB (like RocksDB/LevelDB) for offset storage.
- **Decision**: Use a special internal, compacted topic (`__consumer_offsets`).
- **Rationale**:
  - **Consistency**: Leverages the same replication, durability, and recovery mechanisms as regular topics, ensuring offsets survive broker failures.
  - **Scalability**: Partitions of the internal topic distribute the load across the cluster.
  - **Operational Simplicity**: No separate storage subsystem to manage; log compaction automatically removes obsolete offset commits.
  - **Educational Value**: Reinforces the power of the log as a universal storage abstraction.
- **Consequences**:
  - Offsets are replicated and fault-tolerant.
  - Requires implementing log compaction (or a simplified version) for cleanup.
  - Introduces a bootstrap dependency: the broker must be able to create and write to this topic before handling consumer groups.

Option	Pros	Cons	Chosen?
Internal Topic	Unified storage, replication, scalability	Requires log compaction; circular dependency risk	**Yes**
Dedicated File	Simple, no extra features needed	Not replicated, single point of failure, scales poorly	No
External KV Store	Fast random updates, mature	Adds a new dependency, operational complexity	No

Offset Topic Record Format The `__consumer_offsets` topic uses the same log segment format, but with a specific key and value schema for its records.

Field	Type	Description
Key		
`group_id`	string	The consumer group identifier.
`topic`	string	The topic name.
`partition`	int32	The partition ID.
Value		
`offset`	int64	The committed offset.
`metadata`	string	Optional client metadata string.
`commit_timestamp`	int64	Timestamp of the commit (ms).
`expire_timestamp`	int64	For session timeouts; can be omitted initially.

This topic should be **compacted** so that only the latest offset for each `(group, topic, partition)` key is retained, saving space.

4.3 Cluster Metadata

Cluster metadata is the **directory of the library**—it tells clients and brokers where to find things. It includes the list of live brokers, which broker leads each partition, and topic configurations. This metadata must be consistently available to all participants and updated dynamically as the cluster changes (brokers join/leave, leadership changes).

Metadata Structures

The following structures are maintained by a **metadata coordinator** (which could be a dedicated broker or a consensus service like Raft) and cached by all brokers and clients.

Type	Field	Type	Description
`Broker`	`ID`	`int`	Unique broker identifier (must be stable across restarts).
	`Host`	`string`	Network hostname or IP address for client connections.
	`Port`	`int`	TCP port for client connections.
	`Rack`	`string`	Optional rack identifier for rack-aware replica placement.
`TopicMetadata`	`Name`	`string`	Topic name.
	`Partitions`	`[]PartitionMetadata`	Metadata for each partition in the topic.
	`Config`	`map[string]string`	Topic-level configuration overrides.
`PartitionMetadata`	`Topic`	`string`	Topic name.
	`PartitionID`	`int`	Partition identifier.
	`LeaderID`	`int`	Broker ID of the current partition leader. `-1` if leader unknown.
	`ReplicaIDs`	`[]int`	Ordered list of broker IDs hosting replicas.
	`ISR`	`[]int`	Subset of `ReplicaIDs` currently in-sync.
`ClusterMetadata`	`Brokers`	`map[int]Broker`	Map of broker ID to broker details.
	`Topics`	`map[string]TopicMetadata`	Map of topic name to topic metadata.
	`ControllerID`	`int`	Broker ID of the current controller (manages leadership elections and partition assignments).
	`ClusterID`	`string`	Unique string identifying the cluster (for multi-cluster routing).

Metadata Propagation and Consistency

Metadata is disseminated through a **gossip-like protocol** or via direct responses to client requests. When a client (producer/consumer) requests metadata (e.g., because of an unknown topic or a `NOT_LEADER` error), it contacts any broker, which returns the current `ClusterMetadata`. Brokers update their metadata upon leadership changes, broker failures, or topic creation.

> **Design Insight:** Eventually consistent metadata is acceptable for our system. Clients may temporarily have stale metadata, leading them to send produce/fetch requests to the wrong broker. The receiving broker must reject such requests with a `NOT_LEADER` error, prompting the client to refresh its metadata. This pattern ensures correctness while avoiding the complexity of a strongly consistent metadata store in the initial implementation.

Metadata Storage

Cluster metadata itself must be persisted to survive controller failures. The controller (a designated broker) stores critical metadata in a durable **metadata log**, which could be a special internal topic (`__cluster_metadata`) or a replicated state machine (like Raft). For simplicity in early milestones, we can store metadata in a file on the controller broker, but this becomes a single point of failure.

> **ADR: Metadata Storage for Milestone 1-3**
>
> - **Context**: We need a simple way to persist topic and partition assignments that is sufficient for the educational project before implementing full replication.
> - **Options**:
>   1. **In-memory only**: No persistence; topics and assignments are lost on broker restart.
>   2. **File on controller**: Write metadata to a JSON or binary file on the controller broker's disk.
>   3. **Replicated log**: Use the same WAL/replication mechanism as data topics.
> - **Decision**: Use a file on the controller broker (`metadata.json`).
> - **Rationale**: Simplicity and focus. The primary learning goal in early milestones is the data path (produce/consume), not metadata high availability. A file is easy to implement and provides basic durability for development.
> - **Consequences**: The controller is a single point of failure. If the controller crashes and its disk is lost, metadata is lost. This is acceptable for learning and can be upgraded to a replicated log in later milestones.

Implementation Guidance

This section provides concrete starter code and structure for implementing the data model in Go.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
On-Disk Record Format	Custom binary format with `encoding/binary`	Use Protobuf/FlatBuffers for schema evolution
Offset Storage	File-per-broker (`offsets.json`)	Internal compacted topic (`__consumer_offsets`)
Metadata Storage	JSON file on controller (`metadata.json`)	Embedded Raft (hashicorp/raft) for consensus
Index Access	Read entire index into `[]byte` , binary search	Memory-map index file (`mmap`) for zero-copy

B. Recommended File/Module Structure

```
build-your-own-kafka/
├── internal/
│   ├── types/                   # Core data structures
│   │   ├── topic.go
│   │   ├── partition.go
│   │   ├── record.go
│   │   ├── log.go
│   │   └── metadata.go
│   ├── storage/                 # On-disk formats and access
│   │   ├── segment.go
│   │   ├── index.go
│   │   ├── wal.go               # Provided WAL wrapper
│   │   └── offset_store.go
│   └── protocol/                # Wire format encoding/decoding
│       ├── record_batch.go
│       ├── request_response.go
│       └── encoding.go
└── pkg/
    └── server/                  # Broker server (uses types and storage)
```

C. Infrastructure Starter Code

Here is a complete, ready-to-use Write-Ahead Log (WAL) wrapper for segment operations. This handles the low-level file appends and syncs, allowing the learner to focus on higher-level log management.

```go
// internal/storage/wal.go

package storage


import (

    "encoding/binary"

    "fmt"

    "os"

    "path/filepath"

    "sync"

)


// WAL represents an append-only log file with optional fsync.

type WAL struct {

    file     *os.File

    filePath string

    mu       sync.Mutex

    offset   int64 // current write offset in bytes

}


// OpenWAL opens or creates a WAL file at the given path.

func OpenWAL(path string) (*WAL, error) {

    dir := filepath.Dir(path)

    if err := os.MkdirAll(dir, 0755); err != nil {

        return nil, fmt.Errorf("create wal dir: %w", err)

    }

    file, err := os.OpenFile(path, os.O_CREATE|os.O_RDWR|os.O_APPEND, 0644)

    if err != nil {

        return nil, fmt.Errorf("open wal file: %w", err)

    }

    stat, err := file.Stat()

    if err != nil {

        file.Close()

        return nil, fmt.Errorf("stat wal file: %w", err)

    }

    return &WAL{
```

```
        file:     file,
        filePath: path,
        offset:   stat.Size(),
    }, nil
}

// Append writes data to the WAL and optionally syncs to disk.
// Returns the file offset at which data was written.
func (w *WAL) Append(data []byte, sync bool) (int64, error) {
    w.mu.Lock()
    defer w.mu.Unlock()

    // Write length prefix (8 bytes) followed by data
    var buf [8]byte
    binary.BigEndian.PutUint64(buf[:], uint64(len(data)))
    if _, err := w.file.Write(buf[:]); err != nil {
        return 0, fmt.Errorf("write length: %w", err)
    }
    if _, err := w.file.Write(data); err != nil {
        return 0, fmt.Errorf("write data: %w", err)
    }
    if sync {
        if err := w.file.Sync(); err != nil {
            return 0, fmt.Errorf("sync: %w", err)
        }
    }
    writtenAt := w.offset
    w.offset += int64(8 + len(data))
    return writtenAt, nil
}

// ReadAll reads all entries from the WAL.
func (w *WAL) ReadAll() ([][]byte, error) {
    w.mu.Lock()
    defer w.mu.Unlock()
```

```
    if _, err := w.file.Seek(0, 0); err != nil {
        return nil, fmt.Errorf("seek to start: %w", err)
    }
    var entries [][]byte
    for {
        var lengthBuf [8]byte
        if _, err := w.file.Read(lengthBuf[:]); err != nil {
            if err.Error() == "EOF" {
                break
            }
            return nil, fmt.Errorf("read length: %w", err)
        }
        length := binary.BigEndian.Uint64(lengthBuf[:])
        data := make([]byte, length)
        if _, err := w.file.Read(data); err != nil {
            return nil, fmt.Errorf("read data: %w", err)
        }
        entries = append(entries, data)
    }
    return entries, nil
}

// CurrentOffset returns the current write offset in bytes.
func (w *WAL) CurrentOffset() int64 {
    w.mu.Lock()
    defer w.mu.Unlock()
    return w.offset
}

// Close closes the WAL file.
func (w *WAL) Close() error {
    w.mu.Lock()
    defer w.mu.Unlock()
    return w.file.Close()
```

```
}
```

D. Core Logic Skeleton Code

Below are skeleton implementations for the key data model types with TODOs that map to the design described above.

```go
// internal/types/record.go

package types

import "time"

type Record struct {
    Key       []byte
    Value     []byte
    Timestamp int64
    Headers   map[string][]byte
}

// NewRecord creates a new Record with the current timestamp if not provided.
func NewRecord(key, value []byte, headers map[string][]byte) *Record {
    return &Record{
        Key:       key,
        Value:     value,
        Timestamp: time.Now().UnixMilli(),
        Headers:   headers,
    }
}
```

```go
// internal/types/log.go

package types


type Log struct {

    Segments      []LogSegment

    CurrentOffset int64

    BaseDir       string

    config        LogConfig

}


type LogConfig struct {

    SegmentMaxBytes int64

    IndexInterval   int

}


// Append writes a batch of records to the log, returning the starting offset.

func (l *Log) Append(records []*Record) (int64, error) {

    // TODO 1: Check if active segment exists and has space (l.Segments[len(l.Segments)-1].SizeBytes < config.SegmentMaxBytes)

    // TODO 2: If no space, roll a new segment (create new data and index files with BaseOffset = l.CurrentOffset)

    // TODO 3: Serialize records into a RecordBatch using the binary format (see protocol/record_batch.go)

    // TODO 4: Write the batch to the active segment's data file (using WAL.Append with sync=true for durability)

    // TODO 5: If written bytes exceed IndexInterval, add an index entry (relative_offset, position) to the index file

    // TODO 6: Update active segment's SizeBytes

    // TODO 7: Update l.CurrentOffset by len(records)

    // TODO 8: Return the starting offset (previous CurrentOffset)

    return 0, nil

}


// Read fetches records starting from the given offset, up to maxBytes.

func (l *Log) Read(startOffset int64, maxBytes int32) ([]*Record, error) {

    // TODO 1: Locate the segment containing startOffset (binary search on Segments[i].BaseOffset)

    // TODO 2: Load the segment's index into memory (if not already cached)

    // TODO 3: Use index to find the approximate byte position in the data file

    // TODO 4: Open data file, seek to position, and scan forward to find the batch with startOffset
```

```
    // TODO 5: Deserialize batches, collect records whose offset >= startOffset

    // TODO 6: Stop when accumulated size exceeds maxBytes or end of log

    return nil, nil

}
```

```go
// internal/storage/segment.go

package storage


type LogSegment struct {

    BaseOffset int64

    DataFile   string

    IndexFile  string

    SizeBytes  int64

    wal        *WAL

    index      *Index

}


// Index manages the sparse offset-to-position mapping.

type Index struct {

    entries []IndexEntry

    mu      sync.RWMutex

}


type IndexEntry struct {

    RelativeOffset int32

    Position       int32

}


// AddEntry adds a new index entry when a batch is written at the given position.

func (idx *Index) AddEntry(offsetDelta int32, position int32) {

    // TODO: Append entry to in-memory slice, periodically flush to index file

}


// FindEntry returns the index entry with the largest relativeOffset <= targetDelta.

func (idx *Index) FindEntry(targetDelta int32) (IndexEntry, bool) {

    // TODO: Binary search on idx.entries

    return IndexEntry{}, false

}
```

E. Language-Specific Hints

- **Serialization**: Use `encoding/binary` with `binary.BigEndian` for portability across networks. For variable-length integers (varint), implement a simple version or use `binary.PutVarint` (though note it's little-endian; you may need to adapt).

- **File I/O**: Use `os.File` with `O_APPEND` for log segments. For fsync, call `file.Sync()` after writes when `acks=all`. Use `bufio.Writer` for batching file writes, but be careful to flush before sync.
- **Concurrency**: Protect in-memory structures like `Log.Segments` and `Index.entries` with `sync.RWMutex`. Multiple goroutines can read concurrently, but writes must be exclusive.
- **Memory Management**: Avoid loading entire segment files into memory. Use `io.ReaderAt` and `io.WriterAt` for random access. Consider memory-mapping index files (`mmap`) for efficient binary search.

F. Milestone Checkpoint (Data Model)

To verify your data model implementation works:

1. Run the unit tests for the `types` and `storage` packages:

```
go test ./internal/types/... ./internal/storage/... -v
```
BASH

2. Create a simple test that creates a `Log` , appends a few records, reads them back, and verifies offsets are sequential and content matches.
3. Inspect the on-disk files: after appending, you should see `00000000000000000000.log` and `00000000000000000000.index` in your data directory. The `.log` file should contain the binary batch format; you can inspect it with a hex dump (`hexdump -C file.log`).

G. Common Pitfalls

⚠️ Pitfall: Incorrect offset arithmetic leading to gaps or duplicates

- **Description**: When appending a batch of 5 records starting at offset 100, the next offset must be 105. A common mistake is incrementing by 1 instead of the batch size.
- **Why it's wrong**: Consumers expect contiguous offsets. A gap causes them to stall waiting for the missing offset; duplicates cause reprocessing.
- **Fix**: Maintain `CurrentOffset` as the *next* offset to assign. When a batch of N records is appended, the starting offset is `CurrentOffset` , and after append, `CurrentOffset += N` .

⚠️ Pitfall: Not fsync'ing when required by acknowledgment level

- **Description**: Writing to the file buffer without calling `Sync()` when `acks=all` can lose data if the OS crashes.
- **Why it's wrong**: The producer receives an acknowledgment, but the data may not be on durable storage, violating durability guarantees.
- **Fix**: In `WAL.Append` , conditionally call `file.Sync()` based on the `sync` parameter, which should be set according to the produce request's `acks` .

⚠️ Pitfall: Storing absolute offsets in index files

- **Description**: Using the full 8-byte offset in index entries instead of a 4-byte relative offset wastes space and limits segment size.
- **Why it's wrong**: Index files become unnecessarily large, reducing the effectiveness of caching. Also, segment base offsets can be very large (e.g., after many writes), making 4-byte absolute offsets insufficient.
- **Fix**: Store `relative_offset = actual_offset - segment_base_offset` as int32. Ensure segment rolling happens before `relative_offset` exceeds `math.MaxInt32` (i.e., segment size limit).

5. Component Design: Broker and Topic Partitions

> **Milestone(s):** 1 (Topic and Partitions)

5.1 Responsibility and Scope

The **Broker** is the foundational server node that stores data and handles client requests. Think of it as a **library branch** in a distributed library system: it houses specific shelves (partitions) of books (messages), accepts new donations (producer writes), and allows patrons (consumers) to check out books for reading. Each broker operates independently but coordinates with others to form a complete, fault-tolerant cluster.

The broker's primary responsibilities are:

1. **Log Storage**: Maintain an immutable, append-only log for each partition it hosts, physically persisted to disk with configurable durability guarantees.
2. **Request Handling**: Serve **Produce** and **Fetch** requests from producers and consumers over a binary TCP protocol, ensuring correct ordering and offset tracking.
3. **Topic Management**: Create and manage topics with their partitions, including partition assignment to brokers and metadata distribution.
4. **Local Offset Tracking**: Store and retrieve consumer group offsets for partitions hosted locally (before replication, this is simply local file storage).
5. **Coordination Participation**: Work with the cluster coordinator (which may be a designated broker) for leader election and metadata synchronization.

The scope of this component in Milestone 1 is intentionally bounded: we implement a **single broker** that can manage multiple topics and partitions locally, without yet handling replication or distributed coordination. This allows learners to solidify the core log storage and request handling patterns before adding complexity.

5.2 Mental Model: The Immutable Ledger

Imagine a traditional **bank ledger book** used to record transactions in chronological order. Each page represents a **segment** of the log, and each line entry is a **record** with a sequential number (offset). The ledger has these key properties:

- **Append-Only**: You never erase or modify past entries; you only add new entries at the end. If a mistake occurs, you add a correction entry rather than changing the original.
- **Immutable History**: Once written, entries become permanent history. This simplifies concurrency—readers can access any part of the ledger without blocking writers.
- **Sequential Offsets**: Each entry gets a monotonically increasing number (0, 1, 2...), which serves as a unique identifier and position indicator within that specific ledger.
- **Partitioned Ledgers**: Instead of one giant ledger for all transactions, the bank maintains separate ledger books for different account types (checking, savings). This is **partitioning**—each ledger (partition) maintains its own independent sequence.

In our system, each **partition** is exactly such a ledger. The broker's primary job is to **guard these ledgers**: ensure writes are appended correctly in order, assign proper offsets, and allow readers to efficiently locate entries by their offset number.

> **Design Insight**: The immutable log abstraction is the core innovation of Kafka-like systems. By treating messages as an ordered, append-only sequence rather than transient queues, we gain durability, replayability, and strong ordering guarantees that enable event sourcing and stream processing.

5.3 Public Interface (APIs)

The broker exposes its functionality through a binary RPC protocol over TCP. While the full wire protocol will be detailed in Section 9, we define the logical API methods here. Clients (producers, consumers, administrative tools) interact with the broker through these request/response pairs.

Core Broker RPC Methods

Method Name	Parameters	Returns	Description
`CreateTopic`	`name` (string), `partitionCount` (int), `replicationFactor` (int), `config` (map[string]string)	`error`	Creates a new topic with the specified number of partitions. In Milestone 1, `replicationFactor` is ignored (set to 1). Initializes empty logs for each partition and updates cluster metadata.
`Produce`	`topic` (string), `partition` (int), `records` ([] `Record`), `requiredAcks` (int), `timeout` (int32)	`baseOffset` (int64), `error`	Appends a batch of records to the specified partition's log. Returns the offset assigned to the first record in the batch. `requiredAcks` controls durability (0=no ack, 1=leader ack, -1=all ISR ack).
`Fetch`	`topic` (string), `partition` (int), `offset` (int64), `maxBytes` (int32)	`records` ([] `Record`), `error`	Retrieves records from the partition log starting at `offset`. Returns up to `maxBytes` worth of records. If the requested offset is beyond the log end, waits for new data up to a timeout.
`GetOffsets`	`topic` (string), `partition` (int), `time` (int64)	`offsets` ([]int64), `error`	Returns offset metadata for a partition. `time` can be -1 (earliest), -2 (latest), or a timestamp. Used by consumers to find starting positions.
`CommitOffsets`	`group` (string), `topic` (string), `partition` (int), `offset` (int64), `metadata` (string)	`error`	Commits a consumer group's offset for a specific partition. Stores the offset locally (later, in an internal `__consumer_offsets` topic).
`FetchOffsets`	`group` (string), `topic` (string), `partition` (int)	`offset` (int64), `metadata` (string), `error`	Retrieves the last committed offset for a consumer group and partition.

Internal Management APIs (Used by Other Brokers/Coordinators)

These methods are not directly called by external clients but are used during replication and coordination (Milestone 4).

Method Name	Parameters	Returns	Description
`LeaderForPartition`	`topic` (string), `partition` (int)	`brokerID` (int), `error`	Returns the broker ID currently acting as leader for the partition. Used by clients for request routing.
`UpdateISR`	`topic` (string), `partition` (int), `isr` ([]int)	`error`	Updates the In-Sync Replica set for a partition. Called by the controller after replication changes.
`FetchReplica`	`topic` (string), `partition` (int), `offset` (int64), `maxBytes` (int32)	`records` ([] `Record`), `error`	Similar to `Fetch` but used by follower brokers to replicate data from the leader. May include additional replication metadata.

State Transition Table for Partition Leadership

As the broker manages partitions, each partition can be in different states regarding leadership and replication. This state machine becomes relevant in Milestone 4 but is introduced here for completeness.

Current State	Event	Next State	Actions Taken
`Offline`	`CreateTopic` command received	`Online` (Follower)	Initialize empty log, set `LeaderBrokerID` to self, set `ISR` = [self], start log appender
`Online` (Leader)	Broker receives `LeaderAndIsr` request with leader = self	`Online` (Leader)	Begin accepting produce requests, start tracking ISR, initialize high watermark to LEO
`Online` (Leader)	Broker receives `LeaderAndIsr` request with leader != self	`Online` (Follower)	Stop accepting produce requests, start fetching from new leader, clear ISR list
`Online` (Follower)	Fetched offset catches up to leader's LEO	`Online` (Follower in ISR)	Broker adds itself to leader's ISR via `UpdateISR` request
Any `Online` state	Broker shutdown initiated	`Offline`	Close log files, persist final offsets, notify coordinator

5.4 Internal Behavior: Log Management

The broker's most critical internal subsystem is the **log manager**, which orchestrates the physical storage and retrieval of records for all partitions. Its algorithm ensures atomic appends, efficient reads, and proper log rolling.

Data Structures for Log Management

The broker maintains the following core in-memory structures (as defined in Section 4):

Structure	Field	Type	Description
`Server`	`topics`	`map[string]*Topic`	Registry of all topics known to this broker, keyed by topic name.
`Server`	`logManager`	`*LogManager`	Central coordinator for all partition logs, handling segment rolling and cleanup.
`Topic`	`Name`	`string`	Unique topic identifier.
`Topic`	`Partitions`	`[]Partition`	List of partitions belonging to this topic.
`Partition`	`Topic`	`string`	Topic name this partition belongs to.
`Partition`	`ID`	`int`	Partition number within the topic (0-indexed).
`Partition`	`LeaderBrokerID`	`int`	ID of broker currently leading this partition (self for local partitions).
`Partition`	`Log`	`Log`	The actual log instance containing segments and current offset.
`Partition`	`HighWatermark`	`int64`	Offset up to which all ISR replicas have replicated (for consumers).
`Partition`	`LogEndOffset`	`int64`	The offset of the next message to be appended (LEO).
`Log`	`Segments`	`[]LogSegment`	Ordered list of log segment files, from oldest to newest.
`Log`	`CurrentOffset`	`int64`	The next offset to be assigned (equal to LEO).
`Log`	`BaseDir`	`string`	Directory path where segment files are stored.
`LogSegment`	`BaseOffset`	`int64`	Offset of the first record in this segment.
`LogSegment`	`DataFile`	`string`	Path to the `.log` file containing record batches.
`LogSegment`	`IndexFile`	`string`	Path to the `.index` file mapping offsets to byte positions.
`LogSegment`	`SizeBytes`	`int64`	Current size of the data file in bytes.

Algorithm: Appending Records to a Partition Log

When a `Produce` request arrives for a valid partition, the broker executes this append algorithm:

1. **Validate Request**: Check that the partition exists, the broker is the leader (for now, assume true), and the request size is within configured limits.
2. **Assign Offsets**: For each record in the incoming batch, assign a sequential offset starting from the partition's `LogEndOffset`. The first record gets offset = `LogEndOffset`, the second gets `LogEndOffset + 1`, etc.
3. **Serialize Batch**: Encode the records along with their assigned offsets, timestamps, and headers into the binary **RecordBatch** format (see Section 9.2).
4. **Append to Active Segment**: Write the serialized bytes to the current active segment's data file. The write is performed using the `WAL.Append` method which ensures atomic appends.
5. **Update Index**: For every N records (configurable `IndexInterval`), add an entry to the segment's sparse index mapping the offset delta to the byte position in the data file.
6. **Update In-Memory State**: Increment the partition's `LogEndOffset` by the number of records appended. If `requiredAcks` is 1 or -1, also update the `HighWatermark` (for now, set equal to `LogEndOffset`).
7. **Conditional Sync**: If `requiredAcks` is -1 (all ISR), call `fsync` on the data file to flush OS buffers to disk. For acks=1, we may defer sync for performance.
8. **Roll Segment if Needed**: If the active segment's `SizeBytes` exceeds `SegmentMaxBytes` (e.g., 1 GB), close it and create a new segment with `BaseOffset` = the new `LogEndOffset`.
9. **Return Response**: Send the `baseOffset` (the offset assigned to the first record) back to the producer.

Algorithm: Reading Records from a Partition Log

When a `Fetch` request arrives, the broker must efficiently locate and return records:

1. **Validate Offset**: Check that the requested offset is between the partition's earliest offset (base offset of oldest segment) and `LogEndOffset`. If offset = `LogEndOffset`, optionally wait for new data.
2. **Locate Segment**: Perform binary search on the partition's `Segments` slice to find the segment where `segment.BaseOffset <= targetOffset < nextSegment.BaseOffset`.
3. **Consult Index**: Load the segment's index (if not already in memory). Use `Index.FindEntry` to find the largest index entry with offset <= targetOffset, giving a byte position in the data file.
4. **Seek and Scan**: Open the data file, seek to the byte position, and read forward until reaching `targetOffset`. Since records are stored in batches, we may need to scan through a batch to find the exact starting record.
5. **Collect Records**: Read sequential records until either hitting `maxBytes` limit, reaching end of segment, or exceeding a maximum record count.
6. **Return Results**: Package the records into a `FetchResponse`. Include the `HighWatermark` so consumers know the safe read limit.

Log Segment Rolling and Cleanup

The log manager periodically checks segments and performs maintenance:

1. **Size-Based Rolling**: After each append, if the active segment exceeds `SegmentMaxBytes`, it is rolled.
2. **Time-Based Rolling**: A background thread checks segments older than `SegmentRollTime` (e.g., 7 days) and rolls the active segment if it's too old.
3. **Deletion Policy**: For simplicity, we implement a **retention-based cleanup**. A background task runs every 5 minutes:
   - For each partition, list segments sorted by `BaseOffset`.
   - If total log size exceeds `RetentionBytes` or the oldest segment's timestamp is older than `RetentionTime`, delete the oldest segment file and its index.
   - Update the `Segments` slice accordingly.

> **Design Insight**: Segment-based storage (rather than one giant file per partition) enables efficient old data deletion and parallel I/O. The sparse index trades minor CPU overhead during reads for significant disk space savings, as we only index every N records.

5.5 ADR: Log Storage Backend

> **Decision: File-Based Append-Only Segments with Sparse Indexing**
>
> - **Context**: We need a persistent storage engine for partition logs that provides high-throughput sequential writes, efficient random reads by offset, and easy deletion of old data. The solution must be simple to implement and understand for educational purposes while being performant enough for the core use case.
> - **Options Considered**:
>   1. **Simple file-based append-only logs with custom sparse indexing** (Kafka's approach)
>   2. **Embedded key-value store** (e.g., LevelDB/RocksDB, SQLite)
>   3. **Memory-mapped files with byte-addressable offset indexing**
> - **Decision**: Option 1 — file-based append-only logs with sparse indexing.
> - **Rationale**:
>   - **Conceptual Simplicity**: The mental model of "segments as files" maps directly to the immutable ledger analogy, making it easier for learners to debug and reason about.
>   - **Predictable Performance**: Sequential disk writes are the fastest possible I/O pattern, and reads are mostly sequential scans after index lookup. This matches Kafka's proven performance profile.
>   - **Explicit Control**: We avoid the complexity and black-box nature of an embedded database. Learners understand exactly when data is persisted, indexed, and deleted.
>   - **Ease of Inspection**: Log segments are plain files viewable with hex editors or custom tools, aiding debugging and education.
> - **Consequences**:
>   - **Positive**: Excellent write throughput, straightforward implementation, easy to add compression or encryption per segment.
>   - **Negative**: Manual management of file handles and indexes required. Need to implement our own recovery logic after crashes. No built-in transactions—we must ensure atomic batch writes ourselves.

Option	Pros	Cons	Chosen?
File-based segments	Maximum write throughput, simple mental model, easy inspection and debugging, matches Kafka's design for direct learning	Manual file management, need to implement own indexing and recovery, concurrent access requires careful locking	**Yes** - best for educational goals and performance
Embedded KV store	Built-in recovery, concurrent access handled, potentially simpler API, good for prototyping	Black-box behavior, harder to debug, often optimized for random writes not sequential appends, adds dependency	No - obscures learning objectives
Memory-mapped files	Excellent read performance, OS handles paging, simple offset-to-address arithmetic	Complex to handle file growth, portability concerns, risk of SIGBUS on truncated files, still need custom index	No - increased complexity without sufficient benefit

5.6 Common Pitfalls

⚠️ Pitfall: Forgetting to fsync or syncing too often

Description: After writing records to a file, developers may omit calling `file.Sync()` (or equivalent) to flush OS buffers to disk. Conversely, calling sync after every single write destroys throughput.

Why it's wrong: Without fsync, a broker crash can lose recently "acknowledged" messages (if acks=1 or all), violating durability guarantees. With excessive fsync, throughput drops to disk seek speed (often < 1000 writes/sec).

How to fix: Implement a **sync policy** based on `requiredAcks` and batch size. For acks=1, sync periodically (e.g., every 100ms or 1MB of data). For acks=-1 (all ISR), sync immediately after each write. Use the `WAL.Append` method's `sync` parameter to control this.

⚠️ Pitfall: Incorrect offset gaps or reuse

Description: Assigning non-sequential offsets (e.g., skipping numbers) or reusing offsets after a crash recovery.

Why it's wrong: Consumers rely on strictly monotonic offsets for progress tracking. Gaps confuse offset commit logic and may cause infinite loops. Reused offsets cause duplicate processing and break idempotency.

How to fix: Always derive new offsets from `LogEndOffset`, which must be persisted (in the segment file names or a checkpoint file). On startup, scan segments to reconstruct the exact `LogEndOffset`. Never decrement it.

⚠️ Pitfall: Poor key hashing leading to partition skew

Description: Using a naive hash function (like Java's `Object.hashCode()` equivalent) or not handling null keys properly, causing uneven distribution across partitions.

Why it's wrong: Skewed partition distribution defeats the purpose of partitioning—one partition becomes a hotspot while others are underutilized, limiting scalability.

How to fix: Use a proven hash function (e.g., MurmurHash2/3) and always apply a modulo operation with the number of partitions. For null keys, use a round-robin or random partition assignment to spread load.

⚠️ Pitfall: Not handling concurrent reads and writes

Description: Allowing simultaneous read and write operations on the same log segment without proper synchronization, leading to corrupted reads or panics.

Why it's wrong: Concurrent reads may see partially written records or incorrect offsets. File descriptors may be closed while in use.

How to fix: Use a **reader-writer lock** per partition. Multiple fetches can read concurrently, but an append acquires an exclusive lock. For higher throughput, consider copy-on-write semantics for the active segment.

⚠️ Pitfall: Infinite memory growth from unclosed segments

Description: Keeping all segment file handles and index data in memory forever, causing memory exhaustion as the log grows.

Why it's wrong: A long-running broker will eventually run out of memory and crash, especially with many partitions.

How to fix: Implement an **LRU cache** for index data. Close file handles for inactive segments (not the active one). Reopen files on demand when reads target old segments.

⚠️ Pitfall: Ignoring disk space exhaustion

Description: Appending records without checking available disk space, causing write failures that may corrupt the log.

Why it's wrong: When disk is full, writes may partially succeed, leaving torn records. The broker might crash inconsistently.

How to fix: Before each append, check free space against a configurable threshold (e.g., 1GB). If below threshold, reject writes with an appropriate error. Implement alerting for operators.

5.7 Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Log Storage	Plain files with `os.File`, manual indexing	Memory-mapped files (`mmap`) with atomic appends
Concurrency Control	`sync.RWMutex` per partition	Lock-free ring buffer for appends, RCU for reads
Serialization	Custom binary format with `encoding/binary`	Protocol Buffers or FlatBuffers for schema evolution
File Operations	Direct `os.Open` / `Close` per operation	Pooled file descriptors with reference counting

B. Recommended File/Module Structure

```
build-your-own-kafka/
├── cmd/
│   ├── broker/
│   │   └── main.go                # Broker entry point
│   └── client/                    # Producer/consumer CLI tools
├── internal/
│   ├── broker/
│   │   ├── server.go              # Main Server struct and lifecycle
│   │   ├── handler.go             # Request handling (Produce, Fetch, etc.)
│   │   └── log_manager.go         # LogManager orchestrating partitions
│   ├── storage/
│   │   ├── log.go                 # Partition Log implementation
│   │   ├── segment.go             # LogSegment with index
│   │   ├── wal.go                 # Write-ahead log wrapper (provided below)
│   │   └── index.go               # Sparse index implementation
│   ├── types/
│   │   ├── topic.go               # Topic and Partition structs
│   │   ├── record.go              # Record and RecordBatch
│   │   └── metadata.go            # Broker, ClusterMetadata
│   └── protocol/
│       ├── request.go             # Request structs and parsing
│       └── response.go            # Response building
└── pkg/
    └── wal/                       # Reusable WAL package (could be external)
```

C. Infrastructure Starter Code: WAL Wrapper

Below is a complete, ready-to-use Write-Ahead Log wrapper. This handles atomic appends with optional fsync, which is crucial for durability. Learners can import this directly.

```go
// internal/storage/wal.go

package storage


import (

    "fmt"

    "os"

    "sync"

)


// WAL implements a simple write-ahead log with atomic appends.

type WAL struct {

    file     *os.File

    filePath string

    mu       sync.Mutex

    offset   int64 // current write offset in bytes

}


// OpenWAL opens or creates a WAL file at the given path.

func OpenWAL(path string) (*WAL, error) {

    file, err := os.OpenFile(path, os.O_RDWR|os.O_CREATE|os.O_APPEND, 0644)

    if err != nil {

        return nil, fmt.Errorf("failed to open WAL file: %w", err)

    }



    stat, err := file.Stat()

    if err != nil {

        file.Close()

        return nil, fmt.Errorf("failed to stat WAL file: %w", err)

    }



    return &WAL{

        file:     file,

        filePath: path,

        offset:   stat.Size(),

    }, nil
```

```
}

// Append writes data to the WAL with optional fsync.
// Returns the file offset at which data was written.
func (w *WAL) Append(data []byte, sync bool) (int64, error) {
    w.mu.Lock()
    defer w.mu.Unlock()

    offset := w.offset
    n, err := w.file.Write(data)
    if err != nil {
        return 0, fmt.Errorf("write failed: %w", err)
    }
    if n != len(data) {
        return 0, fmt.Errorf("short write: %d != %d", n, len(data))
    }

    w.offset += int64(n)

    if sync {
        if err := w.file.Sync(); err != nil {
            // We still wrote the data, but sync failed.
            // The file system may have buffered it; we can't guarantee durability.
            return offset, fmt.Errorf("fsync failed: %w", err)
        }
    }

    return offset, nil
}

// ReadAll reads all entries from the WAL (for recovery).
func (w *WAL) ReadAll() ([][]byte, error) {
    w.mu.Lock()
    defer w.mu.Unlock()
```

```go
    if _, err := w.file.Seek(0, 0); err != nil {
        return nil, fmt.Errorf("seek failed: %w", err)
    }

    var entries [][]byte
    buffer := make([]byte, 4096)
    for {
        n, err := w.file.Read(buffer)
        if n > 0 {
            // In a real implementation, you'd need to parse record boundaries.
            // For simplicity, we assume each write is a complete entry.
            entry := make([]byte, n)
            copy(entry, buffer[:n])
            entries = append(entries, entry)
        }
        if err != nil {
            if err.Error() == "EOF" {
                break
            }
            return entries, fmt.Errorf("read error: %w", err)
        }
    }

    return entries, nil
}

// CurrentOffset returns the current write offset in bytes.
func (w *WAL) CurrentOffset() int64 {
    w.mu.Lock()
    defer w.mu.Unlock()
    return w.offset
}
```

```
// Close closes the WAL file.
func (w *WAL) Close() error {
    w.mu.Lock()
    defer w.mu.Unlock()
    return w.file.Close()
}
```

D. Core Logic Skeleton Code

Below are skeleton implementations for the key components that learners should fill in. Each TODO corresponds to a step in the algorithms described in Section 5.4.

Log Manager (orchestrates all partition logs):

```go
// internal/storage/log.go

package storage


import (

    "fmt"

    "path/filepath"

    "sync"

)


// Log represents a partition's log composed of multiple segments.

type Log struct {

    Segments      []LogSegment

    CurrentOffset int64

    BaseDir       string

    config        LogConfig

    mu            sync.RWMutex

}


// LogConfig holds configuration for log behavior.

type LogConfig struct {

    SegmentMaxBytes int64

    IndexInterval   int

    RetentionBytes  int64

}


// Append writes a batch of records to the log and returns the starting offset.

func (l *Log) Append(records []*Record) (int64, error) {

    l.mu.Lock()

    defer l.mu.Unlock()


    // TODO 1: Validate input - ensure records slice is not empty


    // TODO 2: Assign sequential offsets starting from CurrentOffset

    //         For each record, set Record.Offset = CurrentOffset + i


```

```
    // TODO 3: Serialize records into RecordBatch binary format
    //         Use encodeRecordBatch() helper (to be implemented)

    // TODO 4: Get active segment (create first segment if none exists)
    //         activeSegment := l.getOrCreateActiveSegment()

    // TODO 5: Write batch bytes to segment data file using WAL.Append
    //         Call with sync=false for now (we'll add sync policy later)

    // TODO 6: Update segment size and add index entry if needed
    //         if recordsWritten % IndexInterval == 0, add index entry

    // TODO 7: Update CurrentOffset by adding len(records)

    // TODO 8: Check if segment should be rolled (size > SegmentMaxBytes)
    //         if yes, call l.rollSegment()

    // TODO 9: Return the starting offset (CurrentOffset before increment)

    return 0, fmt.Errorf("not implemented")
}

// Read fetches records starting from the given offset.
func (l *Log) Read(startOffset int64, maxBytes int32) ([]*Record, error) {
    l.mu.RLock()
    defer l.mu.RUnlock()

    // TODO 1: Validate startOffset is within range [oldestSegment.BaseOffset, CurrentOffset]

    // TODO 2: Locate segment containing startOffset using binary search
    //         segment := l.findSegment(startOffset)

    // TODO 3: Load segment index (lazy load if not in memory)
```

```
    // TODO 4: Use index to find approximate position in data file
    //         position := segment.Index.FindEntry(offsetDelta)

    // TODO 5: Open data file, seek to position, and scan forward to startOffset

    // TODO 6: Read records sequentially until hitting maxBytes or segment end

    // TODO 7: Decode RecordBatch(es) into []*Record

    // TODO 8: Return records (may be empty if startOffset == CurrentOffset)

    return nil, fmt.Errorf("not implemented")
}

// getOrCreateActiveSegment returns the last segment if it has space, else creates new.
func (l *Log) getOrCreateActiveSegment() (*LogSegment, error) {
    // TODO 1: If Segments is empty, create first segment with BaseOffset = CurrentOffset

    // TODO 2: Get last segment, if its SizeBytes < config.SegmentMaxBytes, return it

    // TODO 3: Otherwise, roll a new segment
    return l.rollSegment()
}

// rollSegment closes the current active segment and creates a new one.
func (l *Log) rollSegment() (*LogSegment, error) {
    // TODO 1: If there's an active segment, close its files

    // TODO 2: Create new segment with BaseOffset = CurrentOffset

    // TODO 3: Generate filenames: {BaseOffset}.log and {BaseOffset}.index

    // TODO 4: Open data file and index file
```

```
    // TODO 5: Append new segment to l.Segments



    // TODO 6: Return the new segment

    return nil, fmt.Errorf("not implemented")

}
```

Broker Request Handler:

```go
// internal/broker/handler.go

package broker

import (
    "context"
    "fmt"
    "github.com/yourusername/byok/internal/protocol"
    "github.com/yourusername/byok/internal/storage"
    "github.com/yourusername/byok/internal/types"
)

// HandleProduce processes a Produce request from a client.
func (s *Server) HandleProduce(req *protocol.ProduceRequest) (*protocol.ProduceResponse, error) {
    // TODO 1: Validate topic exists (look up in s.topics map)

    // TODO 2: Validate partition exists and this broker is leader
    //         partition := topic.Partitions[req.Partition]

    // TODO 3: Convert protocol.RecordBatch to []*types.Record

    // TODO 4: Call partition.Log.Append(records)

    // TODO 5: Handle requiredAcks:
    //         - If 0: return immediately with success
    //         - If 1: wait for append to complete (already done), return
    //         - If -1: (future) wait for ISR replication

    // TODO 6: Build response with base offset and any errors

    // TODO 7: Update partition.HighWatermark (for now = LogEndOffset)

    return &protocol.ProduceResponse{
        BaseOffset: -1, // placeholder
    }, fmt.Errorf("not implemented")
```

```
}

// HandleFetch processes a Fetch request from a consumer.
func (s *Server) HandleFetch(req *protocol.FetchRequest) (*protocol.FetchResponse, error) {
    // TODO 1: Validate topic and partition


    // TODO 2: Check requested offset against partition.HighWatermark
    //         Consumers should not read beyond HighWatermark


    // TODO 3: Call partition.Log.Read(startOffset, maxBytes)


    // TODO 4: Convert []*types.Record to protocol.RecordBatch for response


    // TODO 5: Include HighWatermark in response so consumer knows safe read limit


    return &protocol.FetchResponse{}, fmt.Errorf("not implemented")
}
```

E. Language-Specific Hints

- **File Sync**: Use `file.Sync()` for durability. On Linux, this maps to the `fsync()` system call.
- **Binary Encoding**: Use `encoding/binary` with `binary.BigEndian` for network-portable format (Kafka uses big-endian).
- **Concurrent Maps**: Use `sync.RWMutex` to protect the `topics` map. Consider `sync.Map` for read-heavy workloads once initialized.
- **Path Operations**: Always use `filepath.Join()` instead of string concatenation for cross-platform compatibility.
- **Error Handling**: Use `fmt.Errorf("... %w", err)` to wrap errors with context for debugging.
- **Clean Resource Management**: Implement `Close()` methods for all resources (files, network connections) and use `defer` appropriately.

F. Milestone Checkpoint

After implementing the broker core and log storage:

1. **Run the test suite**:

```
go test ./internal/storage/... -v
go test ./internal/broker/... -v
```

2. **Expected output**: All tests should pass, demonstrating:
    - Log appends return sequential offsets
    - Reading from an offset returns correct records
    - Segment rolling occurs when size limit reached
    - Topic creation creates appropriate directory structure

3. **Manual verification**:
    - Start the broker: `go run cmd/broker/main.go --data-dir /tmp/byok`
    - Use a simple CLI producer (to be built in Milestone 2) or `netcat` to send a Produce request:

```
    echo -n "test message" | nc localhost 9092
```

- Check that a segment file appears in `/tmp/byok/topics/test/0/`
- Verify offset tracking by reading the log via a simple dump utility.

4. **Signs of trouble**:
    - **"No such file or directory"**: Check that `BaseDir` is created with `os.MkdirAll`.
    - **Offsets not sequential**: Ensure `CurrentOffset` is updated atomically with the write.
    - **High memory usage**: You might be keeping all segment files open; implement LRU closing.
    - **Test hangs**: Check for deadlocks—use `go test -race` to detect data races.

6. Component Design: Producer

> **Milestone(s):** 2 (Producer)

6.1 Responsibility and Scope

The **Producer** is a client library that applications use to publish (write) records to topics. Its primary responsibility is to take messages from the application, efficiently route them to the correct partition leader, and ensure they are durably written according to the configured acknowledgment semantics. It shields the application from the complexities of the distributed system, such as broker discovery, leader changes, network retries, and batching for performance.

The Producer's scope encompasses:

1. **Message Serialization:** Converting application-provided keys, values, and headers into the binary `Record` and `RecordBatch` format understood by brokers.
2. **Topic Metadata Management:** Discovering which broker leads each partition of a topic, and refreshing this information when leadership changes.
3. **Partition Selection:** Determining the target partition for a message, using key-based hashing for ordering guarantees or round-robin for load distribution.
4. **Batching and Accumulation:** Grouping multiple messages destined for the same partition leader into single network requests to amortize overhead and achieve high throughput.
5. **Reliability Semantics:** Implementing configurable acknowledgment levels (0, 1, `all`) and corresponding retry logic with exponential backoff to provide at-least-once delivery guarantees.
6. **Connection Management:** Maintaining efficient, persistent network connections to broker leaders and handling connection failures.

Crucially, the Producer is stateless regarding message content; it does not store messages durably itself. Its state is limited to in-flight message batches, metadata caches, and retry counters.

6.2 Mental Model: The Postal Batch Sorter

Imagine you run a large postal service. Every day, individuals (applications) bring you millions of letters (messages) to send. Each letter has a destination city and street (topic and key). Your goal is to deliver them reliably and quickly, but sending each letter individually by courier is prohibitively expensive.

You set up a system of sorting bins:

1. **Sorting by Destination:** You first look up the correct regional sorting facility for each city (broker leader for a partition). Letters for the same facility go into the same large bin (`RecordBatch`).
2. **Batch Dispatch:** Once a bin is full or a timer expires, you seal it and send the entire bin via a single truck (network request) to the regional facility. This is far more efficient than sending individual letters.
3. **Delivery Receipts:** For important letters, you request a signed receipt (`ack`). If the receipt doesn't arrive, you put a copy of the letter in a new bin and try again (retry). For less critical mail, you might just trust the postal system (`acks=0`).
4. **Route Changes:** If a regional facility tells you they're no longer handling a certain street, you update your destination map (metadata) and re-route future letters accordingly.

This model captures the essence of the producer: **accumulation by destination, batch dispatch, and managed reliability.** The "postal service" (producer) handles the complexity so the "individual" (application) has a simple interface: just hand over the letter.

6.3 Public Interface

The Producer exposes a simple, synchronous or asynchronous API for sending messages, along with configuration structs to control its behavior.

Configuration (`ProducerConfig`): The behavior of the producer is finely tuned via a configuration struct passed at creation time.

Field	Type	Description	Default
`BootstrapServers`	`[]string`	Initial list of broker addresses (host:port) for metadata discovery.	(required)
`ClientID`	`string`	Logical name for this producer client, used in logs and server-side quotas.	`"byok-producer"`
`Acks`	`AcksLevel`	Durability guarantee: `AcksNone` (0), `AcksLeader` (1), or `AcksAll` (-1).	`AcksLeader`
`Retries`	`int`	Maximum number of retries for a failed batch before giving up.	`3`
`RetryBackoffMs`	`int`	Base milliseconds to wait before the first retry. Exponential backoff applies.	`100`
`BatchSize`	`int`	Maximum number of bytes to include in a batch before sending.	`16384` (16KB)
`LingerMs`	`int`	Milliseconds to wait for additional messages to fill a batch before sending.	`5`
`BufferMemory`	`int64`	Total bytes of memory the producer can use for unsent batches.	`33554432` (32MB)
`Partitioner`	`Partitioner`	Class implementing partition selection logic (e.g., `HashPartitioner` , `RoundRobinPartitioner`).	`HashPartitioner`
`CompressionType`	`CompressionType`	Compression algorithm for record batches (`none` , `gzip` , `snappy`).	`CompressionNone`

Core API Methods: The primary interaction is through a `Producer` struct with the following methods.

Method	Parameters	Returns	Description
`Send`	`ctx context.Context`, `record *Record`	`(int64, error)`	Synchronously sends a single record. Blocks until the record is acknowledged according to the `Acks` setting, or until `ctx` times out/cancels. Returns the partition and offset the record was assigned, or an error.
`SendAsync`	`record *Record`, `callback func(int64, error)`	`error`	Asynchronously sends a record. The callback is invoked when the send succeeds or fails. Returns immediately unless the internal buffers are full (`ErrBufferFull`).
`Flush`	`ctx context.Context`	`error`	Blocks until all currently buffered (unsent) records are transmitted and acknowledged. Used for graceful shutdown or to ensure delivery at specific points.
`Close`	`ctx context.Context`	`error`	Gracefully shuts down the producer. Flushes any buffered records, waits for pending callbacks, and closes all network connections.

Example Usage:

```go
// Creating a producer
config := &ProducerConfig{
    BootstrapServers: []string{"broker1:9092", "broker2:9092"},
    ClientID:         "my-app",
    Acks:             AcksAll,
    Retries:          5,
}
producer, err := NewProducer(config)
if err != nil { ... }
defer producer.Close(context.Background())

// Synchronous send
offset, err := producer.Send(ctx, &Record{
    Topic: "orders",
    Key:   []byte("order-123"),
    Value: []byte(`{"status": "shipped"}`),
})
if err != nil { ... }
fmt.Printf("Record written to partition %d at offset %d\n", partition, offset)
```

6.4 Internal Behavior: Batching and Sending

Internally, the producer is a complex state machine orchestrating metadata, accumulation, and network I/O across multiple goroutines. Its core algorithm can be broken down into the following steps for a synchronous `Send`.

1. Metadata Fetch and Partition Selection: When a record arrives for a topic, the producer first checks its local metadata cache.

1. If the topic metadata is unknown or stale (e.g., last refresh > `metadata.max.age.ms`), it sends a `MetadataRequest` to one of the bootstrap brokers. The response populates the cache with the list of partitions and their current leader brokers.
2. Using the configured `Partitioner` (e.g., `HashPartitioner`), the producer computes the target partition ID. For a non-nil key, it hashes the key and applies modulus to the partition count. For a nil key, it may use a sticky partition strategy or round-robin.
3. The metadata cache provides the `BrokerID` (and network address) of the leader for that partition.

2. Record Accumulation (Batching): The producer maintains a map of in-memory buffers keyed by `TopicPartition` (Topic + Partition ID). Each buffer holds a `RecordBatch` under construction.

1. The serialized record is appended to the `RecordBatch` for its target `TopicPartition`.
2. The batch is not immediately sent. It waits until either:
   - The total byte size of the batch exceeds `BatchSize`.
   - The `LingerMs` timer for that batch expires.
   - The `Flush` or `Close` method is called. This batching is critical for throughput, as it amortizes the fixed cost of a network round-trip and disk I/O over many records.

3. Batch Dispatch and In-Flight Management: When a batch is ready to send, it is handed off to a **Sender** goroutine.

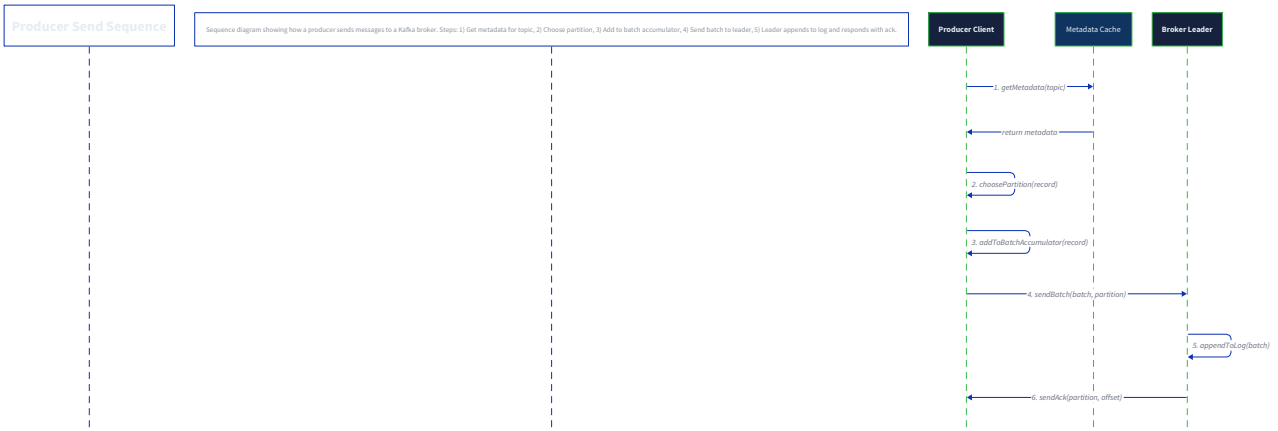
1. The Sender manages a pool of network connections, one per broker leader. It retrieves or creates a connection to the leader broker for the batch's partition.
2. It wraps the `RecordBatch` in a `ProduceRequest` and sends it over the network.
3. The batch is moved to an "in-flight" map, keyed by `TopicPartition`. This tracks batches awaiting acknowledgment.
4. The Sender can multiplex batches for the same broker into a single network request for further efficiency.

4. Acknowledgment Handling and Retry: The producer awaits the broker's `ProduceResponse`.

1. **Success:** If the response indicates success (and meets the `Acks` level), the in-flight batch is removed. For a synchronous `Send`, the calling goroutine is unblocked with the assigned offset. For `SendAsync`, the user's callback is invoked with the offset.
2. **Retriable Error:** Errors like `NotLeaderForPartition`, `NetworkException`, or `Timeout` trigger a retry. a. The batch is re-queued to the accumulator for the same `TopicPartition`. b. The producer's metadata for that topic is marked as stale, forcing a refresh before the next send attempt (to discover the new leader). c. A retry delay is calculated using exponential backoff: `delay = RetryBackoffMs * 2^(attempt)`. The batch is scheduled for re-dispatch after this delay.
3. **Fatal Error:** Errors like `InvalidTopic`, `RecordTooLarge`, or exceeding `Retries` cause the batch to fail permanently. The error is reported to the application.

5. Ordering and Idempotence (Optional): To guarantee exactly-once semantics and preserve ordering across retries, an idempotent producer assigns a monotonic sequence number per `TopicPartition`. The broker rejects duplicates based on this number. For our educational project, this is an advanced extension, but the design should consider leaving room for it (e.g., a `ProducerID` and `SequenceNumber` field in the `RecordBatch`).

The following sequence diagram illustrates this flow for a successful send:



6.5 ADR: Acknowledgment Semantics

Decision: Support Three Acknowledgement Levels (0, 1, all)

- **Context:** The producer must offer trade-offs between durability and latency. Different applications have different needs: a logging system may tolerate some data loss for maximum speed, while a financial transaction processor cannot.
- **Options Considered:**
  1. **Fire-and-forget (`acks=0`):** The producer sends the message and does not wait for any acknowledgment from the broker.
  2. **Leader acknowledgment (`acks=1`):** The producer waits for the partition leader to have written the record to its local log before considering the send successful.
  3. **Full ISR acknowledgment (`acks=all`):** The producer waits for the record to be written to the local log of *all* in-sync replicas (ISR) before success.
- **Decision:** Implement all three levels (`AcksNone` , `AcksLeader` , `AcksAll`) as configurable options.
- **Rationale:** This mirrors Apache Kafka's approach and provides a clear, practical spectrum of durability guarantees for learners to experiment with. Implementing all three demonstrates the incremental complexity: `acks=0` is trivial, `acks=1` introduces waiting for a network response, and `acks=all` requires understanding replication and the High Watermark. This graduated complexity is ideal for learning.
- **Consequences:** The broker's `HandleProduce` method must implement logic for each `acks` level. `AcksAll` requires the broker to track the High Watermark and may introduce higher latency. The producer must handle potential timeouts for `acks=all` if an ISR replica is slow.

Option	Pros	Cons	Chosen?
`acks=0` (Fire-and-forget)	Lowest latency, maximum throughput. Simple to implement.	Possible data loss if broker fails before writing. No backpressure signal.	**Yes** – for throughput-critical, loss-tolerant use cases.
`acks=1` (Leader ack)	Good balance. Protects against leader process crash (record is on disk). Moderate latency.	Data loss possible if leader crashes *after* ack but before replicas copy the data (failover to a non-replica).	**Yes** – the default balance for many applications.
`acks=all` (ISR ack)	Highest durability. Survives `f` broker failures if replication factor > `f+1` .	Highest latency (waits for slowest ISR). Throughput limited by slow replica. Can block if ISR shrinks.	**Yes** – for critical data where loss is unacceptable.

6.6 Common Pitfalls

⚠️ Pitfall: Blocking on a Full Buffer

- **Description:** The `Send` method blocks indefinitely if the total size of unsent batches (`BufferMemory`) is exceeded and the producer cannot drain batches fast enough (e.g., due to slow network or broker).
- **Why it's wrong:** This can cause application threads to hang, leading to a cascading failure. It violates the principle of graceful degradation.
- **Fix:** Implement a bounded wait with a timeout in `Send`, or design `SendAsync` to return an `ErrBufferFull` immediately if the buffer is full, allowing the application to apply backpressure or shed load.

⚠️ Pitfall: Ignoring Leader Changes During Send

- **Description:** A producer caches partition leader metadata. If a leader fails and a new election occurs after the producer has chosen a partition but before it sends the batch, the batch will be sent to the wrong (old leader) broker.
- **Why it's wrong:** The send fails with `NotLeaderForPartition`, incurring a retry delay and unnecessary load on the old leader. In a volatile cluster, this can cause a storm of misdirected requests.
- **Fix:** On receiving `NotLeaderForPartition` or `UnknownTopicPartition` errors, immediately invalidate the cached metadata for that topic and refresh it before the next retry. The metadata should also be refreshed periodically.

⚠️ Pitfall: Duplicate Messages on Retry

- **Description:** When a `ProduceRequest` times out at the producer, it's ambiguous whether the broker processed it. A retry may cause the same record to be appended twice to the log.
- **Why it's wrong:** Consumers will process duplicate records, breaking at-least-once semantics and potentially causing incorrect application state (e.g., double-charging).
- **Fix (Basic):** For `acks=all`, duplicate writes are less likely but still possible on timeout. Educate learners that with retries, the system provides **at-least-once** semantics. Application logic must be idempotent.
- **Fix (Advanced):** Implement the idempotent producer with sequence numbers to allow the broker to deduplicate, enabling **exactly-once** semantics in the log.

⚠️ Pitfall: Poor Key Hashing Leading to Skewed Partitions

- **Description:** Using a naive hash function (like Java's `Object.hashCode()` or Go's default for strings) or applying modulus to a non-power-of-two partition count can lead to uneven distribution of records across partitions.
- **Why it's wrong:** Some partitions become hotspots, limiting the overall throughput of the topic and causing uneven load on brokers.
- **Fix:** Use a robust, deterministic hash function (like MurmurHash2/3) on the key bytes. When performing modulus, ensure the hash value is non-negative. For nil keys, use a sticky random partitioner that batches records to the same partition for a short time to improve batching efficiency.

6.7 Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Network I/O	Synchronous I/O with `net.Dial` and `io.Read/Write`. Simple to understand.	Asynchronous I/O using `goroutines` per connection and channels for batching. Better performance.
Serialization	Manual byte slice construction using `binary.Write` and `append`. Clear and direct.	Protocol Buffer definitions (`protobuf`) for request/response formats. More maintainable and extensible.
Partitioner	Hash-based using `crc32` or `fnv`. Round-robin with a counter.	Implement the "sticky partitioner" from Kafka: fills one batch per partition before moving to the next for better batching.
Compression	None initially.	Integrate `compress/gzip` or a third-party `snappy` library for record batches.

B. Recommended File/Module Structure

Add the following files for the producer client library. It should be a separate package that applications can import.

```
project-root/
  cmd/producer-cli/          # Optional: example CLI tool for testing
    main.go
  internal/producer/         # Producer client library
    producer.go              # Main Producer struct and public API
    config.go                # ProducerConfig and constants
    accumulator.go           # Batch accumulation logic
    sender.go                # Network sender and in-flight management
    partitioner.go           # HashPartitioner, RoundRobinPartitioner
    record_batch.go          # RecordBatch serialization format
    errors.go                # Producer-specific errors (e.g., ErrBufferFull)
  internal/protocol/         # Shared request/response structs and serialization
    produce.go               # ProduceRequest, ProduceResponse
    metadata.go              # MetadataRequest, MetadataResponse
    types.go                 # Common types (Record, RecordBatch)
```

C. Infrastructure Starter Code

Here is a complete, ready-to-use `RecordBatch` serialization helper. This handles the binary format for a batch of records, which is the unit of writing for both the producer and the broker's log.

```go
// internal/protocol/record_batch.go

package protocol

import (
    "bytes"
    "encoding/binary"
    "time"
)

// RecordBatch represents a batch of records as written to the log.
// This is a simplified version. A full implementation includes crc, attributes, etc.
type RecordBatch struct {
    BaseOffset      int64
    PartitionLeaderEpoch int32 // Used for leader epoch tracking (advanced)
    MagicByte       int8   // Version of the batch format (set to 2)
    CRC             uint32 // CRC of the batch data (after this field)
    Attributes      int16  // Bitmask for compression, timestamp type, etc.
    LastOffsetDelta int32  // Delta from BaseOffset for the last record
    FirstTimestamp  int64  // Timestamp of the first record
    MaxTimestamp    int64  // Max timestamp in the batch
    ProducerID      int64  // For idempotence (-1 for none)
    ProducerEpoch   int16  // For idempotence
    BaseSequence    int32  // For idempotence
    Records         []*Record
}

// Record is an individual message.
type Record struct {
    Length     int32  // Delta from the start of the batch
    Attributes int8   // Currently unused
    TimestampDelta int64 // Delta from FirstTimestamp
    OffsetDelta int32 // Delta from BaseOffset
    Key        []byte
    Value      []byte
    Headers    []Header
```

```
}

type Header struct {
    Key   string
    Value []byte
}

// Encode converts the RecordBatch to its on-wire bytes.
func (rb *RecordBatch) Encode() ([]byte, error) {
    buf := new(bytes.Buffer)
    // 1. Write the fixed-size header fields.
    //    Note: We write a placeholder for CRC, compute it later, then overwrite.
    binary.Write(buf, binary.BigEndian, rb.BaseOffset)
    binary.Write(buf, binary.BigEndian, rb.PartitionLeaderEpoch)
    binary.Write(buf, binary.BigEndian, rb.MagicByte)
    crcPos := buf.Len()
    binary.Write(buf, binary.BigEndian, uint32(0)) // Placeholder CRC
    binary.Write(buf, binary.BigEndian, rb.Attributes)
    binary.Write(buf, binary.BigEndian, rb.LastOffsetDelta)
    binary.Write(buf, binary.BigEndian, rb.FirstTimestamp)
    binary.Write(buf, binary.BigEndian, rb.MaxTimestamp)
    binary.Write(buf, binary.BigEndian, rb.ProducerID)
    binary.Write(buf, binary.BigEndian, rb.ProducerEpoch)
    binary.Write(buf, binary.BigEndian, rb.BaseSequence)
    binary.Write(buf, binary.BigEndian, int32(len(rb.Records)))

    // 2. Encode each record.
    for _, rec := range rb.Records {
        // ... (Encode record length, attributes, deltas, key, value, headers)
        // This is detailed serialization logic.
    }

    // 3. Compute CRC over the bytes starting from Attributes to the end.
    data := buf.Bytes()
    crc := computeCRC(data[crcPos+4:]) // Skip placeholder CRC
```

```
    // Overwrite the placeholder CRC.
    binary.BigEndian.PutUint32(data[crcPos:], crc)
    return data, nil
}

// DecodeRecordBatch parses bytes into a RecordBatch.
func DecodeRecordBatch(data []byte) (*RecordBatch, error) {
    // Implementation omitted for brevity.
    // Reads fields in order, validates CRC, decodes records.
    return &RecordBatch{}, nil
}

func computeCRC(data []byte) uint32 {
    // Use a CRC32 implementation (e.g., github.com/klauspost/crc32)
    return 0 // Placeholder
}
```

D. Core Logic Skeleton Code

1. Accumulator (`accumulator.go`): Manages batching per TopicPartition.

```go
// internal/producer/accumulator.go

package producer

import (
    "sync"
    "time"
    "github.com/yourusername/byok/internal/protocol"
)

type TopicPartition struct {
    Topic     string
    Partition int32
}

type Accumulator struct {
    config *ProducerConfig
    batches map[TopicPartition]*RecordBatch
    mu      sync.RWMutex
    cond    *sync.Cond // Used to signal sender when a batch is ready
    closed  bool
}

func NewAccumulator(config *ProducerConfig) *Accumulator {
    acc := &Accumulator{
        config: config,
        batches: make(map[TopicPartition]*RecordBatch),
    }
    acc.cond = sync.NewCond(&acc.mu)
    return acc
}

// Append adds a record to the batch for the given TopicPartition.
// It returns the batch if it's ready to send (size or linger), otherwise nil.
func (a *Accumulator) Append(tp TopicPartition, record *protocol.Record) *protocol.RecordBatch {
    a.mu.Lock()
    defer a.mu.Unlock()
```

```
    batch, exists := a.batches[tp]
    if !exists {
        batch = &protocol.RecordBatch{
            FirstTimestamp: time.Now().UnixMilli(),
            Records:        []*protocol.Record{},
        }
        a.batches[tp] = batch
        // TODO 1: Start a linger timer for this batch in a goroutine.
        //         When the timer fires, call a.readyBatch(tp).
    }

    // TODO 2: Serialize the record to estimate its size in bytes.
    // TODO 3: Add the record to batch.Records.
    // TODO 4: Update batch.LastOffsetDelta and batch.MaxTimestamp.
    // TODO 5: If the batch's estimated size >= config.BatchSize, mark it ready.
    //         Call a.readyBatch(tp) and return the batch.

    return nil // Return nil if batch not ready yet.
}

// readyBatch marks a batch as ready to send and notifies the sender.
func (a *Accumulator) readyBatch(tp TopicPartition) {
    a.mu.Lock()
    defer a.mu.Unlock()
    batch, ok := a.batches[tp]
    if !ok {
        return
    }
    // TODO 6: Remove the batch from the `batches` map.
    // TODO 7: Signal the waiting sender goroutine via a.cond.Signal().
}

// GetReadyBatch blocks until a batch is ready or the accumulator is closed.
// Called by the sender goroutine.
func (a *Accumulator) GetReadyBatch() (TopicPartition, *protocol.RecordBatch) {
```

```
    a.mu.Lock()
    defer a.mu.Unlock()
    for !a.closed {
        // TODO 8: Iterate through `batches` and find one marked as ready.
        //         If found, remove it and return it.
        // TODO 9: If none ready, wait on a.cond.
    }
    return TopicPartition{}, nil
}

func (a *Accumulator) Close() {
    a.mu.Lock()
    defer a.mu.Unlock()
    a.closed = true
    a.cond.Broadcast()
}
```

2. Partitioner (`partitioner.go`): Selects a partition for a record.

```go
// internal/producer/partitioner.go

package producer


import (

    "hash"

    "hash/fnv"

)


type Partitioner interface {

    Partition(topic string, key []byte, numPartitions int32) (int32, error)

}


type HashPartitioner struct {

    hasher hash.Hash32

}


func NewHashPartitioner() *HashPartitioner {

    return &HashPartitioner{hasher: fnv.New32a()}

}


func (p *HashPartitioner) Partition(topic string, key []byte, numPartitions int32) (int32, error) {

    if numPartitions <= 0 {

        return 0, ErrInvalidPartitionCount

    }

    if key == nil {

        // TODO 1: Handle nil key. Common strategy: round-robin across partitions.

        //         You may need state (a counter) per topic. Use sync/atomic.

        return 0, nil

    }

    p.hasher.Reset()

    p.hasher.Write(key)

    hash := int32(p.hasher.Sum32())

    // Ensure non-negative partition.

    partition := (hash & 0x7FFFFFFF) % numPartitions

    return partition, nil
```

```
}
```

3. Sender (`sender.go`): Manages network communication and retries.

```go
// internal/producer/sender.go

package producer

import (
    "context"
    "sync"
    "time"
    "github.com/yourusername/byok/internal/protocol"
)

type Sender struct {
    config        *ProducerConfig
    accumulator   *Accumulator
    metadataCache *MetadataCache
    inFlight      map[TopicPartition]*InFlightBatch
    mu            sync.RWMutex
    connPool      *ConnectionPool
    retryQueue    chan *RetryItem
    done          chan struct{}
}

func NewSender(config *ProducerConfig, acc *Accumulator, meta *MetadataCache) *Sender {
    s := &Sender{
        config:        config,
        accumulator:   acc,
        metadataCache: meta,
        inFlight:      make(map[TopicPartition]*InFlightBatch),
        connPool:      NewConnectionPool(),
        retryQueue:    make(chan *RetryItem, 1000),
        done:          make(chan struct{}),
    }
    go s.run()
    go s.retryLoop()
    return s
}
```

```go
func (s *Sender) run() {
    for {
        select {
        case <-s.done:
            return
        default:
            tp, batch := s.accumulator.GetReadyBatch()
            if batch == nil {
                continue // Accumulator closed
            }
            go s.sendBatch(tp, batch)
        }
    }
}

func (s *Sender) sendBatch(tp TopicPartition, batch *protocol.RecordBatch) {
    var attempt int
    for attempt = 0; attempt <= s.config.Retries; attempt++ {
        // TODO 1: Look up the leader broker for tp using metadataCache.
        //         If metadata is stale, refresh it.
        leaderID, err := s.metadataCache.Leader(tp.Topic, tp.Partition)
        if err != nil {
            // TODO: Handle error, maybe schedule metadata refresh.
            continue
        }
        // TODO 2: Get a connection to the leader from connPool.
        conn, err := s.connPool.Get(leaderID)
        if err != nil {
            // TODO: Handle connection error, maybe mark broker as dead.
            continue
        }

        // TODO 3: Encode the batch into a ProduceRequest.
        req := &protocol.ProduceRequest{
```

```
            Topic:      tp.Topic,
            Partition:  tp.Partition,
            RecordBatch: batch,
            Acks:       s.config.Acks,
        }
        // TODO 4: Send the request and receive a response.
        resp, err := conn.SendProduceRequest(req)
        if err != nil {
            // Network error. Schedule retry.
            s.scheduleRetry(tp, batch, attempt)
            return
        }

        // TODO 5: Check the response error code.
        if resp.ErrorCode == protocol.ErrNone {
            // Success! Notify the original caller (via callback or channel).
            s.completeBatch(tp, batch, resp.Offset)
            return
        } else if isRetriableError(resp.ErrorCode) {
            // Retriable error (e.g., NotLeaderForPartition).
            s.metadataCache.MarkStale(tp.Topic)
            s.scheduleRetry(tp, batch, attempt)
            return
        } else {
            // Fatal error. Fail the batch permanently.
            s.failBatch(tp, batch, resp.ErrorCode)
            return
        }
    }
    // Exhausted retries.
    s.failBatch(tp, batch, protocol.ErrRetriesExhausted)
}

func (s *Sender) scheduleRetry(tp TopicPartition, batch *protocol.RecordBatch, attempt int) {
```

```
    delay := time.Duration(s.config.RetryBackoffMs) * time.Millisecond * (1 << attempt)
    item := &RetryItem{
        tp:    tp,
        batch: batch,
        time:  time.Now().Add(delay),
    }
    // TODO: Implement a priority queue based on `time` instead of a simple channel.
    go func() {
        time.Sleep(delay)
        select {
        case s.retryQueue <- item:
        case <-s.done:
        }
    }()
}

func (s *Sender) retryLoop() {
    for {
        select {
        case <-s.done:
            return
        case item := <-s.retryQueue:
            // TODO: Re-insert the batch into the accumulator for the same tp.
            // This will cause it to be picked up by the sender again.
        }
    }
}

func (s *Sender) completeBatch(tp TopicPartition, batch *protocol.RecordBatch, offset int64) {
    // TODO: Invoke the user's callback or unblock the synchronous Send.
    s.mu.Lock()
    delete(s.inFlight, tp)
    s.mu.Unlock()
}
```

```
func (s *Sender) failBatch(tp TopicPartition, batch *protocol.RecordBatch, err protocol.ErrorCode) {

    // TODO: Invoke the user's callback with error or return error for synchronous Send.

    s.mu.Lock()

    delete(s.inFlight, tp)

    s.mu.Unlock()

}


func (s *Sender) Close() {

    close(s.done)

    s.connPool.Close()

}
```

E. Language-Specific Hints (Go)

1. **Concurrency:** Use a separate goroutine for the main sender loop (`run`) and the retry loop. Use `sync.Cond` for efficient waiting between the accumulator and sender. Protect shared maps (`batches` , `inFlight`) with `sync.RWMutex` .
2. **Network Connections:** Implement a `ConnectionPool` that reuses `net.Conn` for each broker. Set `TCPKeepAlive` and reasonable read/write deadlines. Handle connection errors gracefully by evicting the broken connection from the pool.
3. **Context Propagation:** Use `context.Context` in the public `Send` and `Flush` methods to allow cancellation and timeouts from the application. Propagate this context through the call chain to network requests.
4. **Error Types:** Define specific error types (e.g., `ErrBufferFull` , `ErrTopicNotFound`) in `errors.go` for clear error handling.

F. Milestone Checkpoint

To verify your producer implementation works with the broker from Milestone 1:

1. **Test Setup:** Start a single broker with a topic `test-topic` (2 partitions).
2. **Write a Test Producer:** Create a simple program that uses your producer library to send 100 messages with sequential keys.
3. **Expected Behavior:**
   - Messages should be written to the broker's log segments. Check the data directory for new `.log` files.
   - Use the broker's `HandleFetch` API (or a simple test consumer) to read back the messages. They should be present in order within each partition.
   - For `acks=all` , test by killing the broker leader after a send but before acknowledgment; the producer should retry and eventually succeed once a new leader is elected (Milestone 4).
4. **Signs of Trouble:**
   - **No messages appear:** Check network connectivity, broker logs for errors, and that the producer is looking up the correct leader.
   - **Messages are duplicated:** Your retry logic is likely not handling timeouts correctly. Ensure you are not retrying on ambiguous errors without idempotence.
   - **Producer hangs:** The `BufferMemory` might be too small, or the sender goroutine may be deadlocked. Add debug logs to trace the flow.

7. Component Design: Consumer and Consumer Groups

> **Milestone(s):** 3 (Consumer Groups)

7.1 Responsibility and Scope

The **Consumer** is a client application that subscribes to topics and reads messages from partitions in a controlled, scalable manner. Its primary responsibility is to fetch batches of records from broker partitions, track its consumption progress via offsets, and participate in a **Consumer Group** for coordinated parallel consumption. The **Group Coordinator** (typically a designated broker) manages the metadata and lifecycle of consumer groups, including membership, partition assignment, and offset persistence.

Key Scope Boundaries:

- **Consumer Responsibilities:**
  - Discover topic partition leaders via metadata requests
  - Join/leave consumer groups and maintain heartbeat with the coordinator
  - Execute partition assignment strategies (when acting as group leader)
  - Fetch records from assigned partitions, respecting the high watermark
  - Periodically commit consumed offsets for durability and resume capability
  - Handle partition reassignment during group rebalancing
- **Coordinator Responsibilities:**
  - Maintain group membership state (active members, generation ID)
  - Trigger and orchestrate rebalancing when membership changes
  - Store and serve committed offsets for each group-partition pair
  - Detect and evict dead consumers via heartbeat timeout
  - Act as the single source of truth for group assignment
- **Out of Scope for This Implementation:**
  - Dynamic partition addition to existing topics (would trigger rebalance)
  - Transactional offset commits (exactly-once semantics)
  - Custom assignment strategies beyond range and round-robin
  - Automatic offset reset policies (earliest/latest) — defaults to committed offset
  - Standalone consumers (non-group) reading from explicit partitions

7.2 Mental Model: The Team Reading a Shared Book

Imagine a team assigned to read and summarize a multi-volume encyclopedia (the **topic**). Each volume is a **partition** — an independent, ordered sequence of pages. The team's goal is to divide the work evenly and ensure every page is read exactly once.

The Initial Division: When the team first gathers, they elect a **leader** who inspects all volumes and assigns each volume to exactly one team member. This is the **partition assignment**. Each member notes their starting page number (**offset**) in their assigned volumes.

Parallel Reading with Coordination: Each member reads pages sequentially from their assigned volumes, periodically checking in with the team coordinator to confirm they're still active (**heartbeat**). If a member leaves (goes for coffee) or joins the team, the coordinator calls everyone back to re-divide the volumes — a **rebalance**. During rebalance, everyone stops reading, waits for new assignments, then resumes from their last noted page.

Progress Tracking: Each member writes down the page number they've finished reading in a shared notebook (**offset commit**). If a member restarts (takes a nap), they can consult the notebook to resume from where they left off, avoiding re-reading or skipping pages.

This model illustrates the core consumer group concepts: **shared workload** (partitions assigned to one consumer), **coordination overhead** (rebalances pause consumption), **fault tolerance** (heartbeats detect failures), and **progress persistence** (offset commits enable resume).

7.3 Public Interface

The consumer and coordinator expose APIs for applications and internal communication. These interfaces are expressed as RPC methods the broker implements and the consumer client calls.

Consumer Client API (Application-Facing)

The consumer provides a pull-based interface where the application explicitly requests records.

Method	Parameters	Returns	Description
`Subscribe(topics []string)`	`topics`: List of topic names to subscribe to	`error`	Joins the consumer group and begins partition assignment for the given topics. Triggers initial rebalance.
`Poll(timeout time.Duration)`	`timeout`: Maximum time to wait for records	`[]*Record` (and internal offset update)	Fetches records from assigned partitions. Blocks until records arrive or timeout expires. Automatically handles heartbeats in background.
`CommitSync(offsets map[TopicPartition]int64)`	`offsets`: Map from partition to offset to commit	`error`	Synchronously commits specific offsets for the consumer's group. Waits for coordinator acknowledgment.
`CommitAsync(offsets map[TopicPartition]int64, callback func(error))`	`offsets`: Map to commit, `callback`: Invoked on completion	-	Asynchronously commits offsets. Callback receives error if commit fails.
`Close()`	-	`error`	Leaves consumer group, commits final offsets, and releases network resources.

Configuration Knobs:

- `group.id`: Unique string identifying the consumer group
- `session.timeout.ms`: Timeout after which coordinator considers consumer dead (triggers rebalance)
- `heartbeat.interval.ms`: Frequency of heartbeat signals to coordinator
- `max.poll.interval.ms`: Maximum time between `Poll()` calls before consumer considered stalled
- `auto.offset.reset`: What to do when no committed offset exists (`earliest`, `latest`, `none` – error)
- `enable.auto.commit`: Whether to automatically commit offsets periodically (simplified: we implement manual commit)

Coordinator Internal API (Broker-to-Consumer)

The coordinator implements these RPC endpoints that consumers call for group management.

Method	Request Parameters	Response Fields	Description
`JoinGroup(groupID, memberID, protocolType, protocols)`	`groupID`: Group identifier `memberID`: Current member ID (empty on first join) `protocolType`: Always "consumer" `protocols`: List of supported assignment strategies	`memberID`: Assigned member ID `generationID`: Current group generation `leaderID`: ID of elected leader `protocol`: Chosen assignment strategy `members`: List of group members (only returned to leader)	Registers consumer with group. If first member or rebalance needed, triggers new generation. Returns member list to leader for assignment.
`SyncGroup(groupID, generationID, memberID, assignments)`	`groupID`, `generationID`, `memberID`: Identity `assignments`: Partition assignments (from leader only)	`assignments`: Partition assignments for this member	After JoinGroup, members call this to receive their assigned partitions. Leader provides assignments for all members.
`Heartbeat(groupID, generationID, memberID)`	`groupID`, `generationID`, `memberID`: Identity	`error`	Periodic keep-alive. Coordinator resets session timeout. If generation mismatch or member unknown, signals rejoin.
`OffsetCommit(groupID, offsets)`	`groupID`: Group identifier `offsets`: Map from partition to offset + metadata	`error`	Persists offsets for partitions. Used for manual commit and auto-commit.
`OffsetFetch(groupID, partitions)`	`groupID`: Group identifier `partitions`: List of partitions to fetch offsets for	`offsets`: Map from partition to committed offset + metadata	Retrieves previously committed offsets. Called on consumer startup to resume.

7.4 Internal Behavior: Group Membership & Rebalancing

Consumer group operation is a distributed state machine coordinated by the group coordinator. Each consumer transitions through states, and the group as a whole undergoes phases.

Consumer Member State Machine

Each group member follows the state transitions below. Reference diagram:

Current State	Event	Next State	Actions Taken
UNJOINED	Application calls `Subscribe()`	**JOINING**	Generate temporary `memberID` (empty), send `JoinGroup` request to coordinator.
JOINING	Coordinator responds with `memberID` and `generationID`	**AWAITING_ASSIGNMENT**	Store assigned `memberID`. If elected leader, run partition assignment algorithm. Send `SyncGroup` with assignments (if leader) or empty assignments.
AWAITING_ASSIGNMENT	Coordinator responds to `SyncGroup` with partition assignments	**STABLE**	Update local assignment map. Start fetch loop for assigned partitions. Begin periodic heartbeats.
STABLE	`Poll()` called	**STABLE**	Fetch records from assigned partitions. Reset `max.poll.interval` timer.
STABLE	Heartbeat response indicates `REBALANCE_IN_PROGRESS`	**JOINING**	Stop fetching. Re-join group with current `memberID`.
STABLE	Session timeout (no heartbeat response)	**UNJOINED**	Assume coordinator dead. Re-discover coordinator and rejoin.
STABLE	Coordinator fails heartbeat (generation mismatch)	**JOINING**	Stop fetching. Re-join group with current `memberID`.
ANY	Application calls `Close()`	**UNJOINED**	Send `LeaveGroup` (optional), stop heartbeats, close network connections.

Group Rebalancing Protocol

Rebalancing ensures partitions are redistributed when members join or leave. The coordinator orchestrates this process:

1. **Trigger Detection:** Coordinator detects rebalance trigger:
   - New `JoinGroup` request for a group in `Stable` state
   - Existing member's heartbeat times out (`session.timeout.ms`)
   - Explicit member leave (TCP disconnect, `LeaveGroup` request)
2. **Generation Bump:** Coordinator increments `generationID` for the group, transitions group state to `PreparingRebalance`. It delays responding to new `JoinGroup` requests for up to `rebalance.timeout.ms` to allow existing members to rejoin.
3. **Member Joining:** Coordinator collects `JoinGroup` requests from all (existing and new) members. It chooses a leader (first member to join) and a partition assignment protocol (intersection of all members' supported protocols).
4. **Leader Assignment:** Coordinator responds to `JoinGroup` with member list **only to the leader**. Other members receive empty member list.

5. **Assignment Calculation:** Leader consumer runs partition assignment algorithm (range or round-robin) and sends assignments for all members in its `SyncGroup` request.
6. **Distribution:** Coordinator stores leader's assignments. When each member calls `SyncGroup`, it receives its specific assignment.
7. **Completion:** Once all members have called `SyncGroup`, coordinator transitions group to `Stable` state. Members receive assignments and begin fetching.

Critical Behavior: During rebalance (between steps 2-7), the coordinator responds to `Heartbeat` requests with `REBALANCE_IN_PROGRESS` error, prompting consumers to re-join. This ensures all members synchronize to the new generation.

Partition Assignment Algorithm (Leader Side)

When elected leader, the consumer runs one of the following algorithms to assign partitions to group members.

Range Assignment (Default):

1. Sort topics lexicographically, sort partitions within each topic numerically.
2. Sort consumer members lexicographically by `memberID`.
3. For each topic:
   - Calculate `partitionsPerConsumer = floor(totalPartitions / totalConsumers)`
   - Calculate `consumersWithExtra = totalPartitions % totalConsumers`
   - Iterate through sorted consumers: assign `partitionsPerConsumer + 1` partitions to the first `consumersWithExtra` consumers, then `partitionsPerConsumer` to the rest.
4. Output mapping: `memberID -> list of TopicPartition`.

Example: Topic `orders` with partitions 0-5, consumers C1, C2, C3.

- `partitionsPerConsumer = floor(6/3)=2`
- `consumersWithExtra = 6%3=0`
- Assignment: C1 gets [0,1], C2 gets [2,3], C3 gets [4,5].

RoundRobin Assignment:

1. Flatten all partitions across subscribed topics into a single list, sorted by topic then partition.
2. Sort consumer members lexicographically by `memberID`.
3. Distribute partitions in circular fashion: partition i goes to consumer at position `(i mod totalConsumers)`.
4. Output mapping.

Example: Same topic and consumers. Sorted partitions: [orders-0, orders-1, orders-2, orders-3, orders-4, orders-5].

- Assignment: C1 gets [orders-0, orders-3], C2 gets [orders-1, orders-4], C3 gets [orders-2, orders-5].

> **Design Insight:** Range assignment tends to preserve per-topic partition ordering and is simpler, but can lead to imbalance when subscriptions differ across members. RoundRobin distributes more evenly but scatters partitions of the same topic across consumers. For our educational implementation, we implement both but default to Range for its conceptual clarity.

Offset Commit and Fetch

Offset management is crucial for resume-on-restart semantics. The coordinator stores offsets in a durable store (simplified: in-memory map persisted to a file).

Commit Flow:

1. Consumer periodically (or manually) calls `OffsetCommit` RPC with a map of `TopicPartition -> offset`.
2. Coordinator validates the consumer is still a member of the group and generation matches.
3. Coordinator persists offset to storage (fsync for durability).

4. Coordinator responds success; consumer may discard locally cached commits.

Fetch Flow (on startup):

1. Consumer calls `OffsetFetch` for all partitions it is assigned (or potentially subscribes to).
2. Coordinator returns stored offset for each partition, or `-1` if none exists.
3. Consumer uses returned offset as its starting position. If offset is `-1`, it uses `auto.offset.reset` policy (default to earliest).

Storage Format: Offsets are stored per `(groupID, topic, partition)` tuple. Each entry includes offset, metadata (optional string), and timestamp.

7.5 ADR: Partition Assignment Strategy

> **Decision: Implement Both Range and RoundRobin, Default to Range**

- **Context:** Consumer groups need a deterministic algorithm to map partitions to members. The algorithm must produce balanced assignments while being understandable for learners. Real Kafka supports pluggable strategies; we must choose which to implement first.
- **Options Considered:**
  1. **Range Assignment:** Assign contiguous ranges of partitions per topic to each consumer.
  2. **RoundRobin Assignment:** Distribute partitions in circular order across all consumers.
  3. **Sticky Assignment:** Advanced algorithm that minimizes partition movement during rebalance (out of scope).
- **Decision:** Implement both Range and RoundRobin, with Range as the default. The coordinator will choose the protocol based on what all members support (intersection).
- **Rationale:**
  - **Range** is conceptually simpler to understand and implement — it mirrors how humans naturally divide ordered lists. This aligns with our educational goal.
  - **RoundRobin** introduces the concept of cross-topic balancing and is slightly more complex, but provides a clear contrast in behavior.
  - Implementing both allows learners to experiment with trade-offs and understand protocol negotiation.
  - Sticky assignment is omitted because its optimization (minimal reassignment) adds complexity disproportionate to learning value for Milestone 3.
- **Consequences:**
  - Learners must implement protocol negotiation in `JoinGroup`.
  - Range may cause workload imbalance when members subscribe to different topic subsets (but our simplified implementation assumes uniform subscription).
  - The system can be extended later with custom assignment strategies.

Comparison Table:

Option	Pros	Cons	Suitable For
Range	Simple to implement and reason about; preserves partition ordering per consumer	Can create imbalance when partitions per topic not divisible by consumers; uneven if subscriptions differ	Educational default; good for understanding basic assignment
RoundRobin	More even distribution across all partitions; balances load better	Scatters partitions of same topic across consumers; may hurt locality	Demonstrating alternative strategy; better balance
Sticky	Minimizes partition movement during rebalance; reduces temporary unavailability	Complex algorithm; stateful coordination required	Production systems where rebalance cost is high (out of scope)

7.6 Common Pitfalls

⚠️ Pitfall: Rebalance Storms (Frequent Unnecessary Rebalances)
Description: Consumers repeatedly trigger rebalances due to aggressive timeout settings or misconfigured heartbeats, causing constant consumption pauses.
Why Wrong: System spends more time rebalancing than processing messages; throughput plummets.
Fix: Set `session.timeout.ms` appropriately (e.g., 10-30 seconds). Ensure heartbeat thread runs independently of poll loop. Implement `max.poll.interval.ms` separately for slow processing detection.

⚠️ Pitfall: Zombie Consumers (Double Assignment)
Description: A consumer thought dead (heartbeat timeout) but actually just slow, continues fetching from its assigned partitions while new consumer gets same assignment.
Why Wrong: Two consumers read same partitions → duplicate processing and offset commit conflicts.
Fix: Coordinator must increment `generationID` on rebalance; include generation in fetch requests; broker should reject fetches from stale generations.

⚠️ Pitfall: Offset Commit Races During Rebalance
Description: Consumer commits offsets after being assigned partitions in rebalance, but commit may be applied after new generation is active, corrupting offset for new member.
Why Wrong: New consumer starts from wrong offset, causing missed or repeated messages.
Fix: Coordinator must reject offset commits with stale `generationID`. Consumer should commit offsets *before* joining new rebalance (implement `onPartitionsRevoked` callback in real Kafka).

⚠️ Pitfall: Missing Heartbeats Due to Blocking Poll
Description: Consumer's `Poll()` blocks longer than `session.timeout.ms` (e.g., processing large batch), preventing heartbeat thread from sending keep-alives.
Why Wrong: Coordinator evicts consumer, triggering rebalance even though consumer is active.
Fix: Run heartbeat thread independently with its own timer. Ensure `Poll()` returns quickly; process messages in application thread after fetch.

⚠️ Pitfall: Incorrect Assignment on Heterogeneous Subscriptions
Description: If group members subscribe to different topic lists, naive assignment algorithms may assign partitions for topics a consumer didn't subscribe to.
Why Wrong: Consumer receives partitions it cannot consume; some partitions may be unassigned.
Fix: In `JoinGroup`, each member must list its subscribed topics. Leader should assign only partitions of topics a member subscribes to. Filter assignments before distribution.

7.7 Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Group State Storage	In-memory map with periodic file snapshot (JSON)	Embedded KV store (BadgerDB) with WAL
Network Protocol	Plain TCP with custom binary format (same as produce/fetch)	gRPC with Protocol Buffers for type safety
Heartbeat Scheduler	Goroutine with `time.Ticker`	Dedcheduled timer wheel for efficiency
Offset Storage	Separate file per group (`offsets/groupID.json`)	Internal `__consumer_offsets` topic (like Kafka)

B. Recommended File/Module Structure

```
project-root/
  cmd/
    server/                # Broker main
      main.go
    consumer/              # Consumer client main
      main.go
  internal/
    coordinator/           # Group coordinator logic
      coordinator.go       # Group membership, rebalance orchestration
      assignment.go        # Range and RoundRobin assignment strategies
      offsets.go           # Offset commit/fetch storage
      coordinator_test.go
    consumer/              # Consumer client library
      client.go            # Public Consumer type (Subscribe, Poll, Commit)
      fetcher.go           # Background fetch loop for assigned partitions
      heartbeater.go       # Heartbeat management
      member.go            # Group membership state machine
    protocol/              # Shared wire format
      join_group.go        # JoinGroup request/response structs
      sync_group.go        # SyncGroup request/response
      heartbeat.go         # Heartbeat request/response
      offset_commit.go     # Offset commit/fetch structs
    storage/
      offset_store.go      # Offset persistence abstraction
```

C. Infrastructure Starter Code: Offset Storage

GO

```go
// internal/coordinator/offset_store.go

package coordinator


import (

    "encoding/json"

    "os"

    "path/filepath"

    "sync"

)


// OffsetStore persists consumer group offsets to disk.

// Simplified: one JSON file per group. Not optimized for performance.

type OffsetStore struct {

    baseDir string

    mu      sync.RWMutex

    // in-memory cache: group -> topic -> partition -> offset

    cache   map[string]map[string]map[int32]int64

}


// NewOffsetStore creates or loads offset store from directory.

func NewOffsetStore(baseDir string) (*OffsetStore, error) {

    if err := os.MkdirAll(baseDir, 0755); err != nil {

        return nil, err

    }

    store := &OffsetStore{

        baseDir: baseDir,

        cache:   make(map[string]map[string]map[int32]int64),

    }

    // Load existing offsets from disk (optional)

    // store.loadAll()

    return store, nil

}


// Commit writes offsets for a group.

func (s *OffsetStore) Commit(groupID string, topic string, partition int32, offset int64) error {
```

```
    s.mu.Lock()
    defer s.mu.Unlock()

    if _, ok := s.cache[groupID]; !ok {
        s.cache[groupID] = make(map[string]map[int32]int64)
    }
    if _, ok := s.cache[groupID][topic]; !ok {
        s.cache[groupID][topic] = make(map[int32]int64)
    }
    s.cache[groupID][topic][partition] = offset

    // Persist to disk (simplified: write entire group's offsets)
    return s.persistGroup(groupID)
}

// Fetch retrieves offset for a group-topic-partition.
func (s *OffsetStore) Fetch(groupID, topic string, partition int32) (int64, bool) {
    s.mu.RLock()
    defer s.mu.RUnlock()

    if groupMap, ok := s.cache[groupID]; ok {
        if topicMap, ok := groupMap[topic]; ok {
            offset, ok := topicMap[partition]
            return offset, ok
        }
    }
    return -1, false
}

// persistGroup writes group's offsets to JSON file.
func (s *OffsetStore) persistGroup(groupID string) error {
    groupFile := filepath.Join(s.baseDir, groupID+".json")
    s.mu.RLock()
    data, err := json.MarshalIndent(s.cache[groupID], "", "  ")
```

```
    s.mu.RUnlock()
    if err != nil {
        return err
    }
    // Atomic write: write to temp then rename
    tmpFile := groupFile + ".tmp"
    if err := os.WriteFile(tmpFile, data, 0644); err != nil {
        return err
    }
    return os.Rename(tmpFile, groupFile)
}

// LoadGroup reads offsets from disk (call on startup).
func (s *OffsetStore) LoadGroup(groupID string) error {
    groupFile := filepath.Join(s.baseDir, groupID+".json")
    data, err := os.ReadFile(groupFile)
    if os.IsNotExist(err) {
        return nil // No existing offsets
    }
    if err != nil {
        return err
    }
    var groupMap map[string]map[int32]int64
    if err := json.Unmarshal(data, &groupMap); err != nil {
        return err
    }
    s.mu.Lock()
    s.cache[groupID] = groupMap
    s.mu.Unlock()
    return nil
}
```

D. Core Logic Skeleton Code

Group Coordinator Main Logic:

```go
// internal/coordinator/coordinator.go

package coordinator

import (
    "sync"
    "time"
)

type GroupState int32

const (
    GroupStateStable GroupState = iota
    GroupStatePreparingRebalance
    GroupStateDead
)

type ConsumerGroup struct {
    GroupID      string
    State        GroupState
    GenerationID int32
    LeaderID     string
    Protocol     string // e.g., "range", "roundrobin"
    Members      map[string]*ConsumerMetadata // keyed by memberID
    TimeoutMs    int32
    mu           sync.RWMutex
    // Tracks when rebalance started for timeout
    rebalanceStartedAt time.Time
}

type ConsumerMetadata struct {
    MemberID           string
    ClientID           string
    SubscribedTopics   []string
    AssignedPartitions map[string][]int32 // topic -> partition list
    LastHeartbeat      time.Time
    JoinWait           chan struct{} // signals member that join is complete
```

```go
}

type Coordinator struct {
    groups   map[string]*ConsumerGroup
    offsetStore *OffsetStore
    mu       sync.RWMutex
}

// HandleJoinGroup processes a consumer's request to join a group.
// Returns memberID, generationID, leaderID, protocol, and member list (if leader).
func (c *Coordinator) HandleJoinGroup(groupID, memberID, protocolType string,
    protocols []string, timeoutMs int32) (string, int32, string, string, []*ConsumerMetadata, error) {
    // TODO 1: Lookup or create group for groupID
    // TODO 2: If memberID is empty, generate a unique ID (e.g., "consumer-<uuid>")
    // TODO 3: Validate protocolType == "consumer"
    // TODO 4: If group is Stable and this is first member joining (new consumer), trigger rebalance:
    //   - Set group state to PreparingRebalance
    //   - Increment GenerationID
    //   - Record rebalance start time
    // TODO 5: If group is already PreparingRebalance, add member to Members map
    // TODO 6: If this member is first to join in this generation, set as LeaderID
    // TODO 7: Choose protocol from intersection of all members' protocols (prefer "range")
    // TODO 8: If all expected members have joined or timeout reached, move to next step:
    //   - For leader: return full member list so it can compute assignments
    //   - For followers: return empty member list
    // TODO 9: Return assigned memberID, current GenerationID, LeaderID, chosen Protocol
    return "", 0, "", "", nil, nil
}

// HandleSyncGroup distributes partition assignments to members.
func (c *Coordinator) HandleSyncGroup(groupID string, generationID int32,
    memberID string, assignments map[string]*Assignment) (map[string][]int32, error) {
    // TODO 1: Validate group exists, generation matches, member is in group
    // TODO 2: If member is leader, store assignments for all members
    // TODO 3: If assignments already stored (by leader), return this member's assignment
```

```
    // TODO 4: If all members have retrieved assignments, transition group to Stable state
    // TODO 5: Return assigned partitions for this member: topic -> []partition
    return nil, nil
}

// HandleHeartbeat validates member liveness.
func (c *Coordinator) HandleHeartbeat(groupID string, generationID int32,
    memberID string) error {
    // TODO 1: Lookup group, validate generation matches
    // TODO 2: Update member's LastHeartbeat timestamp
    // TODO 3: If group is PreparingRebalance, return error to signal consumer to rejoin
    // TODO 4: Return nil if successful
    return nil
}

// checkRebalanceTimeout runs periodically to complete rebalance if members don't all join.
func (c *Coordinator) checkRebalanceTimeout() {
    // TODO 1: Iterate groups in PreparingRebalance state
    // TODO 2: If rebalance started more than TimeoutMs ago, proceed with current members
    // TODO 3: Mark missing members as dead, remove from group
    // TODO 4: Trigger assignment with remaining members
}
```

Consumer Client Membership State Machine:

```go
// internal/consumer/member.go

package consumer


import (

    "context"

    "time"

)


type MemberState int


const (

    StateUnjoined MemberState = iota

    StateJoining

    StateAwaitingAssignment

    StateStable

)


type GroupMember struct {

    groupID        string

    memberID       string

    generationID   int32

    state          MemberState

    assigned       map[TopicPartition]bool

    coordinator    *BrokerConnection // connection to group coordinator

    heartbeatInterval time.Duration

    lastHeartbeat   time.Time

    cancelHeartbeat context.CancelFunc

    mu             sync.RWMutex

}


// joinGroup performs the JoinGroup/SyncGroup protocol.

func (m *GroupMember) joinGroup(topics []string) error {

    // TODO 1: Set state to Joining

    // TODO 2: Send JoinGroup request to coordinator with empty memberID (if first time)

    // TODO 3: On response, store assigned memberID, generationID

    // TODO 4: If elected leader, compute assignments for all members using chosen protocol
```

```
    // TODO 5: Send SyncGroup request with assignments (if leader) or empty map
    // TODO 6: On SyncGroup response, update assigned partitions map
    // TODO 7: Start heartbeat goroutine
    // TODO 8: Transition state to Stable
    // TODO 9: Start fetcher for assigned partitions
    return nil
}

// startHeartbeat begins periodic heartbeat to coordinator.
func (m *GroupMember) startHeartbeat(ctx context.Context) {
    ticker := time.NewTicker(m.heartbeatInterval)
    go func() {
        defer ticker.Stop()
        for {
            select {
            case <-ticker.C:
                // TODO 1: Send Heartbeat request
                // TODO 2: If error indicates rebalance needed, call joinGroup again
                // TODO 3: Update lastHeartbeat timestamp
            case <-ctx.Done():
                return
            }
        }
    }()
}
```

Partition Assignment Strategies:

```go
// internal/coordinator/assignment.go

package coordinator


// Assignment represents partitions assigned to a consumer member.

type Assignment struct {

    TopicPartitions map[string][]int32 // topic -> list of partition IDs

}


// RangeAssigner implements range assignment strategy.

type RangeAssigner struct{}


func (r *RangeAssigner) Assign(members []*ConsumerMetadata,

    topics map[string][]int32) map[string]*Assignment {

    // TODO 1: Sort members by MemberID

    // TODO 2: For each topic, sort partitions numerically

    // TODO 3: Calculate partitionsPerConsumer and consumersWithExtra

    // TODO 4: Assign contiguous ranges to each consumer

    // TODO 5: Build Assignment map for each member

    return nil

}


// RoundRobinAssigner implements round-robin assignment.

type RoundRobinAssigner struct{}


func (rr *RoundRobinAssigner) Assign(members []*ConsumerMetadata,

    topics map[string][]int32) map[string]*Assignment {

    // TODO 1: Sort members by MemberID

    // TODO 2: Flatten all partitions into single list: [topic-partition, ...]

    // TODO 3: Distribute in round-robin fashion: partition i -> member i % len(members)

    // TODO 4: Build Assignment map for each member

    return nil

}
```

E. Language-Specific Hints (Go)

1. **Concurrency:** Use `sync.RWMutex` for `ConsumerGroup` and `Coordinator` maps. Heartbeat and fetch loops should run in separate goroutines; coordinate shutdown with `context.Context`.

2. **ID Generation:** Use `crypto/rand` for unique `memberID` (e.g., `"consumer-" + hex.EncodeToString(uuid[:])`). Avoid using client IP/port as it may change.
3. **Timers:** For session timeout, store `LastHeartbeat` as `time.Time` and periodically check `time.Since(last) > sessionTimeout`. Use `time.Ticker` for heartbeat loop.
4. **Network Errors:** When TCP connection to coordinator fails, consumer should rediscover coordinator via metadata request and rejoin group.
5. **State Persistence:** Write offset commits with `os.WriteFile` to temporary file then `os.Rename` for atomicity. Use JSON for simplicity; in production you'd use binary format.

F. Milestone Checkpoint

After implementing consumer groups:

1. **Start a broker:** `go run cmd/server/main.go --port 9092`
2. **Create a topic with 3 partitions:** Use your admin tool or implement a simple `CreateTopic` RPC.
3. **Start two consumers in same group:**

```bash
# Terminal 1
go run cmd/consumer/main.go --group my-group --topic test-topic


# Terminal 2
go run cmd/consumer/main.go --group my-group --topic test-topic
```

4. **Start a producer and send 100 messages:** Messages should be divided between the two consumers.
5. **Kill one consumer (Ctrl+C):** Observe logs: surviving consumer should get reassigned all partitions after rebalance timeout.
6. **Restart killed consumer:** It should rejoin and partitions should be redistributed.
7. **Verify no duplicates:** Count messages received across all consumers; should equal 100 exactly.
8. **Test offset persistence:** Stop all consumers, restart, produce more messages. New consumers should resume from committed offsets.

Expected Observations:

- Consumers log "Assigned partitions: [...]" on each rebalance.
- During rebalance, message consumption pauses briefly.
- If you implement auto-commit, offsets are saved periodically.
- Heartbeat logs (if enabled) show periodic "Heartbeat sent" messages.

Signs of Trouble:

- **Messages duplicated:** Likely generation ID not validated in fetch requests, or offset commits race.
- **Consumer gets no messages:** Check assignment algorithm; verify partitions actually have data.
- **Rebalance loops continuously:** Session timeout too short or heartbeat not sent.
- **Offsets not persisted:** Offset store file not being written (check permissions).

---

8. Component Design: Replication

> **Milestone(s):** 4 (Replication)

8.1 Responsibility and Scope

The **Replication module** is the core subsystem responsible for providing **fault tolerance** and **data durability** in our distributed message queue. Its primary responsibility is to maintain **identical copies** (replicas) of each partition's log across multiple broker nodes, ensuring that if one broker fails, another can seamlessly take over without data loss or unavailability. This module operates within the **replication plane** of the cluster—the internal network path over which brokers communicate to synchronize data.

The scope of the replication module encompasses four critical functions:

1. **Leader-Follower Replication Protocol:** Continuously copying newly appended records from the **leader replica** (the broker responsible for handling all read/write requests for a partition) to one or more **follower replicas**.
2. **In-Sync Replica (ISR) Management:** Dynamically tracking which follower replicas are sufficiently caught up with the leader to be considered "in-sync," and managing the membership of this set based on configurable lag thresholds.
3. **High Watermark Advancement:** Determining the offset up to which all in-sync replicas have replicated data, thereby defining the **durable frontier** that consumers can safely read without risking exposure to data that could be lost during a failover.
4. **Leader Election:** Selecting a new leader replica from the current ISR when the existing leader becomes unavailable, ensuring continuous availability of the partition.

This module directly interacts with the `Log` and `Partition` components within each broker, and communicates with other brokers via internal RPCs. It is a foundational requirement for achieving the **exactly-once semantics** and **guaranteed delivery** that characterize production-grade message systems.

8.2 Mental Model: The Ship's Log Replica

To build intuition, imagine a **captain's logbook** on a sailing ship (the partition log). The captain (the **leader**) records every important event (messages) in chronological order. To protect against the logbook being lost overboard, the captain employs several **first mates** (followers) who each maintain their own copy.

- **Copying Process:** After each entry, the captain calls out the new lines, and the first mates write them into their own logbooks. Some mates write quickly and stay current; others might lag behind if distracted.
- **In-Sync Crew (ISR):** The captain periodically checks which mates have fully copied the latest entries. Only those who are completely up-to-date are considered part of the "in-sync crew." If a mate falls too far behind (beyond a tolerance threshold), they are temporarily removed from this trusted set until they catch up.
- **Safe Reading Point (High Watermark):** When a sailor (consumer) wants to read the log, the captain only allows them to read entries that **every in-sync mate** has successfully copied. This ensures that even if the captain is suddenly lost at sea, the new captain (elected from the in-sync crew) will have a complete copy of everything the sailor has read, guaranteeing no data loss.
- **Captain Election:** If the captain falls ill, the crew holds a quick vote among the in-sync mates to choose a new captain. They select the mate whose log is most complete (has the highest offset). This new captain immediately assumes responsibility for recording new events.

This model illustrates the core trade-off: **durability versus latency**. Waiting for all mates to acknowledge each entry (synchronous replication) maximizes safety but slows down the recording process. The ISR model allows the captain to proceed once a **quorum** of mates (often just the leader itself, or a majority) has acknowledged, striking a practical balance.

8.3 Internal Broker-to-Broker API

Replication is driven by a set of internal Remote Procedure Calls (RPCs) between brokers. These are separate from the client-facing `Produce` and `Fetch` APIs, though they may share the same transport layer. The following table details the key RPCs.

Method Name	Initiator	Target	Parameters	Returns	Description
`LeaderForPartition`	Follower/Coordinator	Any Broker	`Topic string`, `PartitionID int`	`LeaderBrokerID int`, `LeaderEpoch int32`, `Error error`	Discovers the current leader broker for a given partition. Used by followers during startup or after a leader change.
`FetchReplica`	Follower	Leader	`Topic string`, `PartitionID int`, `FollowerBrokerID int`, `FetchOffset int64`, `MaxBytes int32`	`Records []*Record`, `HighWatermark int64`, `LeaderEpoch int32`, `Error error`	The core replication fetch request. A follower uses this to pull records from the leader's log starting at `FetchOffset`. The leader includes the current **high watermark** so the follower knows what is safely committed.
`UpdateISR`	Leader	Metadata Coordinator / Controller	`Topic string`, `PartitionID int`, `NewISR []int`, `LeaderEpoch int32`	`Error error`	Notifies the cluster metadata service of changes to the In-Sync Replica set (e.g., a follower is added or removed). This update must be persisted and propagated to all brokers.
`BeginLeaderElection`	Controller / Coordinator	Candidate Broker	`Topic string`, `PartitionID int`, `CandidateISR []int`	`Success bool`, `Error error`	Initiates a leader election for a partition. The controller proposes a candidate broker (from the ISR) to become the new leader. The candidate validates it is in the ISR and takes leadership.

Method Name	Initiator	Target	Parameters	Returns	Description
`FollowerHeartbeat`	Follower	Leader	`Topic string`, `PartitionID int`, `FollowerBrokerID int`, `FollowerLEO int64`	`CurrentLeaderEpoch int32`, `Error error`	Optional periodic heartbeat from follower to leader, carrying the follower's latest Log End Offset (LEO). This allows the leader to track follower lag without waiting for a fetch request.

> **Design Insight:** The `FetchReplica` RPC is intentionally similar to the client `Fetch` API. This symmetry simplifies the code—a follower is essentially a special consumer that reads from the leader's log. However, replication fetches are continuous, long-polling requests that wait for new data, unlike typical consumer fetches.

In our simplified educational system, we may combine the roles of **Metadata Coordinator** and **Controller** into a single component (the `Coordinator` from previous sections). Therefore, `UpdateISR` and `BeginLeaderElection` would be directed to that coordinator broker.

8.4 Internal Behavior: Follower Sync and ISR Management

The replication process is a continuous loop run by each follower replica for every partition it is assigned to. The leader, in parallel, monitors the progress of all followers to maintain the ISR.

Follower Synchronization Algorithm

For each partition where the broker is a follower:

1. **Determine Leader:** If not known, call `LeaderForPartition` to discover the current leader broker.
2. **Establish Connection:** Open a persistent network connection to the leader's replication endpoint.
3. **Initialize Fetch Offset:** Start fetching from the last offset successfully written to the follower's local log (its **Log End Offset** or **LEO**). If the log is empty, start at 0.
4. **Loop:**
    1. Send a `FetchReplica` request to the leader with the current `FetchOffset`.
    2. If the leader responds with records:
        1. Append the records to the local `Log` in order, ensuring no gaps in offsets.
        2. Advance the local LEO to `FetchOffset + len(records)`.
        3. Optionally, periodically send a `FollowerHeartbeat` with the new LEO to keep the leader informed.
    3. If the leader responds with an error (e.g., `ErrNotLeaderForPartition`):
        1. Break the connection.
        2. Wait a short, randomized backoff period (to avoid thundering herds).
        3. Go to step 1 to rediscover the new leader.
    4. If the leader responds with no new records (i.e., the follower is caught up):
        1. Wait for a configurable `replica.fetch.wait.ms` interval, then repeat the fetch (long polling). This reduces busy-waiting.

Leader ISR Management Algorithm

For each partition where the broker is the leader:

1. **Initialize ISR:** The ISR starts as the set of all assigned replicas (including the leader). This is stored in the `Partition.ISR` field.
2. **Track Follower State:** Maintain, for each follower in the ISR, the last known **last fetched offset** (from `FetchReplica` requests) or **last heartbeat LEO**.
3. **Periodic Check (every `replica.lag.time.max.ms`):**
    1. Calculate the **replica lag** for each follower: `Leader's LEO - Follower's Last Fetched Offset`.
    2. If a follower's lag exceeds `replica.lag.max.messages` **or** its last fetch time is older than `replica.lag.time.max.ms`, remove it from the ISR.
    3. If a previously out-of-sync follower's lag falls to zero (it has caught up), add it back to the ISR.
4. **On ISR Change:**
    1. Persist the new ISR set (e.g., to ZooKeeper or an internal topic).
    2. Broadcast the update via `UpdateISR` to the metadata coordinator so other brokers can update their caches.
5. **Advance High Watermark:**
    1. The **high watermark** (`Partition.HighWatermark`) is the maximum offset for which **all replicas in the current ISR** have acknowledged replication.
    2. After each successful `FetchReplica` response acknowledgment from a follower, the leader recalculates the high watermark as the minimum LEO across all ISR members.
    3. The high watermark is included in every `FetchReplica` response, allowing followers (and consumers) to know what is safely durable.

> **Critical Detail:** The high watermark advancement is **monotonic**. It never moves backward, even if the ISR shrinks. If a follower is removed from the ISR because it is slow, the high watermark can still advance based on the remaining ISR members. This ensures availability at the potential cost of durability if the remaining ISR size falls below a desired minimum (a pitfall we address later).

8.5 ADR: Leader Election Trigger

> **Decision: Coordinator-Initiated Leader Election**

- **Context:** When a partition leader fails (due to broker crash, network partition, or graceful shutdown), a new leader must be elected promptly to maintain partition availability. The election must choose a replica that is **guaranteed to have all committed messages** (i.e., a member of the ISR) to prevent data loss. We need a simple, deterministic mechanism that learners can implement without complex consensus protocols.
- **Options Considered:**
    1. **Explicit Coordinator-Managed Election:** A dedicated controller/coordinator broker monitors leader liveness (via heartbeats). Upon detecting leader failure, it selects a new leader from the ISR and directs it to assume leadership via RPC.
    2. **Follower-Triggered Election:** Followers detect the leader is unresponsive (via fetch timeouts) and initiate a leader election among themselves using a consensus protocol like Raft or a simple voting mechanism.
- **Decision:** We choose **Option 1: Explicit Coordinator-Managed Election**. The `Coordinator` component (introduced for consumer groups) will be extended to act as a **controller** for partition leadership.
- **Rationale:**
    - **Simplicity for Learners:** Implementing a full distributed consensus protocol (like Raft) is a significant undertaking beyond the core learning goals of message queue replication. Coordinator-managed election centralizes the decision logic, making it easier to implement, debug, and understand.

    - **Alignment with Kafka's Design:** Apache Kafka uses a similar controller-based leader election, proving the pattern's effectiveness at scale. This allows learners to map concepts directly to the real system.
    - **Deterministic Outcome:** The coordinator has a global view of the ISR and can make an unambiguous choice (e.g., pick the replica with the highest LEO). This avoids split-brain scenarios that can occur in leaderless voting if network partitions are not properly handled.
    - **Integration with Metadata:** The coordinator already manages cluster metadata (brokers, topics). Extending it to manage partition leadership state is a natural cohesion of responsibilities.
- **Consequences:**
    - **Single Point of Failure:** The coordinator becomes a critical component. If it crashes, leader elections cannot occur until a new coordinator is elected. We mitigate this by making the coordinator role **electable** among brokers (similar to Kafka's controller election), but this adds complexity.
    - **Increased Latency on Failover:** Detection and election involve two hops: follower detects leader failure, reports to coordinator, coordinator elects new leader. This may take slightly longer than a follower immediately taking over, but within acceptable bounds for our educational system.
    - **Simplified Follower Logic:** Followers only need to report leader failure and accept leadership assignments; they don't need election logic.

Option	Pros	Cons	Chosen?
Coordinator-Managed	Simple to implement; Deterministic; Centralized metadata consistency.	Coordinator is a SPOF; Slightly slower failover.	**Yes**
Follower-Triggered	Faster failover; No single point of failure.	Requires complex consensus protocol; Risk of split-brain; Harder to implement correctly.	No

The election algorithm, managed by the coordinator, proceeds as follows:

1. **Failure Detection:** The coordinator receives heartbeat or metadata from brokers. If the leader broker for a partition fails to heartbeat within `session.timeout.ms`, it is considered dead.
2. **ISR Validation:** The coordinator fetches the latest ISR for the partition from its persisted metadata.
3. **Leader Selection:** From the ISR, select the replica with the highest **Log End Offset** (the most up-to-date). If multiple have the same LEO, choose the first by broker ID for determinism.
4. **Leadership Transition:** Send a `BeginLeaderElection` RPC to the chosen broker. The broker validates it is in the ISR, updates its `Partition.LeaderBrokerID` to itself, and begins accepting produce/fetch requests.
5. **Metadata Propagation:** The coordinator updates the `PartitionMetadata` for all brokers and notifies them of the new leader.

8.6 Common Pitfalls

Implementing replication is fraught with subtle bugs that can lead to data loss or inconsistency. Here are the most common pitfalls learners encounter.

⚠️ Pitfall 1: Allowing Unclean Leader Election

- **Description:** Electing a leader that is **not** in the ISR (i.e., a lagging follower) because the ISR has become empty. This new leader may be missing messages that were acknowledged to producers, causing **data loss**.
- **Why It's Wrong:** The fundamental guarantee of durability is violated. Producers that received an acknowledgment (`acks=all`) believe their message is durable, but it is lost.
- **How to Fix:** Implement a strict policy: **never elect a non-ISR replica**. If the ISR becomes empty, the partition must become unavailable (return an error to clients) until a replica recovers and rejoins the ISR. This is the "unclean.leader.election.enable=false" policy in Kafka.

⚠️ Pitfall 2: ISR Shrinking to Zero

- **Description:** If followers are consistently slower than the leader (due to network or disk issues), they may be removed from the ISR one by one until no followers remain. This leaves the leader as the only ISR member, which is risky—if it fails, no eligible successor exists.
- **Why It's Wrong:** The system loses fault tolerance. While the partition remains available, it is one failure away from permanent unavailability.
- **How to Fix:** Monitor ISR size and alert if it falls below a minimum threshold (e.g., `min.insync.replicas`). Producers can be configured to require a minimum ISR size (`acks=all` will fail if ISR size is below this minimum), trading availability for safety. Also, tune replica fetch parameters to reduce lag.

⚠️ Pitfall 3: Incorrect High Watermark Update

- **Description:** Updating the high watermark based on **all replicas** (including out-of-sync ones) or updating it before a follower's write is durable on disk. This can cause the watermark to advance too slowly (hindering consumers) or, worse, to advance too quickly (exposing uncommitted data).
- **Why It's Wrong:** If the high watermark advances before data is durable on followers, and the leader crashes, a new leader may not have that data, yet consumers have already read it (data loss for consumers). Conversely, a stagnant watermark reduces consumer throughput.
- **How to Fix:** The high watermark must be the minimum LEO **only among current ISR members**. Additionally, a follower should only update its LEO (and thus be counted for the leader's watermark calculation) after it has durably appended the records to its local log (fsync). The leader should wait for this acknowledgment.

⚠️ Pitfall 4: Not Handling Leader Epoch

- **Description:** Failing to track a **leader epoch**—a monotonically increasing number for each leader change—can cause **message duplication** or **reordering** after a leader failover.
- **Why It's Wrong:** Without epoch tracking, a former leader that comes back online (e.g., after a network partition) may still think it's the leader and accept writes, creating a split-brain scenario with divergent logs.
- **How to Fix:** Introduce a `LeaderEpoch` field in `PartitionMetadata`. Each new leader increments the epoch. Include the epoch in every replication RPC and client request. Followers and clients reject requests from stale leaders (with older epochs). The leader also truncates its log to the last known offset for each epoch to maintain consistency.

⚠️ Pitfall 5: Busy-Waiting in Follower Fetch Loop

- **Description:** Followers that immediately re-send fetch requests when caught up, consuming excessive CPU and network resources.
- **Why It's Wrong:** Inefficient resource usage; can overwhelm the leader with requests.
- **How to Fix:** Implement **long polling**: the leader holds the fetch request for a short period (`replica.fetch.wait.ms`) if there is no new data, returning immediately when data arrives or the wait time elapses. This reduces the request rate while maintaining low latency.

8.7 Implementation Guidance

This section provides starter code and structure for the replication module in Go.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Replication Transport	Plain TCP with custom binary protocol (reuse `TCPServer`)	gRPC with streaming for fetch requests
Metadata Storage	In-memory map in the coordinator, persisted to a local WAL file	Embedded key-value store (BadgerDB) for fault-tolerant metadata
Leader Election	Centralized coordinator with manual failover	Raft consensus for coordinator fault tolerance
Follower Sync	Periodic fetch with sleep interval	Long-polling fetch with `net.Conn` read timeout

B. Recommended File/Module Structure

Add the following directories and files to your project. This separates replication logic from the core broker and coordinator.

```
project-root/
  internal/
    broker/
      server.go           # Main broker logic (existing)
      log_manager.go      # Log management (existing)
    replication/
      replica_manager.go  # Main replication logic on each broker
      follower_syncer.go  # Follower sync loop
      isr_manager.go      # ISR tracking and high watermark updates
      leader_election.go  # Leader election logic (in coordinator)
    coordinator/
      coordinator.go      # Extended with controller functions
      metadata_store.go   # Persists topic/partition metadata
    protocol/
      replication.go      # Replication RPC request/response structs
```

C. Infrastructure Starter Code: Replication RPC Protocol

First, define the replication-specific protocol messages. These are separate from client protocol messages.

```go
// internal/protocol/replication.go

package protocol


// FetchReplicaRequest is sent by follower to leader to fetch records.

type FetchReplicaRequest struct {

    Topic           string

    PartitionID     int32

    FollowerBrokerID int32

    FetchOffset     int64

    MaxBytes        int32

    // LeaderEpoch for fencing

    CurrentLeaderEpoch int32

}


// FetchReplicaResponse is returned by leader to follower.

type FetchReplicaResponse struct {

    ErrorCode       int16 // e.g., ErrNone, ErrNotLeaderForPartition

    HighWatermark   int64

    LeaderEpoch     int32

    Records         []*Record

}


// UpdateISRRequest is sent by leader to coordinator.

type UpdateISRRequest struct {

    Topic       string

    PartitionID int32

    NewISR      []int32

    LeaderEpoch int32

    // ZK version or similar for optimistic concurrency

    ZKVersion   int32

}


// UpdateISRResponse is returned by coordinator.

type UpdateISRResponse struct {

    ErrorCode int16
```

```
}

// LeaderForPartitionRequest asks for the current leader of a partition.
type LeaderForPartitionRequest struct {
    Topic       string
    PartitionID int32
}

// LeaderForPartitionResponse contains leader info.
type LeaderForPartitionResponse struct {
    ErrorCode       int16
    LeaderBrokerID  int32
    LeaderEpoch     int32
}

// BeginLeaderElectionRequest instructs a broker to become leader.
type BeginLeaderElectionRequest struct {
    Topic       string
    PartitionID int32
    // The ISR at time of election, for validation
    ISR         []int32
    LeaderEpoch int32 // The new epoch
}

// BeginLeaderElectionResponse confirms leadership taken.
type BeginLeaderElectionResponse struct {
    ErrorCode int16
}
```

D. Core Logic Skeleton Code

1. Replica Manager (runs on each broker)

```go
// internal/replication/replica_manager.go

package replication

import (
    "context"
    "sync"
    "time"
    "yourproject/internal/types"
)

// ReplicaManager manages all replica sync tasks for partitions hosted on this broker.
type ReplicaManager struct {
    brokerID      int32
    config        ReplicaConfig
    // Maps TopicPartition -> FollowerSyncer (for partitions where we are follower)
    followers     map[types.TopicPartition]*FollowerSyncer
    // Maps TopicPartition -> ISRManager (for partitions where we are leader)
    leaders       map[types.TopicPartition]*ISRManager
    mu            sync.RWMutex
    ctx           context.Context
    cancel        context.CancelFunc
}

type ReplicaConfig struct {
    FetchWaitMs          int
    ReplicaFetchMinBytes int32
    ReplicaFetchMaxBytes int32
    ReplicaLagTimeMaxMs  int
}

// NewReplicaManager creates a new manager.
func NewReplicaManager(brokerID int32, config ReplicaConfig) *ReplicaManager {
    ctx, cancel := context.WithCancel(context.Background())
    return &ReplicaManager{
        brokerID: brokerID,
```

```
        config:   config,
        followers: make(map[types.TopicPartition]*FollowerSyncer),
        leaders:   make(map[types.TopicPartition]*ISRManager),
        ctx:       ctx,
        cancel:    cancel,
    }
}

// Start begins all sync loops for followers and leaders.
func (rm *ReplicaManager) Start() error {
    // TODO 1: Load partition assignments from local metadata (which partitions this broker hosts)
    // TODO 2: For each partition where this broker is a follower, create a FollowerSyncer and start its run loop
    // TODO 3: For each partition where this broker is the leader, create an ISRManager and start its monitoring loop
    // TODO 4: Start a background goroutine to handle partition assignment changes (e.g., after rebalance)
    return nil
}

// OnNewPartitionAssignment is called when the broker's assigned partitions change.
func (rm *ReplicaManager) OnNewPartitionAssignment(assignments []types.PartitionMetadata) {
    // TODO 1: Compare new assignments with current followers/leaders maps
    // TODO 2: Stop and remove syncers for partitions we are no longer assigned
    // TODO 3: Add new syncers for newly assigned partitions (determine role: leader or follower)
    // TODO 4: For partitions where role changed (follower->leader or vice versa), recreate the syncer
}
```

2. Follower Syncer (per partition where broker is follower)

```go
// internal/replication/follower_syncer.go

package replication

import (
    "context"
    "time"
    "yourproject/internal/types"
)

// FollowerSyncer continuously fetches from the leader for a single partition.
type FollowerSyncer struct {
    brokerID      int32
    topic         string
    partitionID   int32
    config        ReplicaConfig
    log           types.Log // local log to append to
    // Connection pool to talk to other brokers
    connPool      *ConnectionPool
    leaderBrokerID int32
    leaderEpoch   int32
    fetchOffset   int64
    mu            sync.RWMutex
    ctx           context.Context
    cancel        context.CancelFunc
}

// Run is the main sync loop.
func (fs *FollowerSyncer) Run() {
    // TODO 1: Discover leader using LeaderForPartition RPC (retry until successful)
    // TODO 2: Loop until context is done:
    //   a. Send FetchReplicaRequest to leader with current fetchOffset
    //   b. On response:
    //        i. If error is ErrNotLeaderForPartition, break to rediscover leader
    //       ii. If no error, append records to local log (ensure offset continuity)
    //      iii. Advance fetchOffset by number of records appended
```

```
    //       iv. Update high watermark from response (store in partition metadata)
    //   c. If no records returned, sleep for config.FetchWaitMs before next fetch (long polling)
    //   d. If fetch failed (network error), backoff exponentially and retry
}

// AppendToLocalLog writes records to the local log filesystem.
func (fs *FollowerSyncer) AppendToLocalLog(records []*types.Record, leaderHW int64) error {
    // TODO 1: Acquire lock on log to ensure sequential writes
    // TODO 2: Validate that the first record's offset equals the current local LEO (no gaps)
    // TODO 3: Call log.Append(records)
    // TODO 4: Update in-memory high watermark for this partition (min(leaderHW, localLEO))
    // TODO 5: Return any error from append
    return nil
}
```

3. ISR Manager (per partition where broker is leader)

```go
// internal/replication/isr_manager.go

package replication


import (

    "context"

    "time"

    "yourproject/internal/types"

)


// ISRManager tracks followers and manages ISR for a partition where this broker is leader.

type ISRManager struct {

    topic         string

    partitionID   int32

    config        ReplicaConfig

    partition     *types.Partition // reference to the partition object

    // Map followerBrokerID -> lastCaughtUpTime and lastFetchOffset

    followerState map[int32]*followerStatus

    mu            sync.RWMutex

    ctx           context.Context

    cancel        context.CancelFunc

}


type followerStatus struct {

    lastFetchOffset int64

    lastFetchTime   time.Time

}


// Start begins the periodic ISR check.

func (im *ISRManager) Start() {

    ticker := time.NewTicker(time.Duration(im.config.ReplicaLagTimeMaxMs) * time.Millisecond)

    defer ticker.Stop()

    for {

        select {

        case <-im.ctx.Done():

            return
```

```
        case <-ticker.C:
            im.evaluateISR()
        }
    }
}

// evaluateISR checks each follower's lag and updates ISR.
func (im *ISRManager) evaluateISR() {
    // TODO 1: Get current leader LEO from partition.Log
    // TODO 2: For each follower in partition.ReplicaBrokerIDs:
    //   a. Compute lag = leaderLEO - followerState[brokerID].lastFetchOffset
    //   b. Compute time since last fetch
    //   c. If lag > replica.lag.max.messages OR time since last fetch > replica.lag.time.max.ms:
    //        Remove follower from ISR (if present)
    //   d. Else if follower is not in ISR and lag == 0:
    //        Add follower back to ISR
    // TODO 3: If ISR changed:
    //   a. Update partition.ISR
    //   b. Persist new ISR via UpdateISR RPC to coordinator
    //   c. Recalculate high watermark (min LEO across new ISR)
}

// UpdateFollowerState is called when a FetchReplica request is processed.
func (im *ISRManager) UpdateFollowerState(followerID int32, fetchOffset int64, fetchedBytes int) {
    // TODO 1: Update followerState map with new offset and current time
    // TODO 2: If fetchOffset == leaderLEO (follower is caught up), ensure follower is in ISR
}
```

4. Leader Election in Coordinator

```go
// internal/coordinator/leader_election.go

package coordinator

import (
    "context"
    "sort"
    "time"
    "yourproject/internal/types"
)

// LeaderElector runs in the coordinator and manages partition leader elections.
type LeaderElector struct {
    metadataStore *MetadataStore
    brokerPool    *BrokerPool // to send RPCs
    ctx           context.Context
}

// OnBrokerFailure triggers leader election for partitions whose leader was the failed broker.
func (le *LeaderElector) OnBrokerFailure(failedBrokerID int32) {
    // TODO 1: Query metadataStore for all partitions where leader == failedBrokerID
    // TODO 2: For each such partition:
    //   a. Get current ISR from metadata
    //   b. Remove failedBrokerID from ISR (if present)
    //   c. If ISR is empty, log error and mark partition offline (no leader)
    //   d. Else:
    //        i. Select new leader: replica in ISR with highest LEO (from metadata)
    //       ii. Send BeginLeaderElection RPC to selected broker
    //      iii. On success, update metadataStore with new leader and ISR
    //       iv. Propagate metadata update to all brokers
}

// selectNewLeader chooses a replica from ISR to become leader.
func (le *LeaderElector) selectNewLeader(isr []int32, partitionInfo types.PartitionMetadata) int32 {
    // TODO 1: If ISR is empty, return -1 (no leader possible)
    // TODO 2: Fetch LEO for each replica in ISR from metadata (may be stale)
```

```
    // TODO 3: Sort replicas by LEO descending, then by brokerID ascending
    // TODO 4: Return the first replica ID
    return -1
}
```

E. Language-Specific Hints

- **Concurrency:** Use `sync.RWMutex` to protect `Partition.ISR` and `Partition.HighWatermark`. Followers update high watermark on read; leaders update it and ISR on write.
- **Disk Persistence:** When a follower appends records, it must call `log.Sync()` if the leader's request had `acks=all` to ensure durability before acknowledging to leader.
- **Context for Cancellation:** Pass `context.Context` to all long-running loops (follower sync, ISR monitor) to allow graceful shutdown on broker stop.
- **Time:** Use `time.Time` for last fetch timestamps. `time.Since()` is convenient for calculating lag duration.
- **Network Errors:** Use `net.Error` type checks to differentiate temporary vs permanent network failures. Implement exponential backoff with `time.Sleep` for temporary errors.

F. Milestone Checkpoint

After implementing replication, you should be able to verify the following scenario:

1. **Start a 3-broker cluster** (e.g., on ports 9092, 9093, 9094). Designate broker 1 as the coordinator/controller.
2. **Create a topic** with 1 partition and replication factor 3.
3. **Produce messages** with `acks=all`. Observe logs: the leader should append, and followers should fetch and append.
4. **Kill the leader broker** (e.g., kill -9). Wait for the session timeout (e.g., 10 seconds).
5. **Produce more messages** to the same topic. The new leader should be elected and accept writes.
6. **Read all messages** from the beginning. You should see **no data loss**—all messages from step 3 and step 5 should be present, in correct offset order.

Expected Logs:

- Follower logs: "Fetching from leader broker X at offset Y"
- Leader logs: "Updated ISR to [1,2,3]"
- Coordinator logs: "Broker 2 failed, electing new leader for topic T partition 0"
- New leader logs: "Assuming leadership for topic T partition 0, epoch Z"

Debugging Tips:

- If followers are not fetching, check that they know the correct leader (metadata cache).
- If ISR shrinks unexpectedly, check follower fetch frequency and network latency.
- If high watermark does not advance, ensure leader is calculating min LEO across ISR correctly.

---

9. Interactions and Data Flow

> **Milestone(s):** 2 (Producer), 3 (Consumer Groups), 4 (Replication)

Understanding how system components interact is crucial for building a correct distributed message queue. This section details the choreography between producers, brokers, consumers, and coordinators through three key perspectives: sequence diagrams showing the temporal flow of operations, the binary wire protocol that defines their communication format, and the coordination service that

maintains cluster metadata. These three layers—behavioral, syntactic, and infrastructural—form the foundation of reliable distributed communication.

9.1 Key Sequence Diagrams

Sequence diagrams provide a visual timeline of interactions between system components, helping developers understand the causality and concurrency in distributed operations. Think of these diagrams as **air traffic control logs**—they show which aircraft (components) communicated with the tower (brokers) at what time, what instructions were exchanged, and how the overall flow of traffic is coordinated to avoid collisions and ensure safe delivery.

9.1.1 Message Production Flow

The producer-to-broker interaction follows a "prepare-route-send-acknowledge" pattern that balances throughput with durability guarantees. This flow resembles a **courier service delivery process**: a sender packages multiple items into a single shipment (batching), addresses them to specific destinations (partition selection), dispatches them via the most efficient route (network send), and waits for delivery confirmation (acknowledgment).

The complete production sequence involves these steps:

1. **Metadata Resolution**: The producer consults its local `MetadataCache` to determine which broker leads the target partition. If cache is stale or missing, it sends a metadata request to any bootstrap broker.
2. **Record Preparation**: For each message, the producer:
   - Serializes key, value, and headers into a `Record`
   - Applies optional compression to the entire batch
   - Calculates CRC32 checksum for data integrity verification
3. **Partition Assignment**: Using the configured `Partitioner` (typically `HashPartitioner`), the producer determines the target partition:
   - For records with keys: `hash(key) % partition_count` ensures consistent mapping
   - For null-key records: round-robin distribution across partitions
   - The partition assignment determines which broker (leader) receives the batch
4. **Batch Accumulation**: Records are grouped by `TopicPartition` in the `Accumulator`:
   - Batches grow until reaching `BatchSize` bytes or `LingerMs` timeout
   - Each batch becomes a `RecordBatch` with common metadata (base offset, timestamps)
   - The accumulator maintains an in-memory buffer limited by `BufferMemory`
5. **Network Dispatch**: The `Sender` thread retrieves ready batches and:

- Establishes or reuses a TCP connection to the partition leader
- Encodes the batch using the binary protocol (Section 9.2)
- Sends the request with the configured `Acks` level

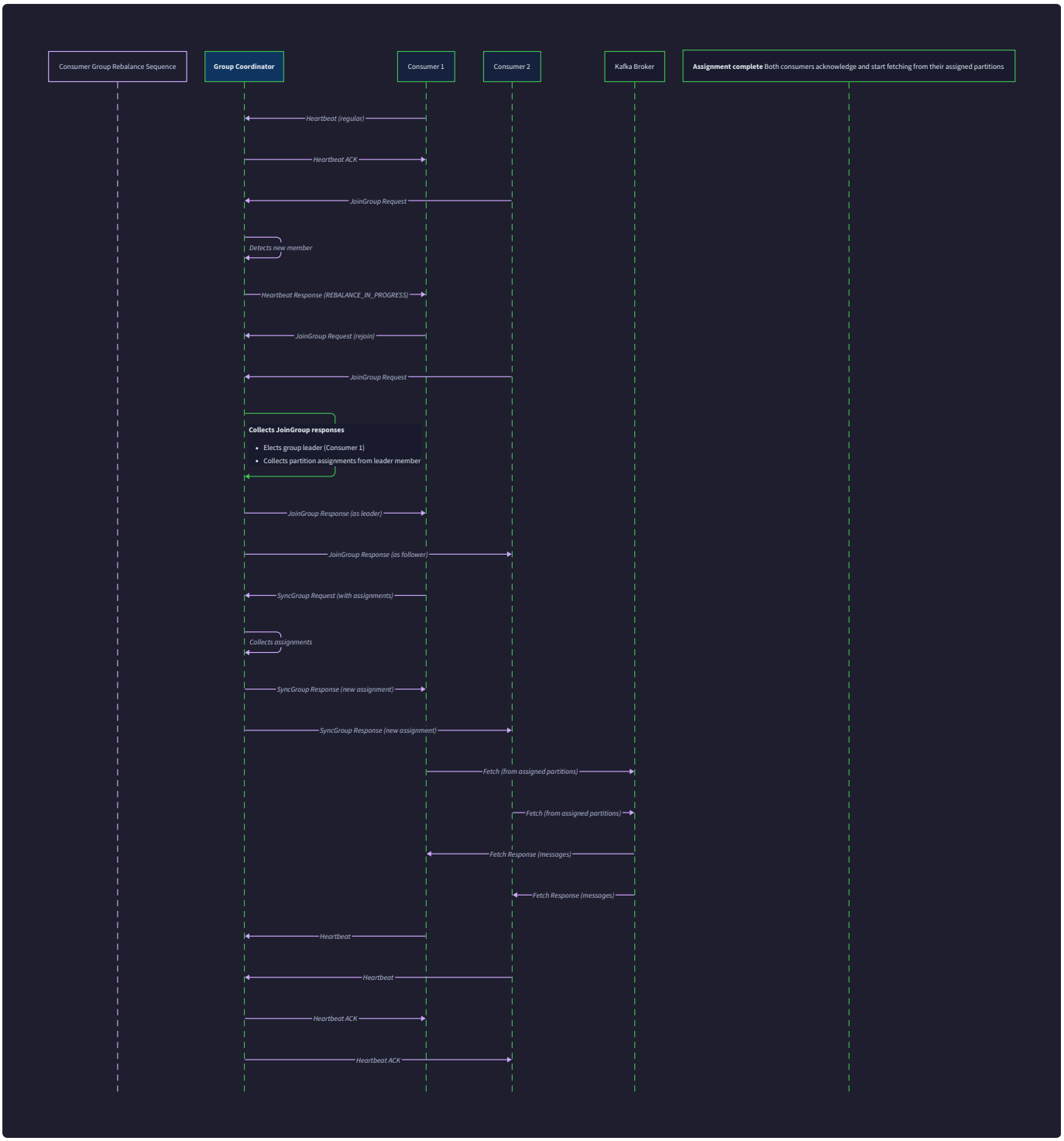
6. **Acknowledgement Handling**: Based on the `Acks` configuration:
   - `AcksNone` (0): Fire-and-forget—no response expected, maximum throughput
   - `AcksLeader` (1): Wait for leader to append to its local log (fsync optional)
   - `AcksAll` (-1): Wait for all ISR replicas to acknowledge (highest durability)
   - On timeout or error, `RetryItem` is scheduled with exponential backoff
7. **Delivery Completion**: Successful acknowledgment includes:
   - Partition ID and starting offset for the batch
   - Leader epoch for fencing protection
   - Error code (0 for success)
   - The producer invokes any registered callback with this information

The critical insight is that batching happens at **two levels**: within the producer (multiple records) and within the broker (multiple batches to disk). This double batching amortizes network and disk I/O overhead, making high-throughput streaming possible.

Step	Component	Action	Key Consideration
1	Producer	Fetch metadata	Cache TTL vs. leader change frequency
2	Producer	Serialize record	Compression trade-off: CPU vs. bandwidth
3	Partitioner	Select partition	Hash consistency during partition count changes
4	Accumulator	Group records	Memory pressure vs. latency trade-off
5	Sender	Send batch	Connection pooling and TCP congestion control
6	Broker	Process request	Durability vs. latency based on acks
7	Broker	Send response	Include leader epoch for client fencing

9.1.2 Consumer Group Rebalancing Flow

Consumer group rebalancing is a distributed consensus protocol that redistributes partitions when membership changes. Imagine a **sports team drafting players**: when a new player joins or leaves, the team captain (coordinator) reassesses everyone's positions, reallocates responsibilities, and ensures each player knows exactly what territory they cover.



The rebalancing protocol follows a three-phase "join-sync-stabilize" pattern:

Phase 1: Membership Change Detection

1. A new consumer calls `Subscribe()` or an existing consumer fails to send `Heartbeat` within `SessionTimeoutMs`
2. The `GroupCoordinator` detects membership change and transitions group state to `GroupStatePreparingRebalance`
3. All existing members receive `ErrRebalanceInProgress` on their next heartbeat, signaling them to rejoin

Phase 2: Join Group Synchronization 4. Each consumer (including the new one) sends `JoinGroupRequest` containing:

- `GroupID` : Identifier for the consumer group
- `MemberID` : Current member ID (empty for new members)
- `ProtocolType` : Always "consumer" for this system

- `Protocols`: List of supported partition assignment strategies (range, round-robin)
- `SessionTimeoutMs`: Maximum time without heartbeat before considered dead

5. The coordinator waits up to `RebalanceTimeoutMs` for all expected members to join:
   - First joiner becomes the "leader" consumer (selected by coordinator)
   - Coordinator records each member's subscribed topics and supported protocols
   - If timeout expires, proceeds with currently joined members (stragglers excluded)

Phase 3: Assignment Distribution 6. The coordinator sends `JoinGroupResponse` to all members containing:

- `GenerationID`: Incremented epoch number for this assignment
- `LeaderID`: Member ID of the designated leader consumer
- Assigned `MemberID` for each consumer (new members receive generated IDs)

7. The leader consumer computes partition assignments using the selected `PartitionAssigner`:
   - Gathers all subscribed topics from member metadata
   - Applies assignment strategy (range or round-robin) to distribute partitions
   - Creates `Assignment` mapping from member ID to topic-partition list
8. Each member sends `SyncGroupRequest`:
   - Leader includes the computed assignments in its request
   - Followers send empty assignment payload
9. Coordinator distributes assignments via `SyncGroupResponse`:
   - Validates leader's assignment covers all partitions
   - Sends each member their specific assigned partitions
   - Updates stored group metadata with new generation

Phase 4: Stable Operation 10. Upon receiving assignments, each consumer: - Updates its `GroupMember` state to `StateStable` - Begins fetching from assigned partitions starting from committed offsets - Resumes periodic heartbeats to maintain membership

11. The coordinator transitions group to `GroupStateStable` and monitors heartbeats

The protocol ensures **exactly-once partition ownership** within a generation: while a consumer holds a generation ID, it uniquely owns its assigned partitions. If a consumer fails to heartbeat, the coordinator increments the generation, invalidating the previous owner's claims.

Rebalance Trigger	Detection Mechanism	Coordinator Action	Consumer Action
New consumer joins	`JoinGroupRequest` from unknown member	Transition to `GroupStatePreparingRebalance`	Re-send join after receiving error
Consumer leaves	Heartbeat timeout after `SessionTimeoutMs`	Mark member dead, trigger rebalance	N/A (consumer is dead)
Consumer crashes	TCP connection drop + heartbeat timeout	Same as leave	N/A (consumer crashed)
Topic metadata changes	Admin API or broker notification	Trigger rebalance if partition count changed	Rejoin when notified
Manual rebalance	External command via API	Force transition to rebalancing state	All members receive error code

9.1.3 Message Consumption Flow

Consumer message fetching follows a "poll-fetch-advance-commit" cycle that balances throughput with memory constraints. Picture a **library checkout system**: patrons (consumers) periodically visit the library (broker), check out several books (records) at once, read them at their own pace, and update their checkout record (commit offset) before returning for more.

The consumption sequence proceeds as follows:

1. **Subscription Initialization**: Consumer calls `Subscribe()` which triggers the join-sync protocol (Section 9.1.2), resulting in assigned partitions.
2. **Offset Initialization**: For each assigned partition, consumer determines starting position:
   - If `auto.offset.reset` = "earliest": starts at partition's lowest available offset
   - If "latest": starts at current `LogEndOffset` (only new messages)
   - If valid committed offset exists: resumes from `OffsetStore` persisted value
   - Consumer maintains `FetchOffset` per partition as its read position
3. **Fetch Request Preparation**: Consumer builds `FetchRequest` containing:
   - Maximum wait time (`MaxWaitMs`) for broker to accumulate data
   - Minimum bytes (`MinBytes`) to return before satisfying request
   - Maximum bytes (`MaxBytes`) per partition to prevent memory overflow
   - Per-partition fetch state: `TopicPartition` , `FetchOffset` , `MaxBytes`
4. **Broker Processing**: For each requested partition, broker leader:
   - Validates consumer has read permission and offset is within range
   - Reads from `Log` starting at `FetchOffset` up to `MaxBytes` or log end
   - Returns records with `HighWatermark` (last safely replicable offset)
   - If `FetchOffset` ≥ `HighWatermark` , waits up to `MaxWaitMs` for new data
   - Returns empty batch if no new data after wait period
5. **Record Delivery**: Consumer receives `FetchResponse` and:
   - Processes records in offset order within each partition
   - Delivers to application via `Poll()` return or callback
   - Advances `FetchOffset` past delivered records
   - Updates metrics (bytes consumed, lag, throughput)
6. **Offset Commitment**: Periodically or explicitly, consumer commits progress:
   - `CommitSync()` blocks until offset stored durably
   - `CommitAsync()` returns immediately, calls callback on completion
   - Offset committed to `__consumer_offsets` internal topic via broker
   - Coordinator validates generation ID to prevent stale commits
7. **Rebalance Handling**: If rebalance occurs during consumption:
   - Consumer completes current batch processing
   - Commits offsets for revoked partitions
   - Releases partition ownership (stops fetching)
   - Rejoins group for new assignment

The **fetch session** optimization is crucial: consumers maintain long-lived connections to brokers, sending incremental fetch requests that only list changed partitions. This reduces request size and parsing overhead for steady-state consumption.

9.2 Wire Protocol Specification

The wire protocol defines the binary format for all network communication between components. Think of it as the **international shipping container standard**: just as standardized containers enable efficient global logistics regardless of cargo type, a well-defined binary protocol enables interoperability between different client implementations and server versions while minimizing parsing overhead.

9.2.1 Protocol Design Principles

The protocol follows several key design decisions documented in this ADR:

> **Decision: Binary Protocol Over Text Protocol**
>
> - **Context**: Need for high-throughput, low-latency communication between distributed components with efficient parsing and minimal serialization overhead.
> - **Options Considered**:
>   1. **Text-based (JSON/XML)**: Human-readable, easy to debug, but high parsing cost and large payload size
>   2. **Binary with TLV (Type-Length-Value)**: Self-describing, flexible evolution, but header overhead per field
>   3. **Fixed-position binary**: Compact, fast parsing, but requires versioning for schema changes
> - **Decision**: Fixed-position binary protocol with versioned requests and explicit schema evolution rules.
> - **Rationale**: Throughput and latency are primary concerns for a messaging system. Binary encoding reduces serialization costs by 5-10x compared to JSON, and fixed-position parsing avoids heap allocations during deserialization. Versioning allows backward-compatible evolution.
> - **Consequences**: Protocol debugging requires specialized tools, but the performance benefits justify the complexity. Schema changes must follow compatibility rules (add-only fields, bump version).

Protocol Aspect	Design Choice	Rationale	Trade-off
Encoding	Binary, big-endian	Network byte order standard	Not human-readable
Framing	Length-prefixed	Simple parsing, no delimiters	Requires length prefix overhead
Versioning	Per-request API key + version	Backward compatibility	Multiple code paths
Error handling	Error code per response partition	Granular failure reporting	Additional complexity
Compression	Per-record-batch	Reduces network bandwidth	CPU overhead, delayed batching

9.2.2 Common Request/Response Structure

All protocol messages follow the same envelope structure:

Field	Type	Description
`Size`	int32	Total message size in bytes (excluding this field)
`APIKey`	int16	Numeric identifier for the request type (Produce=0, Fetch=1, etc.)
`APIVersion`	int16	Version of the API for forward/backward compatibility
`CorrelationID`	int32	Client-generated ID to match responses to requests
`ClientID`	string	Client identifier for debugging (nullable)
`Request/Response Body`	variable	API-specific structured data

The **version negotiation** process: clients send requests with their supported version; brokers respond with the same version if supported, or downgrade/error if not. This enables gradual feature rollout without breaking existing clients.

9.2.3 Produce Request/Response Format

The Produce API writes records to partition logs with configurable durability guarantees.

ProduceRequest (API Key: 0)

Field	Type	Description
`TransactionID`	int64	For idempotent producers, 0 otherwise
`Acks`	int16	Required acknowledgments: -1=all, 0=none, 1=leader
`TimeoutMs`	int32	Maximum time to wait for `Acks` fulfillment
`TopicData`	array	Topics to produce to (see below)

TopicData Array Element

Field	Type	Description
`Topic`	string	Topic name
`Data`	array	Partition data (see below)

PartitionData Array Element

Field	Type	Description
`Partition`	int32	Partition index
`RecordSet`	bytes	Serialized `RecordBatch` (Section 4.2)

ProduceResponse

Field	Type	Description
`Responses`	array	Per-topic response (see below)
`ThrottleTimeMs`	int32	Time client should wait before next request

TopicResponse Array Element

Field	Type	Description
`Topic`	string	Topic name
`PartitionResponses`	array	Per-partition response (see below)

PartitionResponse Array Element

Field	Type	Description
`Partition`	int32	Partition index
`ErrorCode`	int16	0=success, non-zero error codes
`BaseOffset`	int64	Offset of first record in batch
`LogAppendTime`	int64	Broker timestamp when appended
`LogStartOffset`	int64	New log start offset after compaction

The `RecordSet` field contains a complete serialized `RecordBatch` as defined in Section 4.2, including compression. Brokers validate CRC before appending to log.

9.2.4 Fetch Request/Response Format

The Fetch API reads records from partition logs with configurable batching and wait semantics.

FetchRequest (API Key: 1)

Field	Type	Description
`ReplicaID`	int32	-1 for consumers, broker ID for replication
`MaxWaitMs`	int32	Maximum time to block waiting for `MinBytes`
`MinBytes`	int32	Minimum bytes to accumulate before responding
`MaxBytes`	int32	Maximum bytes to return in response
`IsolationLevel`	int8	0=read uncommitted, 1=read committed (≤ high watermark)
`SessionID`	int32	Fetch session ID for incremental fetches
`SessionEpoch`	int32	Fetch session epoch for incremental fetches
`Topics`	array	Topics to fetch from (see below)
`ForgottenTopics`	array	Topics to remove from session (incremental only)

TopicFetch Array Element

Field	Type	Description
`Topic`	string	Topic name
`Partitions`	array	Partitions to fetch (see below)

PartitionFetch Array Element

Field	Type	Description
`Partition`	int32	Partition index
`FetchOffset`	int64	Starting offset to fetch from
`LogStartOffset`	int64	Earliest available offset (for truncation detection)
`MaxBytes`	int32	Maximum bytes to fetch for this partition

FetchResponse

Field	Type	Description
`ThrottleTimeMs`	int32	Time client should wait before next request
`ErrorCode`	int16	Top-level error (0=success)
`SessionID`	int32	Fetch session ID (if incremental)
`Responses`	array	Per-topic response (see below)

TopicFetchResponse Array Element

Field	Type	Description
`Topic`	string	Topic name
`PartitionResponses`	array	Per-partition response (see below)

PartitionFetchResponse Array Element

Field	Type	Description
`Partition`	int32	Partition index
`ErrorCode`	int16	Partition-level error (0=success)
`HighWatermark`	int64	Last committed offset in partition
`LastStableOffset`	int64	Last stable offset (for transactions)
`LogStartOffset`	int64	Earliest available offset
`AbortedTransactions`	array	List of aborted transactions (for read committed)
`RecordSet`	bytes	Serialized `RecordBatch` (empty if no data)

The **incremental fetch session** optimization: when `SessionID` ≠ 0, broker remembers requested partitions across requests. Subsequent requests only need to list changed partitions (`ForgottenTopics` to remove), reducing request size significantly.

9.2.5 JoinGroup Request/Response Format

The JoinGroup API coordinates consumer group membership and leader election.

JoinGroupRequest (API Key: 11)

Field	Type	Description
`GroupID`	string	Consumer group identifier
`SessionTimeoutMs`	int32	Time after which inactive members are removed
`RebalanceTimeoutMs`	int32	Maximum time coordinator waits for members
`MemberID`	string	Current member ID (empty for new members)
`ProtocolType`	string	"consumer" for consumer groups
`Protocols`	array	Supported partition assignment protocols (see below)

GroupProtocol Array Element

Field	Type	Description
`Name`	string	Protocol name (e.g., "range", "roundrobin")
`Metadata`	bytes	Serialized protocol metadata (see below)

ProtocolMetadata Structure

Field	Type	Description
`Version`	int16	Metadata format version
`Topics`	array	Subscribed topics list
`UserData`	bytes	Optional custom data

JoinGroupResponse

Field	Type	Description
`ErrorCode`	int16	Top-level error (0=success)
`GenerationID`	int32	Group generation identifier (increments each rebalance)
`ProtocolName`	string	Selected protocol by coordinator
`LeaderID`	string	Member ID of group leader
`MemberID`	string	Assigned member ID for this consumer
`Members`	array	Group member metadata (only returned to leader)

MemberMetadata Array Element

Field	Type	Description
`MemberID`	string	Member identifier
`Metadata`	bytes	Serialized protocol metadata (same as request)

The coordinator selects the **protocol** based on member intersection: if all members support "range", it's selected; otherwise, coordinator picks highest common protocol. This enables rolling upgrades of assignment strategies.

9.3 Coordination Service Abstraction

The coordination service maintains cluster metadata and enables consensus for critical operations like leader election. Think of it as the **air traffic control database**: it doesn't control planes directly but maintains the authoritative registry of which runways are active, which controllers are on duty, and which flight plans are approved—information all pilots and towers reference to coordinate safely.

9.3.1 Minimal Coordination Interface

For our educational system, we define a simplified coordination interface that can be implemented with various backends (in-memory, etcd, ZooKeeper). This abstraction follows the **registry pattern**: components register themselves and subscribe to changes, receiving notifications when relevant metadata updates.

Core Coordination Interface Methods:

Method Signature	Parameters	Returns	Description
`RegisterBroker(broker *Broker) error`	`broker`: Broker metadata	error	Registers a broker with the cluster; generates unique ID if not set
`DeregisterBroker(brokerID int) error`	`brokerID`: Broker identifier	error	Removes broker from cluster; triggers partition reassignment
`GetBrokers() ([]*Broker, error)`	None	Broker list, error	Returns all currently registered brokers
`CreateTopic(topic *TopicMetadata) error`	`topic`: Topic configuration	error	Creates topic with specified partitions and replication factor
`DeleteTopic(topicName string) error`	`topicName`: Topic to delete	error	Marks topic for deletion; brokers clean up replicas
`GetTopicMetadata(topicName string) (*TopicMetadata, error)`	`topicName`: Topic name	Metadata, error	Returns current partition leadership and replica assignments
`UpdatePartitionLeadership(partition *PartitionMetadata) error`	`partition`: Updated metadata	error	Updates leader and ISR for a partition (atomic compare-and-swap)
`ElectPartitionLeader(topic string, partition int, isr []int) (int, error)`	`topic`, `partition`, `isr`	leaderID, error	Selects new leader from ISR using deterministic algorithm
`WatchBrokers(callback func([]*Broker)) (cancel func(), error)`	`callback`: Change handler	cancel function, error	Registers for broker list changes (for client metadata updates)
`WatchTopic(topicName string, callback func(*TopicMetadata)) (cancel func(), error)`	`topicName`, `callback`	cancel function, error	Registers for topic metadata changes (for producer routing)

State Management Guarantees:

> **Design Principle: Eventual Consistency with Strong Consistency for Critical Paths**
>
> - **Metadata reads** may be slightly stale (cached for performance)
> - **Leadership elections** are linearizable (all observers see same leader)
> - **Partition assignment** uses compare-and-swap to prevent lost updates
> - **Watch notifications** are at-least-once (may receive duplicates)

9.3.2 Metadata Propagation Patterns

Cluster metadata flows through the system using a **publish-subscribe with cache invalidation** pattern. This minimizes coordination traffic while ensuring timely updates:

1. **Broker Registration Flow**:

```
Broker startup → RegisterBroker() → Coordination service stores →
WatchBrokers() callbacks fire → All brokers update ClusterMetadata →
Clients fetch metadata on next request → Updated routing tables
```

2. **Leader Election Flow**:

```
Leader failure detected → ElectPartitionLeader() → Coordination service selects →
UpdatePartitionLeadership() → WatchTopic() callbacks fire →
Followers update their replica state → Clients get ErrNotLeaderForPartition →
Client metadata refresh → New routing established
```

3. **Topic Creation Flow**:

```
Admin creates topic → CreateTopic() → Coordination service validates →
Assign partitions to brokers → Store TopicMetadata →
WatchTopic() callbacks fire → Brokers create local log directories →
Clients can produce/consume
```

The **metadata cache lifecycle** in clients:

- Cache initialized with bootstrap broker list
- On each request, check cache freshness (timestamp)
- On `ErrNotLeaderForPartition` or `ErrUnknownTopicOrPartition`, force refresh
- Refresh interval: min(metadata.max.age.ms, calculated based on error rate)
- Cache entries include generation ID to detect stale updates

9.3.3 Fault Tolerance Considerations

The coordination service itself must be highly available. Our abstraction supports different implementations with varying consistency guarantees:

Implementation	Consistency Model	Failure Handling	Recommended Use
In-memory with single leader	Strong consistency (leader)	Leader election via Raft/ZooKeeper	Small clusters, educational use
Embedded etcd	Linearizable via Raft	Automatic failover, data replication	Production-like deployment
External ZooKeeper	Sequential consistency	Ensemble quorum maintenance	Integration with existing infra
Gossip-based	Eventual consistency	No single point of failure	Large-scale, AP-focused systems

Critical coordination paths requiring strong consistency:

1. **Partition leadership**: To prevent split-brain (two brokers believing they're leader)
2. **Consumer group generation**: To prevent duplicate processing during rebalance
3. **Transaction state**: For exactly-once semantics (future extension)
4. **Controller election**: To ensure single active controller broker

Eventual consistency acceptable for:

1. **Broker liveness**: Slight delay in detecting failures acceptable
2. **Topic configuration**: Temporary inconsistency doesn't cause data loss
3. **Client metadata**: Stale routing corrected via error retry

The coordination service represents a **single point of truth** but not a **single point of failure** when implemented with replication and automatic failover.

9.4 Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Protocol Serialization	Manual byte packing/unpacking	Protocol Buffers with code generation
Connection Management	`net.Conn` with manual pooling	Custom connection pool with keep-alive
Compression	None (simplicity)	`compress/gzip` or third-party Snappy
Coordination Service	In-memory with file persistence	Embedded etcd (`go.etcd.io/etcd/client/v3`)

Recommended File Structure

```
project-root/
  cmd/
    broker/               # Broker entry point
      main.go
    producer/             # Producer CLI example
      main.go
    consumer/             # Consumer CLI example
      main.go
  internal/
    protocol/             # Wire protocol definitions
      api_keys.go         # API key constants (Produce=0, Fetch=1, etc.)
      requests.go         # Request structs and serialization
      responses.go        # Response structs and serialization
      errors.go           # Error code definitions
      framing.go          # Length-prefixed message framing
    network/
      connection.go       # TCP connection wrapper with timeout
      pool.go             # Connection pool management
      server.go           # TCPServer implementation
    coordination/         # Coordination service abstraction
      interface.go        # CoordinationService interface
      memory/             # In-memory implementation
        store.go          # In-memory metadata store
        watcher.go        # Watch notification system
      etcd/               # etcd implementation (optional)
        client.go
    metadata/             # Client metadata cache
      cache.go            # Metadata caching with TTL
      refresher.go        # Background metadata refresh
```

Protocol Framing Infrastructure (Complete)

```go
// internal/protocol/framing.go

package protocol

import (
    "encoding/binary"
    "errors"
    "io"
)

var (
    ErrMessageTooLarge = errors.New("message size exceeds maximum allowed")
    ErrInvalidSize     = errors.New("invalid message size field")
)

const (
    MaxMessageSize = 100 * 1024 * 1024 // 100 MB maximum message size
    SizeFieldLen   = 4                 // int32 size prefix
)

// ReadMessage reads a length-prefixed message from the reader
func ReadMessage(r io.Reader) ([]byte, error) {
    // Read size prefix (4 bytes, big-endian)
    var sizeBuf [SizeFieldLen]byte
    if _, err := io.ReadFull(r, sizeBuf[:]); err != nil {
        return nil, err
    }

    size := int32(binary.BigEndian.Uint32(sizeBuf[:]))
    if size < 0 || size > MaxMessageSize {
        return nil, ErrInvalidSize
    }

    // Allocate buffer and read message body
    buf := make([]byte, size)
```

```
    if _, err := io.ReadFull(r, buf); err != nil {
        return nil, err
    }

    return buf, nil
}

// WriteMessage writes a length-prefixed message to the writer
func WriteMessage(w io.Writer, data []byte) error {
    size := int32(len(data))
    if size > MaxMessageSize {
        return ErrMessageTooLarge
    }

    // Write size prefix
    var sizeBuf [SizeFieldLen]byte
    binary.BigEndian.PutUint32(sizeBuf[:], uint32(size))
    if _, err := w.Write(sizeBuf[:]); err != nil {
        return err
    }

    // Write message body
    _, err := w.Write(data)
    return err
}

// MessageHeader represents the common header for all requests/responses
type MessageHeader struct {
    Size          int32  // Already read by framing layer
    APIKey        int16
    APIVersion    int16
    CorrelationID int32
    ClientID      string // Nullable string
}
```

```go
// DecodeHeader parses the common header from a message buffer
func DecodeHeader(buf []byte) (*MessageHeader, error) {
    if len(buf) < 10 { // Minimum header size (without ClientID)
        return nil, errors.New("buffer too small for header")
    }

    h := &MessageHeader{}
    pos := 0

    // APIKey (int16)
    h.APIKey = int16(binary.BigEndian.Uint16(buf[pos:]))
    pos += 2

    // APIVersion (int16)
    h.APIVersion = int16(binary.BigEndian.Uint16(buf[pos:]))
    pos += 2

    // CorrelationID (int32)
    h.CorrelationID = int32(binary.BigEndian.Uint32(buf[pos:]))
    pos += 4

    // ClientID (nullable string: int16 length + bytes)
    clientIDLen := int16(binary.BigEndian.Uint16(buf[pos:]))
    pos += 2

    if clientIDLen == -1 {
        h.ClientID = ""
    } else if clientIDLen >= 0 {
        if pos+int(clientIDLen) > len(buf) {
            return nil, errors.New("buffer underflow for ClientID")
        }
        h.ClientID = string(buf[pos : pos+int(clientIDLen)])
        pos += int(clientIDLen)
```

```
    } else {
        return nil, errors.New("invalid ClientID length")
    }

    return h, nil
}

// EncodeHeader serializes the header to bytes
func EncodeHeader(h *MessageHeader) ([]byte, error) {
    // Calculate total size (APIKey + APIVersion + CorrelationID + ClientID length + ClientID)
    size := 2 + 2 + 4 + 2 + len(h.ClientID)
    buf := make([]byte, size)
    pos := 0

    // APIKey
    binary.BigEndian.PutUint16(buf[pos:], uint16(h.APIKey))
    pos += 2

    // APIVersion
    binary.BigEndian.PutUint16(buf[pos:], uint16(h.APIVersion))
    pos += 2

    // CorrelationID
    binary.BigEndian.PutUint32(buf[pos:], uint32(h.CorrelationID))
    pos += 4

    // ClientID (nullable string)
    if h.ClientID == "" {
        binary.BigEndian.PutUint16(buf[pos:], uint16(-1))
        pos += 2
    } else {
        binary.BigEndian.PutUint16(buf[pos:], uint16(len(h.ClientID)))
        pos += 2
        copy(buf[pos:], h.ClientID)
```

```
        pos += len(h.ClientID)
    }


    return buf, nil
}
```

Core Protocol Serialization Skeleton

```go
// internal/protocol/requests.go

package protocol

import (
    "encoding/binary"
    "errors"
    "fmt"
)

// ProduceRequest represents a produce API request
type ProduceRequest struct {
    TransactionalID string
    Acks          int16
    TimeoutMs     int32
    TopicData     []TopicProduceData
}

// TopicProduceData represents topic-level produce data
type TopicProduceData struct {
    Topic      string
    Partitions []PartitionProduceData
}

// PartitionProduceData represents partition-level produce data
type PartitionProduceData struct {
    Partition  int32
    RecordSet  []byte // Serialized RecordBatch
}

// DecodeProduceRequest parses a ProduceRequest from bytes
func DecodeProduceRequest(apiVersion int16, buf []byte) (*ProduceRequest, error) {
    req := &ProduceRequest{}
    pos := 0

    // TODO 1: For apiVersion >= 3, read TransactionalID (nullable string)
```

```
    // Hint: Read int16 length, if -1 set to "", else read that many bytes


    // TODO 2: Read Acks (int16)


    // TODO 3: Read TimeoutMs (int32)


    // TODO 4: Read array size (int32) for TopicData
    // For each topic:
    //   a) Read topic name (nullable string)
    //   b) Read array size (int32) for Partitions
    //   c) For each partition:
    //        i) Read Partition (int32)
    //        ii) Read RecordSet size (int32) and bytes
    //        iii) Append to Partitions slice


    // TODO 5: Validate buffer boundaries (pos should equal len(buf))


    return req, nil
}

// EncodeProduceRequest serializes a ProduceRequest to bytes
func EncodeProduceRequest(req *ProduceRequest, apiVersion int16) ([]byte, error) {
    // TODO 1: Calculate total buffer size needed
    // Hint: Iterate through all nested structures counting bytes


    // TODO 2: Allocate buffer with correct size


    // TODO 3: Write TransactionalID if apiVersion >= 3
    // Hint: Nullable string: length (-1 for null) then bytes


    // TODO 4: Write Acks (int16, big-endian)


    // TODO 5: Write TimeoutMs (int32)
```

```
    // TODO 6: Write TopicData array size (int32)
    // For each topic:
    //   a) Write topic name (nullable string)
    //   b) Write Partitions array size (int32)
    //   c) For each partition:
    //        i) Write Partition (int32)
    //        ii) Write RecordSet length (int32) followed by bytes

    // TODO 7: Return buffer

    return nil, nil
}

// HandleProduceRequest processes a produce request on the broker
func HandleProduceRequest(req *ProduceRequest, broker *Server) (*ProduceResponse, error) {
    resp := &ProduceResponse{
        Responses: make([]TopicProduceResponse, 0, len(req.TopicData)),
    }

    for _, topicData := range req.TopicData {
        topicResp := TopicProduceResponse{
            Topic:              topicData.Topic,
            PartitionResponses: make([]PartitionProduceResponse, 0, len(topicData.Partitions)),
        }

        for _, partitionData := range topicData.Partitions {
            partitionResp := PartitionProduceResponse{
                Partition: partitionData.Partition,
            }

            // TODO 1: Validate topic exists and partition is valid

            // TODO 2: Check if this broker is leader for the partition
            // If not, set ErrorCode = ErrNotLeaderForPartition
```

```
            // TODO 3: Validate RecordSet CRC and decompress if needed


            // TODO 4: Append records to partition log
            // baseOffset, err := broker.logManager.Append(...)


            // TODO 5: Based on Acks, wait for replication
            // - AcksNone: immediate success
            // - AcksLeader: wait for local fsync
            // - AcksAll: wait for all ISR acknowledgment


            // TODO 6: Set response fields: BaseOffset, LogAppendTime, LogStartOffset


            topicResp.PartitionResponses = append(topicResp.PartitionResponses, partitionResp)
        }


        resp.Responses = append(resp.Responses, topicResp)
    }


    return resp, nil
}
```

Coordination Service Skeleton

```go
// internal/coordination/interface.go

package coordination

import (
    "context"
    "time"
)

// CoordinationService defines the minimal interface for cluster coordination
type CoordinationService interface {
    // Broker management
    RegisterBroker(ctx context.Context, broker *Broker) error
    DeregisterBroker(ctx context.Context, brokerID int) error
    GetBrokers(ctx context.Context) ([]*Broker, error)

    // Topic management
    CreateTopic(ctx context.Context, topic *TopicMetadata) error
    DeleteTopic(ctx context.Context, topicName string) error
    GetTopicMetadata(ctx context.Context, topicName string) (*TopicMetadata, error)
    ListTopics(ctx context.Context) ([]string, error)

    // Partition leadership
    UpdatePartitionLeadership(ctx context.Context, partition *PartitionMetadata) error
    ElectPartitionLeader(ctx context.Context, topic string, partition int, isr []int) (int, error)
    GetPartitionMetadata(ctx context.Context, topic string, partition int) (*PartitionMetadata, error)

    // Watch/notification system
    WatchBrokers(ctx context.Context, callback func([]*Broker)) (func(), error)
    WatchTopic(ctx context.Context, topicName string, callback func(*TopicMetadata)) (func(), error)

    // Close releases resources
    Close() error
}
```

```go
// In-memory implementation skeleton

// internal/coordination/memory/store.go

package memory


import (

    "context"

    "sync"

    "time"



    "github.com/yourproject/internal/coordination"

    "github.com/yourproject/internal/types"

)


type MemoryCoordinator struct {

    mu sync.RWMutex



    brokers map[int]*types.Broker

    topics  map[string]*types.TopicMetadata



    // Watch subscriptions

    brokerWatchers []func([]*types.Broker)

    topicWatchers  map[string][]func(*types.TopicMetadata)



    closeChan chan struct{}

}


func NewMemoryCoordinator() *MemoryCoordinator {

    return &MemoryCoordinator{

        brokers:       make(map[int]*types.Broker),

        topics:        make(map[string]*types.TopicMetadata),

        topicWatchers: make(map[string][]func(*types.TopicMetadata)),

        closeChan:     make(chan struct{}),

    }

}
```

```go
func (c *MemoryCoordinator) RegisterBroker(ctx context.Context, broker *types.Broker) error {
    c.mu.Lock()
    defer c.mu.Unlock()

    // TODO 1: If broker.ID is 0, assign next available ID

    // TODO 2: Store broker in map

    // TODO 3: Notify all broker watchers
    // Hint: Call each callback in a goroutine to avoid blocking

    return nil
}

func (c *MemoryCoordinator) WatchBrokers(ctx context.Context, callback func([]*types.Broker)) (func(), error) {
    c.mu.Lock()
    defer c.mu.Unlock()

    // TODO 1: Add callback to brokerWatchers slice

    // TODO 2: Return cleanup function that removes the callback

    // TODO 3: Immediately call callback with current broker list

    return func() {}, nil
}

func (c *MemoryCoordinator) ElectPartitionLeader(ctx context.Context, topic string, partition int, isr []int)
(int, error) {
    c.mu.Lock()
    defer c.mu.Unlock()

    // TODO 1: Look up current partition metadata

    // TODO 2: Validate ISR is not empty
```

```
    // TODO 3: Select leader using deterministic algorithm:
    //   - Sort ISR replicas by broker ID
    //   - Pick the replica that was most recently NOT leader (if any)
    //   - Otherwise, pick the first replica in sorted list

    // TODO 4: Update partition metadata with new leader and generation

    // TODO 5: Notify topic watchers of the change

    return 0, nil
}
```

Language-Specific Hints for Go

1. **Binary Serialization**: Use `encoding/binary.BigEndian.PutUint32()` and `binary.BigEndian.Uint32()` for protocol serialization. Create helper functions for nullable strings (int16 length prefix with -1 for null).
2. **Connection Management**: Wrap `net.Conn` with read/write deadlines using `SetReadDeadline()` and `SetWriteDeadline()`. Implement connection pooling with `sync.Pool` for reuse.
3. **Watch Pattern**: Implement watch callbacks using channels or function slices. Use `context.Context` for cancellation. Always call callbacks in goroutines to avoid blocking the coordinator.
4. **Memory Optimization**: For protocol buffers, use `bytes.Buffer` with pre-allocated capacity to reduce allocations. Consider object pooling for frequently allocated structs.
5. **Error Propagation**: Define custom error types for protocol errors with error codes. Use `errors.Is()` and `errors.As()` for error handling in client retry logic.

Milestone Checkpoint: Protocol Implementation

After implementing the wire protocol, test with this verification:

```bash
# Start a broker
go run cmd/broker/main.go --port 9092

# In another terminal, test produce protocol
echo '{"key": "test", "value": "hello"}' | \
go run cmd/producer/main.go --topic test --brokers localhost:9092

# Expected output:
# Message sent to partition 0 at offset 0

# Test consume protocol
go run cmd/consumer/main.go --topic test --group test-group --brokers localhost:9092

# Expected output (after producing):
# Received message at offset 0: key="test", value="hello"

# Verify with network capture
sudo tcpdump -i lo -A port 9092 | grep -A5 -B5 "test"
# Should show binary data (not human-readable)
```

Debugging Tips:

- **Symptom**: "Connection reset by peer" or unexpected EOF
  - **Cause**: Incorrect message framing (size field wrong)
  - **Diagnosis**: Log raw bytes sent/received with hex dump
  - **Fix**: Verify `ReadMessage()` / `WriteMessage()` handle size prefix correctly
- **Symptom**: "Invalid API key" error
  - **Cause**: Wrong byte order or buffer offset
  - **Diagnosis**: Print decoded header fields vs. expected
  - **Fix**: Ensure big-endian encoding throughout
- **Symptom**: Consumer doesn't receive messages after produce
  - **Cause**: Fetch offset not advancing or incorrect high watermark
  - **Diagnosis**: Check FetchResponse contains correct HighWatermark
  - **Fix**: Ensure broker updates high watermark after replication

10. Error Handling and Edge Cases

> **Milestone(s):** 1, 2, 3, 4 (critical for all operational aspects)

Distributed systems operate in a hostile environment where components fail independently and unpredictably. A message queue's reliability hinges not on preventing failures (impossible) but on gracefully recovering from them. This section catalogs the inevitable

failure modes and edge cases your system will encounter, providing a comprehensive playbook for designing robust recovery mechanisms. Think of this as the system's immune system: it must detect pathogens (failures), contain damage (prevent cascading failures), and initiate healing (recovery) while maintaining core functions.

10.1 Failure Mode Categories

Failures in a distributed message queue can be categorized by their origin, duration, and observability. Each category presents distinct challenges for detection and recovery.

10.1.1 Broker Crashes (Process Termination)

Broker crashes occur when a broker process terminates abruptly due to fatal errors, `SIGKILL`, out-of-memory conditions, or administrative shutdown. The broker disappears from the network entirely, leaving its hosted partitions leaderless if they were leading replicas.

Mental Model: The Vanishing Bank Teller Imagine a bank with multiple tellers (brokers) serving customers (clients). One teller suddenly collapses mid-transaction. Customers waiting at that teller's line are stuck, but the bank manager (controller/coordinator) notices the absence and redirects customers to other tellers who have duplicate ledger copies (replicas).

Failure Characteristic	Description	Detection Mechanism
Sudden Termination	Process exits without cleanup (OOM, segmentation fault)	Missing heartbeats in coordination service, TCP connection failures
Graceful Shutdown	Controlled stop with cleanup attempts	Explicit deregistration request before exit
Partial Functionality	Process runs but critical subsystems fail (log full, disk error)	Health checks, failed produce/fetch requests, metrics monitoring

10.1.2 Network Partitions (Split-Brain Scenarios)

Network partitions occur when subsets of brokers can communicate within their group but cannot reach brokers in other groups. This creates "split-brain" scenarios where each partition believes the others have failed, potentially leading to multiple leaders for the same partition.

Mental Model: The Islanded Archipelago Picture an archipelago (broker cluster) where a storm severs communication links between islands. Islands can't coordinate, so each might independently declare itself the new capital (leader) for shared territories (partitions), creating conflicting governance.

Partition Type	Typical Cause	Duration	Impact Severity
Client-Broker	Client network issues, firewall rules	Seconds to minutes	Temporary unavailability for specific clients
Broker-Broker	Switch failures, misconfigured routes	Variable	Replication stalls, ISR shrinkage, potential data inconsistency
Controller Isolation	Controller broker loses connection to majority	Until partition heals	No leadership elections, metadata stagnation

10.1.3 Disk Failures (Storage Corruption)

Disk failures range from complete device failure to silent corruption where reads return incorrect data. The log's durability guarantees depend entirely on stable storage, making disk failures particularly insidious.

Mental Model: The Fading Ledger Envision a ledger (log) whose ink gradually fades (bit rot) or pages tear (sector errors). Transactions recorded yesterday become unreadable today, threatening the entire financial record.

Failure Mode	Manifestation	Detection Challenge
Full Disk	No space for new writes	`ENOSPC` errors on append, monitoring alerts
Corrupted Files	CRC mismatches, invalid offsets	Read validation failures, checksum verification
Slow I/O	Latency spikes beyond tolerance	Timeout monitoring, percentile latency tracking
Controller Failure	RAID/HBA issues affecting all disks	Health checks, SMART monitoring

10.1.4 Client Timeouts and Transient Failures

Clients may experience temporary connectivity issues, processing delays, or resource exhaustion. While not broker failures, these affect system semantics (message delivery guarantees, offset commits).

Mental Model: The Distracted Customer Imagine a customer (consumer) who steps away from the counter (network timeout) while their transaction is being processed. The teller (broker) must decide whether to hold the transaction open or serve the next customer.

Client Issue	Typical Cause	System Impact
Network Timeout	Packet loss, congestion, DNS issues	In-flight requests fail, retries triggered
Processing Stall	GC pause, deadlock, resource exhaustion	Heartbeat failures, session timeout
Clock Skew	NTP misconfiguration, virtualization artifacts	Incorrect timestamps, session expiration confusion

10.2 Recovery Strategies

Each failure category demands specific recovery mechanisms. The system's resilience emerges from the careful orchestration of these strategies across components.

10.2.1 Broker Crash Recovery

When a broker crashes, the system must restore availability without compromising consistency for partitions where the crashed broker was leader.

> **Key Insight:** Recovery from broker crashes follows a two-phase approach: first **detect and isolate** the failed broker, then **reassign and restore** its responsibilities to healthy replicas.

Step-by-Step Recovery Algorithm:

1. **Failure Detection:**
   - Coordination service (`MemoryCoordinator`) monitors broker heartbeats
   - Missing heartbeats for `session.timeout.ms` triggers failure suspicion
   - Controller confirms via TCP connection attempts to all broker ports
   - After confirmation delay (configurable), broker marked as `DEAD`
2. **Leadership Reassignment:**
   - Controller (`LeaderElector`) enumerates all partitions where failed broker was leader
   - For each partition:
     - Remove failed broker from ISR list in `PartitionMetadata`
     - Select new leader from remaining ISR members using deterministic algorithm (e.g., oldest replica in ISR)
     - If ISR is empty, trigger **unclean leader election** decision (see ADR below)
   - Update `PartitionMetadata` across cluster via coordination service
   - Notify all brokers of leadership changes via metadata propagation

3. **Replica Rebuilding (Post-Recovery):**
    - When failed broker restarts, it rejoins cluster as follower for all its assigned partitions
    - For each partition:
        - New leader truncates follower's log to **high watermark** to ensure consistency
        - Follower fetches from leader starting at high watermark + 1
        - Once caught up within `replica.lag.time.max.ms`, rejoins ISR

ADR: Unclean Leader Election Policy

> **Decision: Disallow Unclean Leader Election by Default**
>
> - **Context:** When all ISR replicas for a partition are unavailable (e.g., multiple broker failures), the system faces a choice: wait for an ISR member to recover (availability impact) or elect a non-ISR follower as leader (consistency risk).
> - **Options Considered:**
>   1. **Always wait for ISR:** Never elect non-ISR replicas, guaranteeing no data loss but risking extended unavailability.
>   2. **Allow unclean election:** Elect any available replica when ISR is empty, maximizing availability but risking data loss from un-replicated messages.
>   3. **Configurable policy:** Let operators choose based on topic importance.
> - **Decision:** Implement option 1 (wait for ISR) as default for this educational project, with clear logging when availability is impacted.
> - **Rationale:** Simplicity and safety for learners. Understanding data loss implications requires sophisticated monitoring; better to err on the side of preserving data. Real systems (like Kafka) offer configurable `unclean.leader.election.enable`.
> - **Consequences:** Topics may become unavailable during multi-broker failures, but learners won't silently lose data. Forces consideration of replication factor and broker distribution.

Recovery Strategy	Trigger Condition	Action	Consistency Guarantee
Clean Failover	Leader fails, ISR non-empty	Elect new leader from ISR	No data loss, monotonic offsets
ISR Shrinkage	Follower lags beyond threshold	Remove from ISR, continue with reduced replication	Potentially reduced durability
Unavailable Partition	All ISR members fail	Wait for recovery, reject produce/fetch requests	Strong consistency, availability loss

10.2.2 Network Partition Mitigation

Network partitions create the most complex failure scenarios due to conflicting views of cluster state.

Detection and Containment Strategy:

1. **Dual Monitoring:**
    - **Heartbeat-based:** Coordination service tracks broker liveness via periodic heartbeats
    - **Request-based:** Brokers track peer liveness via replication fetch requests and metadata exchanges
    - Discrepancies between these signals suggest network issues rather than true failures
2. **Fencing via Epochs:**
    - Each partition leader has a monotonically increasing `leader_epoch` stored in `Partition`
    - All client requests include the last known leader epoch
    - Requests with stale epoch are rejected with `ErrNotLeaderForPartition`
    - This prevents clients from accidentally writing to old leaders during partitions

3. **Controller Resilience:**
    - Controller itself can be partitioned from the majority
    - Use **controller epoch** in `ClusterMetadata` to prevent multiple active controllers
    - Brokers reject metadata updates from controllers with stale epoch

Recovery Sequence During Partition Healing:

```
When network partition heals:
1. Previously isolated brokers re-establish TCP connections
2. They discover higher controller/leader epochs from the majority side
3. Isolated leaders step down upon receiving higher epoch notifications
4. Their logs are inspected for divergent writes:
   - Compare high watermarks between old and new leaders
   - Any messages beyond common high watermark are truncated (lost)
5. Brokers rejoin as followers and resynchronize from current leaders
```

10.2.3 Disk Failure Handling

Disk failures threaten the fundamental durability guarantees of the message log.

Tiered Response Strategy:

Failure Severity	Detection	Immediate Response	Long-term Remediation
Full Disk	`ENOSPC` on append	Reject produce requests for affected partitions, continue serving fetches	Add disk space, delete old segments, or migrate partitions
Corrupted Segment	CRC mismatch on read	Mark segment as corrupted, skip to next segment for reads; writes continue to new segment	Attempt repair from replicas, or accept data loss for that segment
Slow I/O	Write latency > `log.flush.timeout.ms`	Log warnings, metrics alerts; continue with degraded performance	Investigate disk health, redistribute load, upgrade hardware
Complete Disk Loss	All I/O operations fail	Mark broker as unhealthy in coordination service, trigger partition reassignment	Replace hardware, restore from replicas, recommission broker

Segmented Isolation Approach: The `Log` design with multiple `LogSegment` files provides natural isolation boundaries. A corrupted segment affects only messages within its offset range, allowing the rest of the partition to remain operational. The system should:

1. Detect corruption via CRC checks in `Log.Read()`
2. Move corrupted segment to quarantine directory
3. Update `Log.Segments` list to skip corrupted segment
4. Log detailed error with segment metadata for admin intervention
5. For replicated partitions, attempt to fetch missing range from other replicas

10.2.4 Client Failure Recovery

Client failures primarily affect consumer progress tracking and producer message delivery semantics.

Consumer Session Recovery:

```
When consumer fails (misses heartbeats):
1. Group coordinator (`ConsumerGroup.State`) marks consumer as dead after `session.timeout.ms`
2. Coordinator triggers rebalance by moving group to `PreparingRebalance` state
3. Remaining consumers rejoin via `JoinGroup`/`SyncGroup`
4. Partitions assigned to failed consumer are redistributed
5. New consumers start fetching from last committed offsets
```

Producer Retry with Idempotence: For `Producer` with `AcksAll`, failed requests require careful retry logic:

```
// Pseudo-algorithm (not code block - prose description)

1. On send failure (timeout, network error, NotLeader error):

2. Refresh metadata cache to discover new partition leader

3. Reassign batch to new leader if partition leadership changed

4. Apply exponential backoff: wait = baseBackoff * 2^attempt

5. Retry up to `Retries` configuration

6. If idempotent producer enabled (optional), include sequence numbers

   to allow broker to deduplicate retried batches
```

GO

10.3 Edge Cases and Race Conditions

Edge cases represent improbable but possible scenarios arising from specific timing of events. Race conditions occur when multiple processes access shared data concurrently, and the outcome depends on the sequence of operations.

10.3.1 Leader Change During Produce Request

Scenario: A producer sends a `ProduceRequest` to partition leader. Mid-processing, leadership changes (old leader crashes or steps down). The request might be partially processed, duplicated, or lost.

Mental Model: The Relaying Runner Imagine a relay race where the baton (message) is passed between runners (brokers). If the current runner (leader) stumbles while holding the baton, and a new runner takes over, what happens to the baton? It might be dropped (lost), held by both (duplicated), or successfully passed (handled correctly).

Timing	Producer Action	Old Leader State	New Leader State	Potential Outcome
Before send	Request in flight	Healthy	Not yet elected	Connection refused, producer retries
During append	Request being written	Crashes mid-write	Elected	Partial write: data may be corrupted or lost
After append, before ack	Waiting for response	Crashes after write	Elected	Duplicate possible if producer retries
After ack, before receipt	Response in flight	Crashes after sending ack	Elected	Producer sees success, consumer may not see data until recovery

Mitigation Strategy:

1. **Leader Epoch in Requests:** Include `leader_epoch` in `ProduceRequest`; new leader rejects requests with stale epoch
2. **Sequence Numbers:** Idempotent producers include `producer_id`, `epoch`, and `sequence_number` for deduplication
3. **High Watermark Validation:** New leader compares its log with old leader's before accepting writes to detect gaps
4. **Client Metadata Refresh:** Producer refreshes metadata on `ErrNotLeaderForPartition` and retries

10.3.2 Duplicate Consumer ID During Rebalance

Scenario: A consumer crashes and restarts quickly, rejoining the group with the same `member_id` before the coordinator has marked it dead. Two instances with the same identity exist temporarily.

Race Condition Timeline:

```
T0: Consumer C1 (member_id="A") sends heartbeat
T1: C1 crashes
T2: C1 restarts, reuses member_id="A", sends JoinGroup
T3: Coordinator still has C1 as active (heartbeat within session timeout)
T4: Coordinator now has two "logical" consumers with same ID
```

Consequences:

- Coordinator might assign partitions to both instances
- Both instances might commit offsets, causing overwrites
- Group state becomes inconsistent

Solution: Generation ID Fencing:

- Each rebalance increments `GenerationID` in `ConsumerGroup`
- Coordinator includes `generation_id` in `JoinGroupResponse`
- Members include `generation_id` in all subsequent requests
- Coordinator rejects requests with stale `generation_id`
- Restarted consumer gets new `member_id` if rejoining after generation change

10.3.3 Partial Write Before Crash (Torn Writes)

Scenario: Broker crashes while appending a `RecordBatch` to disk, writing only part of the batch. On restart, the log contains corrupted data.

Detailed Failure Modes:

Corruption Type	Cause	Detection
Incomplete batch	Crash during `WAL.Append()`	Batch length prefix doesn't match actual bytes
Misaligned offset	Crash between writing record and updating index	Index points to invalid file position
CRC mismatch	Crash during write leaves partial data	CRC validation fails on read

Recovery via Write-Ahead Log with Batch Atomicity:

1. **Write Strategy:** Entire `RecordBatch` written as single `WAL.Append()` operation
2. **CRC Protection:** Compute CRC after serialization, include in header
3. **Fsync Control:** `sync` parameter controls durability trade-off
4. **Restart Recovery:** `Log` scans segments on startup:
    - Read forward until incomplete batch detected (length mismatch)
    - Truncate file to last valid batch boundary
    - Rebuild index from remaining valid data
    - Log truncation offset for admin review

10.3.4 Zombie Consumers in Rebalance

Scenario: A consumer is considered dead by coordinator (missed heartbeats) but is actually alive and still fetching messages. It becomes a "zombie" – processing messages without the coordinator's knowledge.

Root Causes:

1. **Network Partition:** Consumer isolated from coordinator but connected to partition leaders
2. **Heartbeat Thread Stall:** Consumer process alive but heartbeat goroutine blocked
3. **Clock Skew:** Consumer's clock ahead, causing early heartbeat timeout calculation

Dangers:

- Zombie continues committing offsets, interfering with new consumer's progress
- Messages processed twice (by zombie and replacement consumer)
- Offset commits race condition

Fencing via Generation ID:

```
Solution implemented in HandleFetch:
1. Each FetchRequest includes (group_id, member_id, generation_id)
2. Broker validates against local cache of valid consumers
3. If generation_id stale or member not in current generation:
   - Return error to zombie
   - Zombie must rejoin group to get current generation
4. Partition leaders get valid member list from coordinator periodically
```

10.3.5 ISR Shrinkage to Empty Set

Scenario: All followers for a partition fall behind the leader due to network issues, broker failures, or sustained high load. The ISR shrinks to just the leader, then the leader fails, leaving no in-sync replicas.

Progression:

```
Initial: ISR = [Leader L, Followers F1, F2]
Step 1: F1 network partition → removed from ISR after replica.lag.time.max.ms
Step 2: F2 disk slowdown → falls behind → removed from ISR
Step 3: ISR = [L] (single replica)
Step 4: L crashes → No in-sync replicas remain
```

Recovery Options:

1. **Wait for Recovery:** Don't elect new leader until at least one replica recovers and catches up
2. **Best-Effort Election:** Elect the replica with most recent data (highest LEO)
3. **Admin Intervention:** Manual override via tooling

Implementation Guidance:

- Track `lastCaughtUpTime` for each follower in `ISRManager`
- Configurable `unclean.leader.election.enable` per topic
- Log warnings when ISR size drops below `min.insync.replicas`
- Expose metrics for admin monitoring

10.3.6 Offset Commit Race During Rebalance

Scenario: Consumer commits offsets while rebalance is in progress. The commit might apply to old partition assignment, causing offsets to be associated with wrong partitions after reassignment.

Timeline:

```
T0: Consumer C1 has partitions [P1, P2]
T1: C1 commits offset for P2
T2: Rebalance starts (new consumer joins)
T3: Coordinator reassigns P2 to C2
T4: C1's offset commit arrives (delayed network)
T5: Offset for P2 stored under C1's member_id, but C2 now owns P2
```

Solution: Generation-aware Offset Commits:

- `CommitOffset` request includes `generation_id`
- Coordinator rejects commits with stale `generation_id`
- Consumers should commit offsets **before** rejoining group during rebalance
- Alternatively, commit offsets **after** receiving new assignment (Kafka's approach)

10.3.7 Producer Retry Causing Duplicate Sequence

Scenario: Idempotent producer sends batch with sequence number N, gets timeout, retries with same sequence number N, but first batch actually succeeded and was durably stored.

Without Proper Deduplication:

```
P1: Send batch (producer_id=100, epoch=1, seq=5) → Timeout
P2: Retry same batch (pid=100, epoch=1, seq=5) → Success
P3: First batch eventually succeeds (network delay) → Duplicate in log
```

Broker-side Deduplication:

- Maintain `lastSequence` per `(producer_id, producer_epoch, partition)`
- Reject batches with sequence ≤ `lastSequence`
- Accept batches with sequence = `lastSequence + 1`
- Gap in sequence numbers (sequence > `lastSequence + 1`) indicates lost messages → return error

10.3.8 Log Truncation During Follower Sync

Scenario: Leader truncates its log (e.g., after unclean leader election or admin operation) while follower is fetching. Follower may have already replicated data that no longer exists on leader.

Detection and Recovery:

1. Leader includes `log_start_offset` in `FetchResponse`
2. Follower compares its fetch offset with leader's `log_start_offset`
3. If `fetch_offset < log_start_offset`, follower must truncate its log
4. Follower rewinds to `log_start_offset` and re-fetches from there
5. Any local messages beyond truncation point are discarded

Implementation in `FollowerSyncer.AppendToLocalLog()`:

- Validate `leader_hw` and `log_start_offset` against local log
- If divergence detected, truncate local log to common point
- Log warning with offset details for admin review

10.4 Failure Recovery Summary Table

Failure Mode	Detection Method	Recovery Action	Consistency Impact	Availability Impact
Broker crash (leader)	Missed heartbeats, TCP failure	Elect new leader from ISR	None if ISR > 1	Brief unavailability during election
Broker crash (follower)	Missed heartbeats	Remove from ISR, replicate to other followers	Reduced durability margin	None
Network partition	Divergent metadata views, heartbeat failures	Fencing via epochs, wait for healing	Potential divergent writes if unclean election allowed	Partial availability loss
Disk full	`ENOSPC` on write	Reject writes, alert admin	Potential data loss if buffers overflow	Write unavailability for affected partitions
Consumer timeout	Missed heartbeats	Rebalance, redistribute partitions	At-least-once reprocessing possible	Brief consumption pause during rebalance
Producer retry storm	High error rate, timeout spikes	Exponential backoff, circuit breaker	Potential duplicates without idempotence	Increased latency, reduced throughput
Controller failure	Missed controller heartbeats	Elect new controller from brokers	Metadata update stall	No new leader elections until new controller
Zombie consumer	Generation ID mismatch in fetch	Reject fetches, force rejoin	Prevents duplicate processing	Brief disruption for zombie consumer

Implementation Guidance

> **Technology Note:** While production systems use sophisticated monitoring (Prometheus, health checks), our educational implementation can rely on simpler timeouts and periodic checks. The key is implementing the recovery logic correctly, not building enterprise monitoring.

A. Recommended Failure Detection Infrastructure

Simple Health Check Endpoint: Add to your `Server` a basic HTTP health endpoint (separate port) that checks:

- Disk space above threshold
- WAL write/read test
- Coordination service connectivity
- Memory usage

Timeout Configuration Table:

Configuration	Default Value	Purpose	Override Guidance
`session.timeout.ms`	10000 (10s)	Consumer heartbeat timeout	Increase for GC-heavy environments
`replica.lag.time.max.ms`	30000 (30s)	Max time follower can lag before ISR removal	Adjust based on network reliability
`request.timeout.ms`	30000 (30s)	Client request timeout	Match expected network latency
`log.flush.timeout.ms`	1000 (1s)	Max time for log fsync	Tune based on disk performance

B. Core Recovery Logic Skeletons

Leader Election on Broker Failure:

GO

```go
// In internal/coordinator/leader_elector.go

// OnBrokerFailure triggers leader election for all partitions where the failed
// broker was the leader. Called by coordination service when broker marked dead.
func (e *LeaderElector) OnBrokerFailure(failedBrokerID int32) error {
    // TODO 1: Query coordination service for all partitions where
    //         failedBrokerID == PartitionMetadata.LeaderID

    // TODO 2: For each affected partition:
    //   a. Get current ISR from PartitionMetadata
    //   b. Remove failedBrokerID from ISR slice
    //   c. If ISR is empty:
    //        - Log critical warning "No in-sync replicas for partition"
    //        - Skip election (partition unavailable) OR
    //        - If unclean election enabled, use all replicas as candidates

    // TODO 3: Select new leader using deterministic algorithm:
    //   - Prefer replicas in original ISR order (oldest first)
    //   - Ensure new leader is alive (check broker heartbeats)

    // TODO 4: Update PartitionMetadata with new LeaderID and ISR

    // TODO 5: Notify all brokers of metadata change via coordination service

    // TODO 6: Return any errors encountered during the process
    return nil
}

// selectNewLeader chooses a replica from ISR to become the new leader.
// Returns -1 if no suitable leader found (ISR empty).
func (e *LeaderElector) selectNewLeader(isr []int32, partitionInfo PartitionMetadata) int32 {
    // TODO 1: Filter ISR to only alive brokers (check broker heartbeats)

    // TODO 2: If filtered ISR empty, return -1 (partition unavailable)
```

```
    // TODO 3: Apply deterministic selection:
    //   - Sort alive ISR members by broker ID
    //   - Choose first in sorted list (simplest strategy)
    //   - Alternative: choose replica with highest LEO (requires extra metadata)

    // TODO 4: Return selected broker ID
    return -1
}
```

Consumer Zombie Detection in Fetch Handler:

```go
// In internal/broker/fetch_handler.go


// validateConsumerGeneration checks if consumer is part of current generation.

// Returns ErrUnknownMember if consumer is zombie (stale generation).

func (h *FetchHandler) validateConsumerGeneration(

    groupID string,

    memberID string,

    generationID int32,

) error {

    // TODO 1: Get group from coordinator using groupID



    // TODO 2: If group not found, return ErrUnknownMemberId



    // TODO 3: Check if memberID exists in group.Members map



    // TODO 4: Compare generationID with group.GenerationID



    // TODO 5: If generationID < group.GenerationID, consumer is zombie

    //         Return ErrUnknownMemberId to force rejoin



    // TODO 6: If member not in current members list, return ErrUnknownMemberId



    return nil

}


// HandleFetch now includes validation:

func (h *FetchHandler) HandleFetch(req *FetchRequest) (*FetchResponse, error) {

    // TODO: For consumer fetch requests (ReplicaID = -1):

    //   Call validateConsumerGeneration for each unique (group, member)

    //   Reject entire request if any consumer is zombie



    // ... rest of fetch logic

}
```

Log Recovery on Broker Startup:

```go
// In internal/log/log_manager.go


// RecoverLogs scans all log directories and recovers from partial writes.

// Called during broker startup before accepting client requests.

func (lm *LogManager) RecoverLogs() error {

    // TODO 1: Walk data directory to find all topic-partition directories



    // TODO 2: For each partition directory:

    //   a. List segment files sorted by base offset

    //   b. For each segment file:

    //        i. Open data file and index file

    //        ii. Read from start, validating each RecordBatch CRC

    //        iii. When invalid batch found (CRC mismatch or length wrong):

    //              - Truncate file at last valid batch boundary

    //              - Rebuild index from remaining valid data

    //              - Log warning with truncation offset

    //        iv. Update Log.Segments list and Log.CurrentOffset



    // TODO 3: For replicated partitions, compare with leader's high watermark

    //         Truncate beyond high watermark if follower is ahead



    // TODO 4: Return any unrecoverable errors (e.g., disk errors)

    return nil

}
```

Network Partition Detection via Dual Heartbeats:

```go
// In internal/coordinator/memory_coordinator.go


// checkBrokerHealth performs dual health checking to detect network partitions.

func (mc *MemoryCoordinator) checkBrokerHealth() {

    // TODO 1: For each registered broker:

    //   a. Check last heartbeat time (coordination plane)

    //   b. Check last metadata request time (data plane proxy)



    // TODO 2: If heartbeat stale but metadata recent:

    //   - Log warning "Possible network partition for broker X"

    //   - Don't mark as dead immediately

    //   - Increase suspicion level



    // TODO 3: If both stale beyond thresholds:

    //   - Mark broker as dead

    //   - Trigger leader election for affected partitions



    // TODO 4: If heartbeat recent but metadata stale:

    //   - Broker may be overloaded or have disk issues

    //   - Consider removing from ISR for hosted partitions

}
```

C. Debugging Tips for Common Failure Scenarios

Symptom	Likely Cause	Diagnostic Steps	Fix
Messages not consumed after consumer restart	Offset commit failed during shutdown	1. Check offset store for group 2. Verify last commit timestamp 3. Check consumer logs for commit errors	Ensure `CommitSync()` called before `Close()`, increase `request.timeout.ms`
Producer hangs indefinitely	All replicas down for target partition	1. Check partition ISR size 2. Verify broker health 3. Check producer logs for `ErrNotLeaderForPartition`	Configure `retries` and `retry.backoff.ms`, monitor broker health
Consumer group stuck in rebalance	One consumer not responding to JoinGroup	1. Check coordinator logs for timeout members 2. Verify all consumers can reach coordinator 3. Check for clock skew between machines	Increase `session.timeout.ms`, synchronize NTP, ensure network connectivity
High watermark not advancing	Followers not catching up to leader	1. Check follower fetch logs 2. Verify network between brokers 3. Check disk I/O on followers	Increase `replica.fetch.wait.ms`, check network links, monitor disk performance
Duplicate messages consumed	Consumer processed messages, crashed before commit	1. Check offset lag before/after crash 2. Verify commit frequency settings 3. Check for zombie consumers	Decrease `auto.commit.interval.ms`, implement synchronous commits, add generation fencing
Log segment corruption	Disk error or improper shutdown	1. Check CRC validation logs 2. Verify disk SMART status 3. Check for OOM kill during write	Implement segment CRC validation, add disk monitoring, ensure proper shutdown hooks

D. Testing Failure Recovery

Integration Test Skeleton for Broker Failure:

```go
// In tests/integration/broker_failure_test.go

func TestLeaderElectionOnBrokerFailure(t *testing.T) {

    // TODO 1: Start 3 brokers in test cluster


    // TODO 2: Create topic with replication factor 3, partitions 1


    // TODO 3: Produce messages to establish leadership


    // TODO 4: Kill leader broker (simulate crash)


    // TODO 5: Wait for election timeout


    // TODO 6: Verify new leader elected from ISR


    // TODO 7: Produce more messages to new leader


    // TODO 8: Verify all messages preserved (no data loss)


    // TODO 9: Restart dead broker


    // TODO 10: Verify it rejoins as follower and catches up

}
```

Property-based Test for Idempotent Producer:

```go
// Use gopter for property testing
func TestProducerIdempotence(t *testing.T) {
    parameters := gopter.DefaultTestParameters()
    properties := gopter.NewProperties(parameters)

    properties.Property("retried batches produce no duplicates", prop.ForAll(
        func(initialSeq int32) bool {
            // TODO: Implement test that:
            // 1. Sends batch with sequence N
            // 2. Simulates network timeout
            // 3. Retries same batch
            // 4. Verifies only one copy in log
            return true
        },
        gen.Int32Range(1, 1000),
    ))

    properties.TestingRun(t)
}
```

> **Key Learning Point:** The most robust systems aren't those that never fail, but those that fail in predictable, recoverable ways. Your implementation should prioritize clear failure modes and recovery paths over trying to prevent all failures.

11. Testing Strategy

> **Milestone(s):** 1, 2, 3, 4 (essential for validating all system behaviors)

Building a distributed message queue is an exercise in managing complexity. The system's correctness depends on subtle interactions between concurrent components, network communication, and persistent storage. A comprehensive testing strategy is therefore not merely a validation step—it is a fundamental design tool that helps you reason about the system's behavior, catch bugs early, and build confidence in the implementation. This section provides a structured approach to testing, from unit tests of individual algorithms to integration tests that simulate real-world cluster operations.

11.1 Testing Approach

Testing a distributed system is akin to stress-testing a suspension bridge. You must first validate each cable and beam in isolation (unit tests), then assemble them into spans and verify their connections (integration tests), and finally subject the entire structure to extreme loads and unexpected forces (property-based and fault injection tests) to ensure it won't collapse under real-world conditions.

Our testing strategy employs three complementary layers, each targeting different levels of abstraction and failure modes:

1. Unit Tests: The Foundation of Logic Verification Unit tests focus on isolated components—individual functions, methods, and data structures—in a controlled environment. Their primary goal is to validate algorithmic correctness and internal state transitions without the complexity of network or disk I/O. For example, testing that the `HashPartitioner.Partition` function consistently routes the same key to the same partition, or that the `Index.FindEntry` binary search returns the correct file position. These tests should be fast, deterministic, and run without external dependencies.

> **Key Insight:** Mock all I/O and external dependencies in unit tests. Use dependency injection to replace network clients, file systems, and random number generators with deterministic test doubles. This ensures tests remain reliable and fast, even as the system grows.

2. Integration Tests: Validating Component Interactions Integration tests assemble multiple components and verify they work together correctly. This layer is crucial for our system because the core value emerges from interactions—between producers and brokers, consumers and coordinators, and leaders and followers. We recommend two types of integration tests:

- **In-process cluster tests:** Launch multiple `Server` instances within the same process, connected via loopback network interfaces. This allows testing full cluster behaviors (rebalancing, leader election) without the overhead of process spawning.
- **Component integration tests:** Test specific interfaces between components, such as a `Producer` sending to a real `Server` over TCP, or a `ConsumerGroup` interacting with a `Coordinator`.

These tests should use real network sockets and file I/O, but run within a controlled test environment (e.g., temporary directories, deterministic port assignment).

3. Property-Based Tests: Exploring Edge Cases Systematically Property-based testing (PBT) is a powerful technique for uncovering edge cases that example-based tests might miss. Instead of writing specific test cases (e.g., "send 3 messages"), you define properties that should *always* hold true for any valid input, and a test framework generates thousands of random inputs to verify them. For our system, important properties include:

- **Idempotence:** Sending the same batch of records twice (with the same producer ID and sequence) should result in exactly one set of records in the log.
- **Ordering within partitions:** For any sequence of messages sent to the same partition, the offsets returned must be strictly increasing and contiguous when no failures occur.
- **No data loss:** The concatenation of all messages read by a consumer (after accounting for commits) must equal the concatenation of all messages sent by producers.

PBT frameworks like gopter for Go can generate complex scenarios: random network partitions, broker crashes, and concurrent client operations.

ADR: Testing Strategy Layering

Decision: Three-Layer Testing Pyramid with Heavy Integration Focus

- **Context:** Our educational system must validate both algorithmic correctness (unit tests) and complex distributed behaviors (integration). Learners need immediate feedback on logic errors, but also confidence that components work together.
- **Options Considered:**
  1. **Unit-heavy pyramid:** Traditional approach with 70% unit, 20% integration, 10% end-to-end tests. Fast but misses interaction bugs.
  2. **Integration-heavy "hourglass":** Equal emphasis on unit and end-to-end, with integration as the thick middle layer. Slower but catches more distributed bugs.
  3. **Property-based emphasis:** Rely primarily on generated tests to explore state space. Powerful but requires significant upfront investment and expertise.
- **Decision:** Adopt an integration-heavy hourglass model, supplemented by property-based tests for core invariants.
- **Rationale:** The greatest learning value and most subtle bugs in distributed systems arise from component interactions. Integration tests provide concrete, observable behaviors that learners can debug. Property-based tests complement by systematically exploring edge cases that manual test design might miss.
- **Consequences:** Test suite will run slower than unit-only approaches, requiring careful use of test parallelism and short timeouts. However, it will provide higher confidence in system correctness and better prepare learners for real-world debugging.

Option	Pros	Cons	Chosen?
Unit-heavy pyramid	Fast execution, clear isolation, easy to debug	Misses interaction bugs, gives false confidence	No
Integration-heavy hourglass	Catches distributed bugs, validates real workflows	Slower, more complex test setup	**Yes**
Property-based emphasis	Explores vast state space, finds obscure edge cases	Steep learning curve, harder to debug failures	Supplementary

Common Pitfalls in Testing Distributed Systems

⚠️ **Pitfall: Non-deterministic tests due to timing and concurrency** Tests that rely on real-time delays (`time.Sleep`) or uncoordinated goroutines will fail intermittently, eroding trust in the test suite.

- **Why it's wrong:** Flaky tests mask real bugs and waste debugging time. They often pass in development but fail in CI.
- **How to fix:** Use synchronized channels or condition variables to signal when events occur. For time-based operations, inject a mock clock that can be advanced manually in tests.

⚠️ **Pitfall: Not cleaning up resources between tests** Leaving open files, network sockets, or goroutines from one test can affect subsequent tests.

- **Why it's wrong:** Tests become order-dependent and may fail when run in isolation vs. as a suite.
- **How to fix:** Use `t.Cleanup()` or `defer` to close resources. For goroutines, pass a `context.Context` that gets cancelled when the test ends.

⚠️ **Pitfall: Over-mocking eliminates integration value** Mocking every dependency, including the local log storage or in-memory maps, reduces integration tests to unit tests of the mock setup.

- **Why it's wrong:** You're not testing the actual interaction between components, just your assumptions about how they *should* interact.
- **How to fix:** Use real implementations for internal components (like the `WAL` or `Log`), but mock only external boundaries (network between separate processes, disk I/O if too slow).

11.2 Milestone Verification Checkpoints

Each milestone in the project builds upon the previous one, creating a cumulative verification challenge. The following checkpoints provide concrete, executable scenarios you can use to validate that your implementation meets the core requirements. These are not exhaustive tests, but they represent critical paths that, if working correctly, indicate fundamental correctness.

Checkpoint Structure: For each milestone, we define:

1. **Test Scenario:** A narrative description of the operation to test.
2. **Preconditions:** State of the system before the test.
3. **Steps:** Concrete actions to perform (manually or via test code).
4. **Expected Output:** What should happen after each step.
5. **Verification Method:** How to observe and confirm the expected behavior.

Milestone 1: Topic and Partitions

Aspect	Details
Test Scenario	Create a topic with multiple partitions, produce messages with and without keys, and verify consistent partitioning and offset ordering.
Preconditions	Single broker running with clean data directory.
Steps	1. Create topic "orders" with 3 partitions. 2. Produce 5 messages: two with key="user1", two with key="user2", one with null key. 3. Fetch metadata to see partition assignments. 4. Read messages from each partition individually.
Expected Output	- Topic creation succeeds. - Messages with same key land in same partition. - Offsets within each partition are sequential (0,1,...). - Null-key message distributes via round-robin. - `LogEndOffset` for each partition matches message count.
Verification Method	Inspect broker logs or use a debug API to dump partition contents. Programmatically verify offsets and key-to-partition mapping.

Milestone 2: Producer

Aspect	Details
Test Scenario	Producer with `acks=all` sends batched messages, survives a leader failover, and retries without duplicates (idempotence optional).
Preconditions	3-node cluster with topic "logs" (replication factor 3). Producer configured with batch size=2, linger=100ms.
Steps	1. Send 10 messages asynchronously with callbacks. 2. Kill the partition leader while messages are in-flight. 3. Wait for new leader election. 4. Call `Flush()` and verify all callbacks fired with success.
Expected Output	- All 10 messages are eventually acknowledged. - No duplicate messages appear in the log (check offsets). - Producer logs show retry attempts after leader failure. - High watermark advances only after all ISRs acknowledge.
Verification Method	Compare sent message count with total records in log segments. Check producer callback invocation count.

Milestone 3: Consumer Groups

Aspect	Details
Test Scenario	Two consumers join a group, receive balanced partition assignments, commit offsets, and trigger rebalance when one leaves.
Preconditions	Topic "events" with 4 partitions, containing 100 pre-existing messages.
Steps	1. Start Consumer A in group "grp1", subscribe to "events". 2. Start Consumer B in same group, subscribe to "events". 3. Let each consumer poll 20 messages and commit offsets. 4. Stop Consumer B gracefully. 5. Observe Consumer A's partition reassignment.
Expected Output	- Initial assignment: each consumer gets 2 partitions (balanced). - Each consumer reads only from its assigned partitions. - Offset commits are persisted and survive consumer restart. - After B leaves, A gets all 4 partitions within session timeout. - No messages are processed twice (thanks to committed offsets).
Verification Method	Check coordinator's assignment state. Verify committed offsets in `__consumer_offsets` internal topic. Count total unique messages processed.

Milestone 4: Replication

Aspect	Details
Test Scenario	Leader-follower replication maintains consistency during network partition and recovers via ISR management.
Preconditions	3-node cluster, topic "audit" with RF=3, all replicas in sync.
Steps	1. Produce 50 messages with `acks=all`. Verify all replicas have same LEO. 2. Isolate follower 2 from leader (simulate network partition). 3. Produce 20 more messages. 4. Wait for replica.lag.time.max.ms to expire. 5. Check ISR size (should shrink to 2). 6. Heal network partition, verify follower catches up and rejoins ISR.
Expected Output	- First 50 messages replicated to all followers. - ISR shrinks after follower 2 lags beyond threshold. - Messages 51-70 are only replicated to remaining ISR members. - After healing, follower 2 fetches missing messages and rejoins ISR. - High watermark never exceeds LEO of slowest ISR member.
Verification Method	Query each replica's log end offset. Monitor ISR changes via metadata. Verify consumer reads only up to high watermark.

> **Design Insight:** These checkpoints intentionally stress the "happy path" *and* failure scenarios. A system that works only when nothing goes wrong is useless in production. Each milestone's checkpoint includes at least one injected failure (leader kill, consumer stop, network partition) to validate recovery logic.

11.3 Tools and Libraries

The Go ecosystem provides excellent testing libraries that align with our layered approach. The following recommendations balance simplicity for learners with power for complex scenarios.

Core Testing Framework: `testing` + `testify` Go's built-in `testing` package is sufficient for most needs, but `testify` adds valuable assertions and mocking capabilities.

Library	Purpose	Key Features	Example Use
`testing` (standard)	Basic test framework	`t.Run()` for subtests, `t.Helper()`, `t.Cleanup()`	All unit and integration tests
`testify/assert`	Readable assertions	`assert.Equal()`, `assert.NoError()`, helpful failure messages	Replacing `if err != nil { t.Fatal() }` patterns
`testify/require`	Assertions that stop test	`require.True()` stops test immediately on failure	Precondition validation in tests
`testify/suite`	Test suite organization	`SetupSuite()`, `TearDownTest()` methods	Structuring integration test lifecycle
`testify/mock`	Mock object generation	Generate mocks for interfaces, verify calls	Mocking `ConnectionPool` in producer tests

Mocking and Dependency Injection For unit testing components in isolation, generate mocks for their dependencies. The `testify/mock` package works well, but consider interface design that facilitates testing:

```go
// Example: Interface for network connection to allow mocking
type Connection interface {
    Send(request []byte) (response []byte, err error)
    Close() error
}

// Real implementation
type TCPConnection struct { /* ... */ }

// Mock implementation
type MockConnection struct {
    mock.Mock
}
func (m *MockConnection) Send(request []byte) ([]byte, error) {
    args := m.Called(request)
    return args.Get(0).([]byte), args.Error(1)
}
```

Property-Based Testing: `gopter` `gopter` (GO Property Tester) generates random inputs and shrinks failing cases to minimal examples.

Feature	Benefit for Our System
Arbitrary value generation	Create random `Record` objects with varied keys, values, headers
Stateful command testing	Model producer/consumer interactions as state transitions
Shrinking	Reduce failing 1000-message test to 3-message minimal case

Concurrency and Race Detection Go's race detector (`go test -race`) is essential for catching data races in concurrent code. Additionally:

Pattern	Implementation	Purpose
Synchronized test verification	Use channels to collect results from goroutines	Verify concurrent producer sends
Context cancellation	`context.WithTimeout` in tests	Prevent hanging tests on deadlock
WaitGroups	`sync.WaitGroup` to await goroutine completion	Ensure all test goroutines finish

Network Testing Utilities For integration tests that require real network communication:

Utility	Purpose	Example
`net.Listen("tcp", "localhost:0")`	Get free port for test servers	Start broker on random port
`httptest.Server` (if using HTTP)	In-memory HTTP server for client tests	Not used in our binary protocol
`io.Pipe()`	Connect reader/writer without sockets	Test protocol encoding/decoding

Temporary Resources Management Always use temporary directories for test data to avoid collisions and ensure cleanup:

```go
func TestBroker_RecoversLogs(t *testing.T) {
    tempDir := t.TempDir() // Automatically cleaned up after test
    broker := NewBroker(tempDir)
    // ... test logic
}
```

Integration Test Helper Pattern Create a test helper that spins up an in-process cluster:

```go
type TestCluster struct {
    Brokers []*Server
    TempDirs []string
}

func NewTestCluster(t *testing.T, size int) *TestCluster {
    // Initialize 'size' brokers with temporary directories
    // Set up inter-broker communication via loopback
    // Return cluster handle with cleanup via t.Cleanup()
}
```

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Test Framework	`testing` + `testify`	`ginkgo` + `gomega` (BDD style)
Mocking	`testify/mock` (manual)	`gomock` (code generation)
Property Testing	Manual random generation	`gopter` with custom generators
Concurrency Testing	`go test -race`, channels	`stress` test with goroutine profiling
Network Testing	Loopback TCP with real sockets	`netem` for network fault injection
Temporary Storage	`t.TempDir()`	RAM disk for faster I/O

B. Recommended File/Module Structure for Tests

```
project-root/
  internal/
    broker/
      broker.go
      broker_test.go           # Unit tests for broker logic
    log/
      log.go
      log_test.go              # Unit tests for log operations
    protocol/
      encode_decode.go
      encode_decode_test.go    # Property tests for wire format
    coordinator/
      coordinator.go
      coordinator_integration_test.go  # Integration with real network
  test/
    integration/
      cluster_test.go          # Multi-broker integration tests
      producer_consumer_test.go # End-to-end workflows
    property/
      ordering_property_test.go # PBT for ordering guarantees
    helpers/
      test_cluster.go          # TestCluster helper
      test_client.go           # Test producer/consumer clients
  cmd/
    server/
      main.go
    server_integration_test.go # Integration test for main binary
```

C. Infrastructure Starter Code: Test Helper Utilities

```go
// test/helpers/test_cluster.go

package helpers


import (

    "context"

    "fmt"

    "net"

    "path/filepath"

    "sync"

    "testing"

    "time"


    "github.com/stretchr/testify/require"

)


// TestCluster manages an in-process broker cluster for integration tests.

type TestCluster struct {

    t       *testing.T

    size    int

    brokers []*Server

    tempDirs []string

    mu      sync.RWMutex

}


// NewTestCluster creates a cluster of 'size' brokers.

func NewTestCluster(t *testing.T, size int) *TestCluster {

    require.Greater(t, size, 0, "cluster size must be positive")



    tc := &TestCluster{

        t:      t,

        size:   size,

        brokers: make([]*Server, size),

        tempDirs: make([]string, size),

    }
```

```
// Create coordination service (in-memory for tests)
coord := NewMemoryCoordinator()

// Initialize each broker
for i := 0; i < size; i++ {
    tempDir := t.TempDir()
    tc.tempDirs[i] = tempDir

    // Find free port
    listener, err := net.Listen("tcp", "localhost:0")
    require.NoError(t, err)
    addr := listener.Addr().(*net.TCPAddr)
    listener.Close()

    config := Config{
        Host:   "localhost",
        Port:   addr.Port,
        DataDir: tempDir,
        BrokerID: int32(i + 1),
    }

    broker, err := NewServer(config, coord)
    require.NoError(t, err)

    tc.brokers[i] = broker

    // Start broker in background
    go func(b *Server) {
        if err := b.Start(); err != nil {
            t.Logf("broker %d stopped: %v", i, err)
        }
    }(broker)
```

```
        // Register broker with coordination service
        err = coord.RegisterBroker(context.Background(), &Broker{
            ID:   int32(i + 1),
            Host: "localhost",
            Port: int32(addr.Port),
        })
        require.NoError(t, err)
    }

    // Wait for brokers to be ready
    tc.waitForBrokersReady()

    // Cleanup on test completion
    t.Cleanup(func() {
        tc.Shutdown()
    })

    return tc
}

// waitForBrokersReady polls each broker until it responds to metadata requests.
func (tc *TestCluster) waitForBrokersReady() {
    deadline := time.Now().Add(10 * time.Second)
    for i, broker := range tc.brokers {
        for time.Now().Before(deadline) {
            // Try to connect to broker's admin port (simplified)
            conn, err := net.DialTimeout("tcp",
                fmt.Sprintf("%s:%d", broker.config.Host, broker.config.Port),
                100*time.Millisecond)
            if err == nil {
                conn.Close()
                break
            }
            time.Sleep(100 * time.Millisecond)
```

```
        }
        require.True(tc.t, time.Now().Before(deadline),
            "broker %d failed to start within timeout", i)
    }
}

// Shutdown gracefully stops all brokers.
func (tc *TestCluster) Shutdown() {
    tc.mu.Lock()
    defer tc.mu.Unlock()

    for _, broker := range tc.brokers {
        if broker != nil {
            broker.Stop() // Assumes Stop() method exists
        }
    }
}

// GetBroker returns the broker at index i (0-based).
func (tc *TestCluster) GetBroker(i int) *Server {
    tc.mu.RLock()
    defer tc.mu.RUnlock()
    require.Less(tc.t, i, tc.size, "broker index out of range")
    return tc.brokers[i]
}

// CreateTopic creates a topic on the cluster with default settings.
func (tc *TestCluster) CreateTopic(topic string, partitions, replicationFactor int) error {
    // Implementation creates topic metadata via coordinator
    // TODO: Implement using coordinator API
    return nil
}
```

D. Core Logic Skeleton Code: Property-Based Test Example

```go
// test/property/ordering_property_test.go

package property

import (
    "testing"


    "github.com/leanovate/gopter"
    "github.com/leanovate/gopter/gen"
    "github.com/leanovate/gopter/prop"
)

// TestPartitionOrderingProperty verifies that offsets within a partition
// are always strictly increasing and contiguous when no failures occur.
func TestPartitionOrderingProperty(t *testing.T) {
    parameters := gopter.DefaultTestParameters()
    parameters.MinSuccessfulTests = 1000
    parameters.MaxSize = 100 // Max messages per test run

    properties := gopter.NewProperties(parameters)

    // Generator for a slice of records with random keys/values
    recordsGen := gen.SliceOf(genRecord())

    properties.Property("offsets are strictly increasing within partition",
        prop.ForAll(func(records []*Record) bool {
            // TODO 1: Create a fresh in-memory Log for testing
            // TODO 2: Append all records using Log.Append
            // TODO 3: Read back records from offset 0
            // TODO 4: Verify each record's offset increases by exactly 1
            // TODO 5: Verify the sequence of records matches input sequence
            return true // placeholder
        }, recordsGen))

    properties.TestingRun(t)
```

```
}

// genRecord generates a random Record for property testing.
func genRecord() gopter.Gen {
    return gen.Struct(reflect.TypeOf(&Record{}), map[string]gopter.Gen{
        "Key":   gen.SliceOf(gen.UInt8()).SuchThat(func(v interface{}) bool { return len(v.([]byte)) <= 1024 }),
        "Value": gen.SliceOf(gen.UInt8()).SuchThat(func(v interface{}) bool { return len(v.([]byte)) <= 1024 }),
        "Headers": gen.SliceOf(genHeader()),
    })
}

// genHeader generates a random Header.
func genHeader() gopter.Gen {
    return gen.Struct(reflect.TypeOf(Header{}), map[string]gopter.Gen{
        "Key":   gen.AlphaString(),
        "Value": gen.SliceOf(gen.UInt8()),
    })
}
```

E. Language-Specific Hints for Go Testing

1. **Table-Driven Tests:** Use this pattern extensively for unit tests with multiple input cases:

```go
func TestHashPartitioner(t *testing.T) {
    tests := []struct{
        name string
        key []byte
        numPartitions int32
        want int32
    }{
        {"nil key", nil, 5, 0},
        {"key 'abc'", []byte("abc"), 3, 1},
    }
    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            p := &HashPartitioner{}
            got, err := p.Partition("test", tt.key, tt.numPartitions)
            require.NoError(t, err)
            require.Equal(t, tt.want, got)
        })
    }
}
```

2. **Test Parallelization:** Use `t.Parallel()` in independent tests to speed up execution, but avoid it for tests that share resources (like a global test cluster).
3. **Golden Files:** For protocol serialization tests, consider using golden files to compare expected binary output:

```go
golden := filepath.Join("testdata", "produce_request_v2.golden")
if *update {
    os.WriteFile(golden, encoded, 0644)
}
expected, _ := os.ReadFile(golden)
require.Equal(t, expected, encoded)
```

F. Milestone Checkpoint Verification

After implementing each milestone, run the corresponding verification test:

```bash
# Milestone 1: Topic and Partition basics
go test ./internal/broker -run TestBroker_CreateTopicAndProduce -v

# Expected output:
# === RUN   TestBroker_CreateTopicAndProduce
# --- PASS: TestBroker_CreateTopicAndProduce (0.2s)
# PASS

# Milestone 2: Producer with batching and retries
go test ./internal/producer -run TestProducer_SendWithLeaderFailover -v

# Milestone 3: Consumer group rebalancing
go test ./internal/coordinator -run TestConsumerGroup_Rebalance -v

# Milestone 4: Replication and ISR management
go test ./internal/replication -run TestReplication_ISRShrinkAndRecover -v
```

G. Debugging Tips for Test Failures

Symptom	Likely Cause	How to Diagnose	Fix
Test hangs indefinitely	Deadlock in goroutines, missing channel receive	Use `pprof` goroutine dump, add test timeouts with `context.WithTimeout`	Ensure all goroutines have cancellation paths, use `select` with `ctx.Done()`
Intermittent test failures	Race condition, timing dependency	Run with `go test -race`, add `t.Log` statements to trace execution order	Add proper synchronization (mutexes, channels), make tests deterministic
"Address already in use"	Previous test didn't clean up sockets	Check for proper `t.Cleanup()` or `defer server.Stop()`	Ensure all servers are stopped, use random ports with `:0`
Disk space errors in CI	Temporary files not cleaned up	Check `t.TempDir()` usage, look for leftover files	Always use `t.TempDir()`, not manual `/tmp` creation
Consumer group rebalance storms	Heartbeat intervals too short, session timeout too low	Log rebalance triggers, increase test timeouts	Adjust `session.timeout.ms` and `heartbeat.interval.ms` in test config

> **Final Testing Principle:** Write tests that fail in the way you expect. If a bug is fixed, add a test that would have caught it. Tests are not just verification—they are executable documentation of your system's intended behavior.

12. Debugging Guide

> **Milestone(s):** 1, 2, 3, 4 (Debugging is relevant to all milestones)

Building a distributed message queue is a complex endeavor with many moving parts. Even with careful design and implementation, bugs are inevitable. This debugging guide provides a systematic approach to diagnosing and fixing common issues that arise during

development. Think of debugging a distributed system like being a **mechanical engineer troubleshooting a multi-engine aircraft**: you need to check each subsystem (engines, fuel lines, electrical) individually while also understanding how they interact, using both instrument readings (logs) and test procedures (diagnosis techniques) to isolate the root cause.

12.1 Common Bug Symptoms

Distributed system bugs manifest in observable ways that affect either correctness (wrong results) or performance (slow results). Learning to recognize these symptoms is the first step toward diagnosis.

1. **Messages Lost (Missing Data)**
   - *Description*: Messages sent by producers never appear when consumers fetch, or only some messages are delivered. This violates the at-least-once delivery guarantee and indicates data loss.
   - *Example Scenario*: A producer sends 100 messages with `acks=all`, but a consumer reading from the beginning only receives 85 messages, with no apparent pattern to which ones are missing.
2. **Duplicate Messages (Extra Data)**
   - *Description*: The same message appears multiple times in consumer output, either immediately or after restart. This violates at-most-once semantics and can cause incorrect application behavior.
   - *Example Scenario*: After a consumer crashes and restarts, it reprocesses messages it had already processed before the crash, even though it had committed offsets.
3. **Consumers Stuck (No Progress)**
   - *Description*: One or more consumers in a group stop fetching new messages despite producers continuing to send data. The consumer appears "frozen" at a particular offset.
   - *Example Scenario*: In a three-consumer group, two consumers continue processing messages while the third remains stuck at offset 42, never advancing despite new messages being available in its assigned partitions.
4. **High Latency (Slow Performance)**
   - *Description*: Message delivery takes significantly longer than expected, with producers experiencing slow acknowledgment times or consumers experiencing long delays between message batches.
   - *Example Scenario*: A producer configured with 10ms `linger.ms` actually waits 500ms before sending batches, or a consumer calling `Poll()` takes seconds to return even when messages are available.
5. **Rebalance Storms (Frequent Churn)**
   - *Description*: Consumer groups undergo constant rebalancing, with partitions being reassigned frequently even without consumers joining or leaving. This wastes resources and disrupts processing.
   - *Example Scenario*: A stable consumer group suddenly starts rebalancing every 2-3 seconds, causing repeated pauses in message consumption without any apparent reason.
6. **Inconsistent Ordering (Sequence Violation)**
   - *Description*: Messages with the same key appear in different order when consumed, violating the per-key ordering guarantee within a partition.
   - *Example Scenario*: A producer sends messages with keys A, B, A (in that order), but the consumer receives them as A, A, B, even though both A messages went to the same partition.
7. **ISR Shrinkage to Empty (No Replicas Available)**
   - *Description*: The In-Sync Replica set for a partition becomes empty, making the partition unavailable for writes or unsafe reads, often after a series of failures.
   - *Example Scenario*: After a network partition isolates the leader, all followers are removed from ISR, and when the leader fails, no eligible replica exists to become the new leader.
8. **Memory Leak (Growing Resource Usage)**

- *Description*: Broker or client memory usage grows continuously over time without stabilizing, eventually leading to out-of-memory crashes.
- *Example Scenario*: A broker's RSS memory increases by 10MB per hour even with constant message volume, eventually crashing after 24 hours.

12.2 Diagnosis Techniques

Effective debugging requires systematic observation and hypothesis testing. These techniques provide the "instruments" to understand what's happening inside your system.

Structured Logging with Context

> **Mental Model**: Think of structured logs as a **flight data recorder** (black box) for your distributed system. Each component continuously records its state, decisions, and interactions with timestamps, allowing you to reconstruct events leading to a failure.

- **Implementation Approach**: Instead of `fmt.Printf`, use a structured logging library that supports:
  - **Levels**: DEBUG (internal state), INFO (normal operations), WARN (potential issues), ERROR (failures)
  - **Fields**: Key-value pairs attached to each log message (e.g., `partition=3`, `offset=142`, `consumer_id="c1"`)
  - **Correlation IDs**: Unique identifiers passed through request chains to trace messages across components
- **Critical Log Points to Add**:
  - *Message Flow*: Log when messages are appended to log (with offset), when batches are sent/received, when consumers fetch messages
  - *State Changes*: Log partition leadership changes, ISR membership updates, consumer group state transitions
  - *Timing*: Log request durations, batch accumulation times, network round-trip times
  - *Errors*: Log all errors with full context (what operation failed, with what parameters, what error code)
- **Example Log Configuration**:

```go
// In initialization
log.SetLevel(log.DebugLevel)
log.SetFormatter(&log.JSONFormatter{})

// In broker append logic
log.WithFields(log.Fields{
    "topic": topic,
    "partition": partitionID,
    "offset": newOffset,
    "batch_size": len(batch.Records),
    "leader_epoch": leaderEpoch,
}).Info("Appended records to partition")
```

Internal State Inspection via Admin APIs

> **Mental Model**: Think of state inspection as a **submarine's control panel** with gauges for each subsystem. By exposing internal metrics and state through diagnostic endpoints, you can check "vital signs" without stopping the system.

- **What to Expose**:
  - *Broker Metrics*: Memory usage, open file descriptors, Goroutine count, request queue depths
  - *Partition State*: Log end offset, high watermark, ISR members, leader epoch, follower lag
  - *Consumer Group State*: Member list, assigned partitions, last heartbeat time, generation ID
  - *Producer State*: Batch accumulator sizes, in-flight request counts, retry queue depth
- **Implementation Approach**: Create a simple HTTP endpoint (`/debug/state`) that returns JSON representations of key data structures. Consider thread-safe snapshot approaches:

```go
// Example: Broker state snapshot
type BrokerDebugState struct {
    Topics     map[string]TopicDebugState `json:"topics"`
    Goroutines int                        `json:"goroutines"`
    MemoryMB   float64                    `json:"memory_mb"`
    Uptime     string                     `json:"uptime"`
}

// Called from debug endpoint handler
func (b *Broker) GetDebugState() BrokerDebugState {
    b.mu.RLock()
    defer b.mu.RUnlock()

    state := BrokerDebugState{
        Topics: make(map[string]TopicDebugState),
        Goroutines: runtime.NumGoroutine(),
        MemoryMB: getMemoryUsageMB(),
    }

    for name, topic := range b.topics {
        state.Topics[name] = topic.GetDebugState()
    }
    return state
}
```

Timeout-Based Bottleneck Identification

> **Mental Model**: Think of timeouts as **circuit breakers** in an electrical system. When a component takes too long to respond, the timeout "trips" and allows the system to continue with degraded functionality, while also signaling where bottlenecks exist.

- **Strategic Timeout Placement**:
  1. **Network Timeouts**: Set deadlines on all socket reads/writes (e.g., 30 seconds)
  2. **Request Timeouts**: Time out entire RPC operations (produce, fetch, join group)
  3. **Internal Operation Timeouts**: Time out log flushes, lock acquisitions, channel operations
- **Diagnosis Technique**: When timeouts occur, examine:
  - *Which operation* timed out (produce, fetch, heartbeat)
  - *Which component* was involved (which broker, which partition)
  - *What else* was happening at that time (other requests, garbage collection, disk I/O)
  - *Correlation* with other symptoms (high latency, stuck consumers)
- **Timeout Configuration Table**:

Timeout Type	Default Value	Purpose	What to Check When Hit
`socket.timeout.ms`	30000	Network read/write deadline	Network connectivity, broker load
`request.timeout.ms`	30000	Complete RPC operation	Broker processing time, lock contention
`rebalance.timeout.ms`	60000	Maximum rebalance duration	Consumer join/leave coordination
`fetch.timeout.ms`	500	Consumer poll wait	Message availability, consumer lag
`heartbeat.timeout.ms`	10000	Consumer liveness detection	Network partitions, consumer GC pauses

Controlled Experimentation (Scientific Method)

> **Mental Model**: Think of debugging as conducting **laboratory experiments**. You formulate hypotheses about root causes, design tests to isolate variables, run experiments, and analyze results.

- **Process**:
  1. **Observe**: Document the symptom precisely (what, when, where, how often)
  2. **Hypothesize**: Propose a potential root cause based on understanding of the system
  3. **Experiment**: Design a minimal test to verify/reject the hypothesis
  4. **Analyze**: Compare actual results with expected results
  5. **Iterate**: Refine hypothesis based on results
- **Example Experiment for Message Loss**:
  - *Observation*: 15% of messages lost with `acks=1`
  - *Hypothesis*: Leader is crashing after acknowledging but before replicating to disk
  - *Experiment*: Add detailed logging at leader: log immediately before and after disk sync
  - *Analysis*: Check if crash logs show unsynced writes; verify with `WAL.Append(..., sync=true)`

Concurrency Race Detection

> **Mental Model**: Think of race conditions as **traffic intersections without signals**. Multiple threads (cars) approach simultaneously, and depending on timing, may collide or proceed safely. Detection tools are like traffic cameras capturing these interactions.

- **Go-Specific Tools**:
  - `go test -race`: Run tests with race detector enabled (adds overhead but finds data races)
  - `sync/atomic` operations: For simple counters where lock overhead is unacceptable
  - `context.Context` for cancellation: Properly propagate cancellation through goroutines
- **Common Race Patterns to Watch For**:
  - *Read-Modify-Write*: Incrementing a counter without synchronization
  - *Check-Then-Act*: Checking a condition then acting on it without holding lock
  - *Publication*: Publishing a reference before initialization completes

12.3 Symptom → Cause → Fix Table

This table maps observable symptoms to their most likely root causes, diagnosis steps, and specific fixes. Use it as a starting point for your debugging journey.

Symptom	Likely Cause	Diagnosis Steps	Fix
Messages Lost	1. **Producer using `acks=0` (fire-and-forget)** with network failures 2. **Leader crashes** after acknowledging but before persisting to disk (`acks=1`) 3. **ISR shrinks to empty**, then unclean leader election discards unreplicated data 4. **Consumer commits offset** before processing, then crashes	1. Check producer configuration for `AcksNone` 2. Examine broker logs for crashes right after produce acknowledgments 3. Monitor ISR size metrics; check for follower lag exceeding threshold 4. Check consumer commit logic (offset committed before business logic)	1. Use `AcksLeader` or `AcksAll` for durability 2. Ensure `WAL.Append` calls `sync=true` when `acks>=1` 3. Configure `unclean.leader.election.enable=false` 4. Commit offsets **after** successful message processing
Duplicate Messages	1. **Producer retries** without idempotency after transient errors 2. **Consumer rebalance** causes reprocessing of uncommitted offsets 3. **Consumer commits offset** after processing but crashes before commit persists 4. **Leader epoch mismatch** during leadership change causes old leader to accept writes	1. Check producer logs for retry attempts with same batch 2. Examine consumer group generation changes; check offset commits during rebalance 3. Check disk sync on offset commit store (`OffsetStore.persistGroup`) 4. Check leader epoch validation in produce request handling	1. Implement idempotent producer with sequence numbers 2. Implement cooperative rebalancing with offset commit before partition revocation 3. Use synchronous offset commits with disk sync 4. Validate leader epoch in all write paths; reject stale writes
Consumers Stuck	1. **Heartbeat failures** causing coordinator to mark consumer dead 2. **Fetch loop blocked** on slow I/O or deadlock 3. **No messages** at current offset (log compacted or truncated) 4. **Assignment mismatch** - consumer thinks it	1. Check coordinator logs for heartbeat timeouts 2. Add debug logs to fetch loop; check for lock acquisition 3. Compare consumer offset with log start offset 4. Verify assignment in `ConsumerMetadata` vs actual fetch requests	1. Increase `session.timeout.ms` or fix long GC pauses 2. Add timeouts to all blocking operations; fix deadlocks 3. Implement auto-offset reset to `earliest` or `latest` 4. Validate assignments in `Poll()` before fetching

Symptom	Likely Cause	Diagnosis Steps	Fix
	has partitions it doesn't		
High Latency	1. **Small batch sizes** causing excessive network round trips 2. **Disk I/O contention** from multiple partitions on same disk 3. **Memory pressure** causing excessive garbage collection 4. **Network bottlenecks** between producers and broker cluster	1. Measure batch sizes sent; check `BatchSize` and `LingerMs` settings 2. Monitor disk I/O wait times; check if partitions share data directory 3. Monitor Go GC pause times and heap size 4. Measure network latency between client and broker nodes	1. Tune `batch.size` and `linger.ms` for throughput/latency tradeoff 2. Separate partition data to different physical disks 3. Optimize memory allocations; reuse buffers; set GOGC appropriately 4. Ensure brokers are geographically close to producers or use compression
Rebalance Storms	1. **Session timeout too short** for consumers with GC pauses 2. **Heartbeat thread blocked** by long-running operations 3. **Coordinator overload** causing delayed heartbeat processing 4. **Network flakiness** causing intermittent connectivity loss	1. Check consumer GC logs for long pauses (> session timeout) 2. Verify heartbeat runs in dedicated goroutine without blocking 3. Monitor coordinator CPU/memory during rebalances 4. Check network packet loss statistics between consumers and coordinator	1. Increase `session.timeout.ms` to accommodate GC pauses 2. Run heartbeat in separate goroutine with its own context 3. Add load shedding to coordinator; scale horizontally 4. Improve network reliability or implement exponential backoff on rejoin

Symptom	Likely Cause	Diagnosis Steps	Fix
Inconsistent Ordering	1. **Key hash collision** sending same key to different partitions 2. **Producer retries** reordering messages due to partial failures 3. **Multiple producers** writing to same partition without coordination 4. **Consumer reading from wrong offset** due to incorrect index lookup	1. Test partitioner with known keys; verify consistent mapping 2. Examine producer logs for out-of-order retry attempts 3. Check if multiple producer instances use same partition 4. Verify `Index.FindEntry` logic returns correct position	1. Ensure `HashPartitioner.Partition` uses stable hash algorithm 2. Implement producer idempotency with sequence numbers 3. Use single producer per partition or leader epoch fencing 4. Test index lookup with edge cases (exact match, between entries)
ISR Shrinkage to Empty	1. **Follower lag threshold too small** for normal operation 2. **Network partition** isolating followers from leader 3. **Follower I/O issues** causing slow replication 4. **Leader not updating follower state** in `ISRManager`	1. Check `replica.lag.time.max.ms` vs actual replication latency 2. Examine network connectivity between broker nodes 3. Monitor follower disk I/O metrics and replication fetch times 4. Verify `ISRManager.UpdateFollowerState` is called on successful fetch	1. Increase `replica.lag.time.max.ms` to accommodate temporary lag 2. Implement network health checks and partition detection 3. Separate replication traffic to dedicated network interfaces 4. Ensure follower state updates are not silently dropped
Memory Leak	1. **Goroutine leak** from unbounded channel operations or missing `context` cancellation 2. **Cache growth without eviction** in metadata or connection pools 3. **File descriptor leak** from unclosed log segment files 4. **Reference cycles** preventing garbage collection	1. Monitor goroutine count over time; check for `go` statements without cleanup 2. Track cache sizes (`MetadataCache` , `ConnectionPool`) 3. Check `lsof` output for growing open files in broker process 4. Use heap profiling to identify reference chains	1. Add `defer cancel()` to all `context.WithCancel` ; close all channels 2. Implement LRU eviction or size limits on caches 3. Ensure `LogSegment` files are closed when no longer needed 4. Break cycles using weak references or explicit cleanup methods

Debugging Workflow Example: Diagnosing Duplicate Messages

Let's walk through a concrete debugging scenario using the techniques above:

1. **Observe Symptom**: Consumer application reports processing the same message ID twice.
2. **Initial Investigation**:
   - Check consumer logs: "Committing offset 142 for partition topic-0"
   - Check producer logs: "Retrying batch for topic-0 partition 0 (attempt 2/3)"
   - Correlation: Retry occurs around same time as offset commit
3. **Form Hypothesis**: Producer retry after successful send but before acknowledgment received.
4. **Design Experiment**:
   - Add logging to producer: "Batch sent with sequence X" and "Received ack for sequence X"
   - Add logging to consumer: "Processing offset Y" and "Committed offset Y"
   - Reproduce with network simulation: Add artificial latency between send and ack
5. **Analyze Results**:
   - Logs show: Send seq=5 → Network delay → Timeout → Retry seq=5 → Original ack arrives → Retry succeeds → Two copies
   - Consumer processes offset 142 twice because it crashes between processing and commit
6. **Root Cause**: Two issues combined:
   - Producer: No idempotency, retries create duplicates
   - Consumer: Offset committed asynchronously, crash causes reprocessing
7. **Implement Fix**:
   - Producer: Add `ProducerID` and sequence numbers to `RecordBatch`, broker deduplicates
   - Consumer: Change `CommitAsync` to `CommitSync` after processing completes
8. **Verify Fix**:
   - Run integration test with injected network failures
   - Confirm no duplicates in 10,000 message test
   - Monitor for performance regression from synchronous commits

Implementation Guidance

> **Mental Model**: Think of debugging infrastructure as **scaffolding around a building under construction**. You build temporary platforms (logging), install safety nets (metrics), and create access points (debug APIs) that help you work on the main structure, then remove or minimize them when the building is complete.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Logging	`logrus` or `zap` for structured JSON logging	OpenTelemetry with distributed tracing to correlate requests across services
Metrics	Prometheus client library exposing `/metrics` endpoint	StatsD/DataDog integration with dashboards and alerting
Profiling	Go's built-in `pprof` HTTP endpoints (`/debug/pprof/`)	Continuous profiling with Parca or Pyroscope for production
Debug API	Simple HTTP server with JSON endpoints for internal state	gRPC health checks and reflection for deeper inspection
Chaos Testing	Manual network partition simulation (`iptables`)	ChaosMesh or Litmus for automated failure injection

B. Recommended File/Module Structure

```
project-root/
├── cmd/
│   ├── broker/                # Broker executable
│   │   └── main.go
│   ├── producer-cli/          # Producer CLI tool
│   │   └── main.go
│   └── consumer-cli/          # Consumer CLI tool
│       └── main.go
├── internal/
│   ├── debug/                 # Debugging infrastructure
│   │   ├── debug_server.go    # HTTP server for debug endpoints
│   │   ├── metrics.go         # Prometheus metrics collection
│   │   ├── pprof.go           # pprof endpoint setup
│   │   └── state_dump.go      # State snapshot utilities
│   ├── logging/               # Structured logging setup
│   │   ├── logger.go          # Logger initialization
│   │   ├── context.go         # Context with correlation IDs
│   │   └── fields.go          # Common log field definitions
│   └── ... (other components)
└── scripts/
    ├── simulate-failure.sh    # Script to inject network failures
    ├── analyze-logs.py        # Log analysis utility
    └── memory-profile.sh      # Memory profiling script
```

C. Infrastructure Starter Code: Structured Logger Wrapper

```go
// internal/logging/logger.go

package logging

import (
    "context"
    "os"
    "runtime"
    "strings"

    log "github.com/sirupsen/logrus"
)

type ctxKey string

const (
    correlationIDKey ctxKey = "correlation_id"
    componentKey     ctxKey = "component"
)

// InitLogger configures the global logger with structured JSON output
func InitLogger(level string, enableCaller bool) {
    logLevel, err := log.ParseLevel(level)
    if err != nil {
        logLevel = log.InfoLevel
    }
    log.SetLevel(logLevel)

    log.SetFormatter(&log.JSONFormatter{
        TimestampFormat: "2006-01-02T15:04:05.999Z07:00",
        FieldMap: log.FieldMap{
            log.FieldKeyTime:  "timestamp",
            log.FieldKeyLevel: "severity",
            log.FieldKeyMsg:   "message",
            log.FieldKeyFunc:  "caller",
```

```
        },
    })

    log.SetOutput(os.Stdout)

    if enableCaller {
        log.SetReportCaller(true)
    }
}

// WithContext returns a log entry with fields from context
func WithContext(ctx context.Context) *log.Entry {
    entry := log.NewEntry(log.StandardLogger())

    if ctx != nil {
        if corrID, ok := ctx.Value(correlationIDKey).(string); ok && corrID != "" {
            entry = entry.WithField("correlation_id", corrID)
        }
        if component, ok := ctx.Value(componentKey).(string); ok && component != "" {
            entry = entry.WithField("component", component)
        }
    }

    // Add caller info if enabled
    if log.StandardLogger().ReportCaller {
        if pc, file, line, ok := runtime.Caller(1); ok {
            funcName := runtime.FuncForPC(pc).Name()
            // Simplify file path
            if idx := strings.LastIndex(file, "/"); idx != -1 {
                file = file[idx+1:]
            }
            entry = entry.WithField("caller",
                log.Fields{"file": file, "line": line, "function": funcName})
        }
```

```
    }


    return entry
}

// NewContextWithCorrelation creates a new context with correlation ID
func NewContextWithCorrelation(parent context.Context, corrID string) context.Context {
    return context.WithValue(parent, correlationIDKey, corrID)
}

// NewContextWithComponent creates a new context with component name
func NewContextWithComponent(parent context.Context, component string) context.Context {
    return context.WithValue(parent, componentKey, component)
}

// LogLevelFromString safely parses log level
func LogLevelFromString(level string) log.Level {
    l, err := log.ParseLevel(level)
    if err != nil {
        return log.InfoLevel
    }
    return l
}
```

D. Core Logic Skeleton: Debug State Collection

```go
// internal/debug/state_dump.go

package debug

import (
    "encoding/json"
    "fmt"
    "runtime"
    "sync"
    "time"

    "github.com/yourproject/internal/types"
)

// DebugStateCollector gathers state from various components
type DebugStateCollector struct {
    broker    *types.Server
    startTime time.Time
    mu        sync.RWMutex
    metrics   map[string]interface{}
}

// NewDebugStateCollector creates a new collector
func NewDebugStateCollector(broker *types.Server) *DebugStateCollector {
    return &DebugStateCollector{
        broker:    broker,
        startTime: time.Now(),
        metrics:   make(map[string]interface{}),
    }
}

// CollectBrokerState gathers comprehensive broker state
func (d *DebugStateCollector) CollectBrokerState() (map[string]interface{}, error) {
    d.mu.Lock()
    defer d.mu.Unlock()
```

```
    state := make(map[string]interface{})

    // TODO 1: Collect basic broker info (ID, host, port, uptime)
    // TODO 2: Collect memory statistics using runtime.MemStats
    // TODO 3: Collect goroutine count and thread information
    // TODO 4: Collect topic and partition state (call broker.GetDebugState())
    // TODO 5: Collect consumer group state from coordinator
    // TODO 6: Collect replication state from replica manager
    // TODO 7: Collect connection pool statistics
    // TODO 8: Collect request queue depths and processing times
    // TODO 9: Collect disk usage information for log directories
    // TODO 10: Collect any custom metrics stored in d.metrics

    return state, nil
}

// RecordMetric stores a custom metric for debugging
func (d *DebugStateCollector) RecordMetric(name string, value interface{}) {
    d.mu.Lock()
    defer d.mu.Unlock()
    d.metrics[name] = value
}

// GetMemoryStats returns Go memory statistics
func (d *DebugStateCollector) GetMemoryStats() map[string]interface{} {
    var memStats runtime.MemStats
    runtime.ReadMemStats(&memStats)

    return map[string]interface{}{
        "alloc_mb":        float64(memStats.Alloc) / 1024 / 1024,
        "total_alloc_mb":  float64(memStats.TotalAlloc) / 1024 / 1024,
        "sys_mb":          float64(memStats.Sys) / 1024 / 1024,
        "num_gc":          memStats.NumGC,
```

```
        "last_gc_pause_ns": memStats.PauseNs[(memStats.NumGC+255)%256],
        "goroutines":      runtime.NumGoroutine(),
        "cgo_calls":       runtime.NumCgoCall(),
    }
}

// GetUptime returns formatted uptime
func (d *DebugStateCollector) GetUptime() string {
    uptime := time.Since(d.startTime)
    days := int(uptime.Hours() / 24)
    hours := int(uptime.Hours()) % 24
    minutes := int(uptime.Minutes()) % 60
    seconds := int(uptime.Seconds()) % 60

    if days > 0 {
        return fmt.Sprintf("%dd %dh %dm %ds", days, hours, minutes, seconds)
    }
    return fmt.Sprintf("%dh %dm %ds", hours, minutes, seconds)
}

// JSONString returns state as pretty-printed JSON
func (d *DebugStateCollector) JSONString() (string, error) {
    state, err := d.CollectBrokerState()
    if err != nil {
        return "", err
    }

    data, err := json.MarshalIndent(state, "", "  ")
    if err != nil {
        return "", err
    }

    return string(data), nil
}
```

E. Language-Specific Hints (Go)

1. **Memory Profiling**: Use `go tool pprof -alloc_space http://localhost:6060/debug/pprof/heap` to find memory leaks.
2. **Block Profiling**: Use `go tool pprof http://localhost:6060/debug/pprof/block` to find goroutine blocking.
3. **Race Detection**: Always run tests with `go test -race ./...`; add `-race` flag to production for critical services.
4. **Context Usage**: Pass `context.Context` through all async operations; use `context.WithTimeout` for operations that should complete within a deadline.
5. **Structured Logging**: Use `logrus.Fields` for key-value pairs rather than formatted strings.
6. **Metrics Collection**: Use Prometheus `Gauge`, `Counter`, and `Histogram` for different types of measurements.
7. **Debug Endpoints**: Use `net/http/pprof` package and register handlers: `import _ "net/http/pprof"`.
8. **Testing Network Failures**: Use `nettest` package for simulated network conditions in tests.

F. Debugging Tips Table

Symptom	How to Diagnose	Useful Command/Tool	What to Look For
High CPU	CPU profiling	`go tool pprof -seconds 30 http://localhost:6060/debug/pprof/profile`	Functions with highest `cum` (cumulative) time
Memory Leak	Heap profiling	`go tool pprof -alloc_space http://localhost:6060/debug/pprof/heap`	Objects with highest `inuse_space` retained over time
Goroutine Leak	Goroutine dump	`curl http://localhost:6060/debug/pprof/goroutine?debug=2`	Goroutines stuck in `chan send` or `chan receive`
Slow Disk I/O	Disk monitoring	`iostat -x 1` (Linux)	High `await` or `%util` on data disk
Network Issues	Packet capture	`tcpdump -i any port 9092 -w capture.pcap`	Retransmissions, zero windows, RST packets
Deadlock	Mutex profiling	`go tool pprof http://localhost:6060/debug/pprof/mutex`	Mutexes with high contention times
GC Pressure	GC trace	`GODEBUG=gctrace=1 ./broker`	High `STW` (stop-the-world) pause times

G. Milestone Checkpoint: Debugging Verification

After implementing logging and debugging infrastructure, verify it works:

1. **Start your broker** with debug logging enabled:

```
LOG_LEVEL=debug ./broker --port 9092 --data-dir ./data
```

2. **Check debug endpoints**:

```
curl http://localhost:6060/debug/state | jq .
curl http://localhost:6060/debug/pprof/goroutine?debug=1 | head -50
```

3. **Produce test messages** and verify logs show the flow:

```
./producer-cli --topic test --message "debug test"
# Check broker logs for: "Appended records", "Handled produce request"
```

4. **Simulate a failure** and verify error logging:

```
# In another terminal, kill the broker process
pkill -9 broker
# Check logs for: "Received signal", "Shutting down", "Closing WAL"
```

5. **Expected Output**:
    - Structured JSON logs for all major operations
    - Debug endpoints return valid JSON with broker state
    - Errors include sufficient context for diagnosis
    - No panics or data races (run with `-race` flag)
6. **Signs of Problems**:
    - Logs are missing key fields (partition, offset, correlation ID)
    - Debug endpoints timeout or return incomplete data
    - Memory usage grows continuously during idle period
    - Goroutine count increases without bound

---

13. Future Extensions

> **Milestone(s):** All (post-core extension opportunities)

Having successfully implemented the core distributed message queue with partitioned topics, producer batching, consumer groups, and leader-follower replication, you now possess a solid foundation in distributed systems principles. This section explores potential advanced features you could implement to deepen your understanding and extend the system's capabilities. Each extension represents real-world challenges faced by production message brokers like Apache Kafka, and implementing them will sharpen your skills in performance optimization, consistency guarantees, and resource management.

13.1 Potential Feature Additions

The core system you've built provides reliable, ordered message delivery with basic fault tolerance. However, production systems require additional features to handle diverse workloads, optimize resource usage, and guarantee stronger semantics. Below are several advanced extensions, each with a mental model to build intuition before technical details.

Feature	Mental Model	Core Benefit	Complexity
Compression	A vacuum-sealed package: multiple items compressed into a smaller space for efficient shipping, then decompressed at destination.	Reduces network bandwidth and disk storage by 60-90% for text-based messages.	Medium
Exactly-Once Semantics	A bank's transaction ledger with debit/credit pairs: each operation is atomic and idempotent, ensuring final balance reflects exactly one execution of each transfer.	Eliminates duplicate processing in stream processing pipelines, critical for financial use cases.	High
Quotas & Throttling	A highway toll system with speed limits and vehicle quotas: controls throughput per client to prevent any single user from overwhelming shared infrastructure.	Protects system stability from misbehaving clients and enables multi-tenancy.	Medium
Log Compaction	A key-value store's "last value wins" semantics: older updates for the same key are periodically discarded, retaining only the latest state.	Enables using the log as a durable, replayable database of current state rather than infinite history.	High
Custom RPC Layer	Replacing postal mail with a dedicated courier service: designing your own protocol optimized for your specific communication patterns.	Reduces serialization overhead, enables zero-copy transfers, and provides finer control over connection management.	High
Tiered Storage	A library archive: frequently accessed recent books stay on main shelves, while older volumes move to deep storage, retrievable when needed.	Dramatically reduces storage costs for long-retention topics while maintaining access to historical data.	Very High
Transaction Support	A database's ACID transactions across multiple partitions: all writes within a transaction are committed or aborted together.	Enconsistent updates across multiple partitions, crucial for event-driven microservices.	Very High
Schema Registry	A central dictionary for data formats: producers and consumers agree on message structure via shared schema definitions, enabling evolution.	Prevents data corruption from format mismatches and enables efficient binary serialization.	Medium

Compression: The Space-Saving Courier

Imagine you run a courier service shipping books between libraries. Instead of sending each book individually, you pack multiple books into a single box, vacuum-seal it to reduce volume, then label the box with its contents. At the destination, the box is opened and books returned to normal size. **Compression** works similarly: multiple records in a `RecordBatch` are compressed together using algorithms like GZIP, Snappy, or LZ4 before transmission and storage, then decompressed when consumers fetch them.

The key insight is that compression operates at the **batch level**, not individual records, because compression algorithms achieve better ratios with more data. Your existing `RecordBatch` structure already has an `Attributes` field with bits for compression type—this extension involves implementing the actual compression/decompression logic in the `RecordBatch.Encode()` and `DecodeRecordBatch()` methods, plus configuring the `Producer` to compress batches meeting a minimum size threshold.

Exactly-Once Semantics: The Bank Teller's Ledger

Consider a bank teller processing deposits. Each transaction gets a unique sequence number in the ledger. If the network fails after the deposit but before the customer receives confirmation, the customer might retry—but the teller checks the sequence number and ignores duplicates. **Exactly-once semantics** extends your idempotent producer (which prevents duplicates within a single partition) to span multiple partitions and consumer side effects via **transactional producers** and **read-committed isolation**.

This requires three coordinated mechanisms: 1) **Transactional IDs** for producer fencing, 2) **Transaction coordinator** (a specialized broker role) managing two-phase commit across partitions, and 3) **Transaction markers** written to logs to indicate commit/abort boundaries. Consumers in `read_committed` mode only deliver messages after the commit marker, avoiding exposure to uncommitted data.

Quotas & Throttling: The Highway Traffic Control

A highway system uses toll booths, speed limits, and vehicle quotas to ensure no single route becomes congested. **Quotas** in messaging systems similarly limit the byte rate or request rate per client, user, or topic to prevent a single misconfigured producer from overwhelming broker network I/O or a greedy consumer from monopolizing CPU. Throttling involves measuring traffic, comparing against quotas, and delaying excess requests or adding backpressure.

Your system already has natural measurement points: `Server.HandleProduce()` and `Server.HandleFetch()` can track bytes per client ID. The challenge is implementing fair, responsive throttling without introducing substantial overhead. A token bucket algorithm per client, checked before processing each request, provides smooth rate limiting.

Log Compaction: The Librarian's Archive Purge

Imagine a librarian periodically scanning shelves, removing all but the latest edition of each book title. **Log compaction** does this for keyed messages: it periodically rewrites log segments, discarding older records for keys that have more recent updates, while retaining all records for keys without updates (including null-keyed records). This transforms the log from an infinite append-only history into a finite, replayable key-value store.

Compaction requires background threads scanning `LogSegment` files, creating new compacted segments, and atomically swapping them. The `Log` needs to track the **clean offset**—the point before which all keys are guaranteed to be compacted. Consumers can then optionally read from this offset to get the latest state of all keys.

Custom RPC Layer: Building Your Own Postal Service

While you've implemented a basic length-prefixed TCP protocol, production systems often optimize further with custom binary protocols supporting multiplexing, header compression, and zero-copy transfers. Designing a **custom RPC layer** involves creating a wire format tailored to your specific request/response patterns, potentially using frameworks like gRPC or building directly on epoll/kqueue for high throughput.

This extension would replace your current `TCPServer` and request handling with a more sophisticated transport that better leverages system calls, reduces allocations, and provides better connection pooling. It's an excellent deep dive into network programming and performance optimization.

13.2 Design Considerations for Extensions

Each extension presents unique design challenges and requires modifications to your existing architecture. Below we analyze each feature's impact through the lens of Architecture Decision Records (ADRs), comparing implementation approaches and highlighting integration points with your current components.

13.2.1 Compression: Batch-Level vs Record-Level

> **Decision: Implement Compression at RecordBatch Level**
>
> - **Context**: Messages often have high redundancy (JSON fields, repeated text), especially within related records produced together. We need to reduce network and storage overhead without excessive CPU cost.
> - **Options Considered**:
>   1. **Record-level compression**: Each `Record` compressed individually.
>   2. **Batch-level compression**: Entire `RecordBatch` compressed as a unit.
>   3. **Streaming compression**: Continuous compression across batch boundaries.
> - **Decision**: Implement batch-level compression using the existing `RecordBatch.Attributes` field to indicate compression type.
> - **Rationale**: Batch-level achieves better compression ratios (more data for dictionary-based algorithms) and aligns with the natural unit of network transfer and disk I/O. It also simplifies implementation: compression/decompression happens at the same boundaries as existing serialization.
> - **Consequences**: Producers must accumulate enough records to make compression worthwhile; smaller batches gain less benefit. Consumers must handle decompression transparently. CPU overhead concentrated at producer and consumer rather than brokers.

Option	Pros	Cons	Suitable For
Record-level	Fine-grained, no batching delay	Poor compression ratio, high overhead	Large individual messages (>1MB)
Batch-level	Good ratio, aligns with existing unit	Requires minimum batch size	Typical throughput-optimized workloads
Streaming	Best ratio across many batches	Complex state management, harder random access	Very high throughput archival

Integration with Current Design:

- **Producer**: The `Sender.sendBatch()` method would compress the serialized batch bytes if `batch.Attributes` indicates compression and batch size exceeds `compression.threshold`.
- **Broker**: The `Log.Append()` stores compressed bytes directly; brokers don't decompress except for validation. The `Log.Read()` returns compressed batches to consumers.
- **Consumer**: `DecodeRecordBatch()` detects compression via `Attributes` and decompresses before parsing individual records.
- **Configuration**: Add `CompressionType` and `CompressionThreshold` to `ProducerConfig`.

The `RecordBatch.Encode()` algorithm extends to:

1. Serialize records to temporary buffer
2. If compression enabled and buffer size > threshold:
   - Compress buffer using selected algorithm
   - Set compression bits in `Attributes`
3. Calculate CRC on compressed data (if using)
4. Write batch header followed by compressed data

13.2.2 Exactly-Once Semantics: Transaction Coordinator Placement

> **Decision: Embed Transaction Coordinator in Existing Brokers**
>
> - **Context**: Transactions require a coordinator to manage two-phase commit across multiple partitions, which may be hosted on different brokers. We need to decide where this coordinator runs.
> - **Options Considered**:
>   1. **Dedicated coordinator broker**: Single broker elected as transaction coordinator.
>   2. **Embedded in all brokers**: Each broker can coordinate transactions for producers that choose it.
>   3. **External service**: Separate process/container managing transactions.
> - **Decision**: Embed transaction coordination capability in all brokers, with producers hashing transactional ID to select a coordinator.
> - **Rationale**: This follows Kafka's design, ensuring scalability (coordination load distributes across cluster) and fault tolerance (coordinator failure only affects its transactions). It leverages existing broker infrastructure for persistence and networking.
> - **Consequences**: Each broker needs additional transaction state; producer must discover coordinator via hash; transaction recovery requires scanning logs.

Implementation Outline:

1. **Transaction Coordinator Component**: New `TransactionCoordinator` type in each broker, managing `TransactionalId → ProducerIdMapping` and `TransactionLog`.
2. **Transaction Protocol**:
   - `InitProducerId(TransactionalId)` → returns `ProducerId, Epoch` (for fencing)
   - `AddPartitionsToTransaction(TxnId, [TopicPartition])`
   - `EndTransaction(TxnId, CommitOrAbort)`
3. **Transaction Log**: Special internal topic `__transaction_state` storing transaction metadata.
4. **Consumer Read Committed**: Modify `Log.Read()` to skip messages between `BEGIN` and `COMMIT/ABORT` markers when `isolation.level=read_committed`.

Your existing `Partition` log would need to support **transaction markers**—special control records written during two-phase commit. The `HighWatermark` advancement logic must consider that messages before an uncommitted transaction marker shouldn't be exposed to `read_committed` consumers.

13.2.3 Quotas: Measurement and Enforcement Points

> **Decision: Enforce Quotas at Request Handler with Token Bucket**
>
> - **Context**: Need to limit client throughput to protect cluster stability. Must decide where to measure and how to enforce limits.
> - **Options Considered**:
>   1. **Network layer enforcement**: Throttle at TCP accept/read level.
>   2. **Request handler enforcement**: Delay processing after parsing request but before business logic.
>   3. **Response throttling**: Allow processing but delay response.
> - **Decision**: Implement token bucket algorithm in `Server.HandleProduce()` and `Server.HandleFetch()` methods, tracking bytes per client ID.
> - **Rationale**: Request handler has access to structured request information (client ID, topic, byte count) and can make precise decisions. Token bucket provides smooth throttling (not all-or-nothing) and accumulates credits during idle periods.
> - **Consequences**: Adds per-request overhead for quota checking; delayed responses may cause client timeouts.

Integration Points:

- **Quota Config**: Add to `Server` config: `quota.producer.byte.rate`, `quota.consumer.byte.rate` (bytes/sec per client).
- **Quota Manager**: New `QuotaManager` with `CheckQuota(clientID string, bytes int) time.Duration` returning delay needed.
- **Request Handling**:

```go
// In Server.HandleProduce

delay := s.quotaManager.CheckQuota(clientID, len(request.Data))

if delay > 0 {

    time.Sleep(delay) // Or implement asynchronous delaying

}
```

- **Metrics**: Track throttling time per client for observability.

A key challenge is **quota distribution across broker cluster**—if a client connects to multiple brokers, each sees only partial traffic. A simplified approach enforces quotas per-broker, which works for clients with stable broker connections.

13.2.4 Log Compaction: Background vs Online Compaction

> **Decision: Implement Background Compaction Thread per Log**
>
> - **Context**: Need to periodically clean logs while maintaining availability for reads and writes.
> - **Options Considered**:
>   1. **Background thread**: Separate goroutine periodically scans and rewrites segments.
>   2. **Online compaction**: Compact during normal writes (append-only with periodic merging).
>   3. **Offline compaction**: Stop serving partition, compact entire log, resume.
> - **Decision**: Background compaction thread per `Log` that runs when segments meet compaction criteria.
> - **Rationale**: Background compaction minimizes impact on read/write latency. It can be scheduled during low-load periods. The algorithm mirrors Kafka's log cleaner.
> - **Consequences**: Increases disk I/O during compaction; requires careful coordination to avoid compacting active segments.

Compaction Algorithm:

1. **Select Segments**: Choose log segments where `dirtyRatio = (segment.size - clean.size) / segment.size > minDirtyRatio`.
2. **Build Key Map**: Read selected segments, track latest offset per key.
3. **Write Clean Segment**: Create new segment containing only latest records for each key (plus all records without keys).
4. **Atomically Replace**: Swap old segments with new clean segment in `Log.Segments`.
5. **Update Clean Offset**: Record offset before which all keys are compacted.

Your `Log` would need new fields:

```go
type Log struct {
    // ... existing fields
    CleanOffset      int64        // Offset before which compaction is guaranteed
    CompactConfig    CompactConfig
    compacting       bool         // Guard against concurrent compaction
}
```

Consumers reading with `isolation.level=read_committed` and from `clean.offset` would see a consistent snapshot of the latest value for each key—effectively turning your message queue into a persistent key-value store.

13.2.5 Custom RPC Layer: Protocol Design Trade-offs

> **Decision: Implement Connection Multiplexing with Request Batching**
>
> - **Context**: Current simple TCP server handles one request per connection serially, limiting throughput.
> - **Options Considered**:
>   1. **HTTP/2 with gRPC**: Use standard multiplexed protocol.
>   2. **Custom binary with framing**: Design own length-prefixed multiplexed protocol.
>   3. **UDP-based protocol**: For lower latency but lossy transport.
> - **Decision**: Design custom binary protocol supporting request pipelining and connection multiplexing.
> - **Rationale**: Custom protocol allows optimization for specific patterns (small metadata requests mixed with large data transfers). Avoids HTTP overhead while maintaining reliability via TCP.
> - **Consequences**: Significant implementation effort; must handle backpressure, timeouts, and connection lifecycle.

Protocol Enhancements:

1. **Multiplexing**: Single connection carries multiple concurrent request/response streams with correlation IDs.
2. **Zero-copy for large fetches**: Use `sendfile()` or similar to transfer log segments without copying to userspace.
3. **Header compression**: Repeated headers (topic names, client IDs) compressed across requests.
4. **Binary encoding**: Replace current ad-hoc encoding with Protocol Buffers or similar for schema evolution.

This extension would essentially replace your entire `network` package with a more sophisticated implementation, touching almost every component. It's the most invasive but educational extension for understanding high-performance network services.

13.2.6 Tiered Storage: Hot/Warm/Cold Architecture

> **Decision: Implement Transparent Tiering with Async Offload**
>
> - **Context**: Logs grow indefinitely, consuming expensive local SSD storage. Need to move older data to cheaper object storage (S3-like).
> - **Options Considered**:
>   1. **Manual tiering**: Users manually move segments between tiers.
>   2. **Transparent offload**: System automatically moves segments based on age.
>   3. **Hierarchical storage**: OS-level tiering (e.g., LVM cache).
> - **Decision**: Implement background offload of closed log segments to object storage, with local LRU cache for recently accessed segments.
> - **Rationale**: Transparent operation requires no application changes. Async offload minimizes performance impact. Object storage provides durability and virtually unlimited capacity.
> - **Consequences**: Adds complexity for fetch path (check local, then remote); requires object storage integration; network latency for cold reads.

Architecture Impact:

- **LogSegment**: Add `StorageTier` field (`LOCAL`, `REMOTE`, `ARCHIVED`) and `RemoteURI`.
- **Log.Read()**: If segment is remote, fetch to local cache (possibly with prefetch).
- **TierManager**: Background goroutine scanning for segments older than `retention.ms` to offload.
- **Configuration**: Add `tiered.storage.enabled`, `remote.storage.endpoint`, `local.cache.size`.

This extension transforms your broker from a storage node to a **caching layer** over durable object storage—a common pattern in modern data systems like Kafka with Tiered Storage.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Compression	Standard library `compress/gzip`	C bindings to Snappy or LZ4 via cgo
Transaction Coordinator	In-memory state with WAL for recovery	Full Paxos/Raft for coordinator fault tolerance
Quota Enforcement	Simple token bucket per client	Distributed quota tracking via gossip protocol
Log Compaction	Single-pass scanner building map in memory	Multi-pass with disk-based hash tables for large logs
Custom RPC	Connection-per-request with pipelining	Reactor pattern with epoll/kqueue event loop
Tiered Storage	Mock object storage (local filesystem)	Integration with MinIO or AWS S3 SDK

Recommended Module Structure for Extensions

```
project-root/
  internal/
    compression/           # Compression algorithms
      codec.go            # Interface: Compress(data []byte) []byte
      gzip.go             # GZIP implementation
      snappy.go           # Snappy implementation (optional)
    transaction/
      coordinator.go      # TransactionCoordinator
      log.go              # Transaction log (internal topic)
      producer_id.go      # Producer ID generation and fencing
    quotas/
      manager.go          # QuotaManager with token buckets
      metrics.go          # Throttling metrics collection
    compaction/
      cleaner.go          # LogCleaner background thread
      strategy.go         # Compaction strategy interface
    storage/
      tier/               # Tiered storage
        manager.go        # TierManager for offloading
        cache.go          # LRU cache for remote segments
        s3.go             # S3 client (optional)
    protocol/             # Enhanced RPC layer
      framing.go          # Multiplexed frame encoding
      connection.go       # Managed connection with pipelining
      reactor.go          # Event loop (advanced)
```

Compression Starter Code

```go
// internal/compression/codec.go

package compression


type Codec interface {

    Compress(src []byte) ([]byte, error)

    Decompress(src []byte) ([]byte, error)

    Name() string

}


// internal/compression/gzip.go

package compression


import "compress/gzip"

import "bytes"


type GzipCodec struct{}


func (g *GzipCodec) Compress(src []byte) ([]byte, error) {

    var buf bytes.Buffer

    w := gzip.NewWriter(&buf)

    if _, err := w.Write(src); err != nil {

        return nil, err

    }

    if err := w.Close(); err != nil {

        return nil, err

    }

    return buf.Bytes(), nil

}


func (g *GzipCodec) Decompress(src []byte) ([]byte, error) {

    r, err := gzip.NewReader(bytes.NewReader(src))

    if err != nil {

        return nil, err

    }

    defer r.Close()
```

```
    var buf bytes.Buffer
    if _, err := buf.ReadFrom(r); err != nil {
        return nil, err
    }
    return buf.Bytes(), nil
}

func (g *GzipCodec) Name() string { return "gzip" }

// In RecordBatch.Encode() modification
func (b *RecordBatch) Encode() ([]byte, error) {
    // 1. Serialize records to buffer (as before)
    // 2. Check if compression configured
    if b.Attributes & CompressionMask != CompressionNone {
        // 3. Get appropriate codec
        codec := compression.GetCodec(b.Attributes)
        compressed, err := codec.Compress(buffer.Bytes())
        if err != nil {
            return nil, err
        }
        // 4. Update buffer with compressed data
        buffer.Reset()
        buffer.Write(compressed)
    }
    // 5. Continue with CRC and final encoding
}
```

Transaction Coordinator Skeleton

```go
// internal/transaction/coordinator.go

package transaction


type TransactionCoordinator struct {

    brokerID      int32

    logManager    *LogManager

    pendingTxns   map[string]*TransactionMetadata // key: transactionalID

    producerState map[string]*ProducerState       // key: transactionalID

    mu            sync.RWMutex

}


type TransactionMetadata struct {

    TransactionalID    string

    ProducerID         int64

    ProducerEpoch      int16

    State              TransactionState

    Partitions         []TopicPartition

    TimeoutMs          int32

    LastUpdateTime     time.Time

}


// HandleInitProducerId processes request for new producer ID

func (tc *TransactionCoordinator) HandleInitProducerId(

    transactionalID string,

    timeoutMs int32,

) (producerID int64, producerEpoch int16, err error) {

    // TODO 1: Check if transactionalID already exists in producerState

    // TODO 2: If exists, validate epoch for fencing (reject if old epoch)

    // TODO 3: Generate new producerID (monotonically increasing)

    // TODO 4: Initialize new transaction metadata with state EMPTY

    // TODO 5: Write to transaction log for durability

    // TODO 6: Return producerID and epoch

}
```

```
// HandleAddPartitions processes request to add partitions to transaction
func (tc *TransactionCoordinator) HandleAddPartitions(
    transactionalID string,
    partitions []TopicPartition,
) error {
    // TODO 1: Look up transaction metadata
    // TODO 2: Validate producer epoch matches
    // TODO 3: Ensure transaction is in ONGOING state
    // TODO 4: Add partitions to metadata (deduplicate)
    // TODO 5: Write updated metadata to transaction log
}

// HandleEndTransaction processes commit/abort request
func (tc *TransactionCoordinator) HandleEndTransaction(
    transactionalID string,
    commit bool,
) error {
    // TODO 1: Look up transaction metadata
    // TODO 2: Validate producer epoch
    // TODO 3: Prepare PREPARE_COMMIT or PREPARE_ABORT marker
    // TODO 4: Write PREPARE marker to transaction log (fsync)
    // TODO 5: For each partition in transaction:
    //   - Write transaction marker (COMMIT/ABORT) to partition log
    // TODO 6: After all markers written, write COMPLETE to transaction log
    // TODO 7: Clean up transaction state
}
```

Language-Specific Hints for Go

- **Compression**: Use `compress/gzip` for simplicity, but consider `github.com/golang/snappy` for better speed if needed.
- **Concurrent Compaction**: Use `sync.RWMutex` in `Log` with `TryLock()` to skip compaction if log is busy.
- **Token Bucket Quotas**: Implement with `time.Ticker` refilling tokens and `sync/atomic` for token count.
- **Transaction Recovery**: On coordinator startup, scan transaction log to rebuild in-memory state of incomplete transactions.
- **Tiered Storage**: Use `io.ReaderAt` interface for random access to remote objects; implement LRU cache with `container/list` and `map`.

Milestone Checkpoint for Compression Extension

Verification Test:

```bash
# Run compression integration test
cd project-root
go test ./internal/compression/... -v -run TestCompressionRatio

# Expected output should show:
# === RUN   TestCompressionRatio
#     compression_test.go:45: GZIP ratio: 85% (1500 bytes → 225 bytes)
# --- PASS: TestCompressionRatio (0.02s)

# Manual verification:
# 1. Start broker with compression enabled
# 2. Use producer with compression.type=gzip
# 3. Produce 1000 messages with repetitive JSON content
# 4. Check disk usage in log directory - should be ~15% of uncompressed size
# 5. Consume messages - should decompress transparently
```

Debugging Tips for Compression:

- **Symptom**: Consumer receives garbled data or fails to decode.
  - **Cause**: Compression attributes mismatched between producer and consumer.
  - **Diagnosis**: Check `RecordBatch.Attributes` bits in stored log segment (hex dump).
  - **Fix**: Ensure `DecodeRecordBatch()` reads compression bits and uses correct decompressor.
- **Symptom**: High CPU usage on producer.
  - **Cause**: Compressing very small batches (worse than no compression).
  - **Diagnosis**: Log batch sizes before compression.
  - **Fix**: Increase `batch.size` or set `compression.threshold` higher.

Each extension represents a significant engineering challenge that will deepen your understanding of distributed systems trade-offs. Start with compression (easiest) to gain confidence, then progress to more complex features like transactions. Remember that production systems evolve incrementally—each feature builds upon a solid foundation, which you now possess.

14. Glossary

> **Milestone(s):** 1, 2, 3, 4 (all foundational concepts)

This glossary defines the key terms, acronyms, and domain-specific vocabulary used throughout this design document. Building intuition for these concepts is essential for understanding the architecture and implementation of a distributed message queue.

14.1 Terms and Definitions

The following table provides an alphabetical reference of technical terms used in this project.

Term	Definition
Accumulator	A producer component that temporarily batches multiple records by their destination `TopicPartition` before sending them as a single network request. It improves throughput by amortizing the overhead of network round-trips and serialization.
Acks (Acknowledgments)	The producer's configured durability guarantee, represented by `AcksLevel`. `AcksNone` (0) means fire-and-forget; `AcksLeader` (1) waits for the partition leader to persist the record; `AcksAll` (-1) waits for all in-sync replicas (ISR) to acknowledge.
API Key	A numeric identifier in the wire protocol's `MessageHeader` that specifies the type of request (e.g., `ProduceRequest`, `FetchRequest`, `JoinGroupRequest`).
BaseOffset	The offset of the first record in a `LogSegment`. All offsets within the segment are relative to this base.
Broker	A server node in the cluster that stores partition replicas and handles client requests. A broker hosts a subset of partitions for various topics, acting as both a storage layer and a network endpoint.
Compaction	See **Log Compaction**.
Consumer	A client application that subscribes to and reads records from topics. Consumers belong to a `ConsumerGroup` to coordinate parallel consumption.
Consumer Group	A set of consumers that cooperate to consume a topic. The group coordinator assigns each partition of a subscribed topic to exactly one member of the group, enabling parallel consumption while preserving ordering guarantees within each partition.
Control Plane	The network path dedicated to coordination operations, such as consumer group management (`JoinGroup`, `SyncGroup`, `Heartbeat`), metadata propagation, and leadership elections. This is distinct from the **Data Plane** which carries the actual message flow.
Controller	A designated broker (with the lowest or elected ID) that manages partition leadership elections and replica assignments for the entire cluster. It acts as the centralized decision-maker for partition state changes.
Coordination Service	A service (like the in-memory `MemoryCoordinator` or an external system like ZooKeeper) responsible for maintaining and distributing cluster metadata (brokers, topics, partition leadership) and enabling consensus for operations like leader election.
Correlation ID	A client-generated integer included in every request's `MessageHeader`. The server echoes it back in the corresponding response, allowing the client to match asynchronous requests with their responses.
Data Plane	The network path dedicated to the actual flow of messages, including produce requests (writes) and fetch requests (reads). This is distinct from the **Control Plane** used for coordination.
Exactly-once Semantics	A delivery guarantee where each message is processed exactly once, even in the face of producer retries, consumer restarts, or broker failures. This typically requires idempotent producers and transactional commits.
Follower Syncer	A component running on a broker that continuously fetches new records from a partition leader (via `FetchReplica` requests) to keep its local replica synchronized. Each follower maintains its own `fetchOffset`.
Generation ID	A monotonically increasing number (`GenerationID` in `ConsumerGroup`) that identifies the current epoch of a consumer group. It is incremented on each successful rebalance and used to fence out stale members from previous generations.
Group Coordinator	A specific broker (elected per consumer group) that manages consumer group membership, heartbeats, partition assignment, and offset commits for that group. It runs the group membership protocol.

Term	Definition
High Watermark	The offset of the last record that has been successfully replicated to all In-Sync Replicas (ISR). Consumers can only read up to the high watermark; records beyond it are considered "uncommitted" and could be lost if the leader fails.
Idempotent Producer	A producer configured to use sequence numbers (`ProducerID` , `ProducerEpoch` , `BaseSequence`) per partition to detect and discard duplicate batches caused by retries, enabling at-least-once semantics without duplicates.
Incremental Fetch	An optimization in the `FetchRequest` protocol where consumers only list partitions that have changed since their last request, reducing the size of requests when most partitions have no new data.
Index (Sparse Index)	A file (`IndexFile`) associated with a log segment that maps some record offset deltas to their byte positions in the data file. It is "sparse" because it does not index every record, trading some precision for smaller size.
Internal Topic	A special topic used by the system for its own metadata storage. In this project, `__consumer_offsets` is an example used to persistently store consumer group offset commits.
ISR (In-Sync Replica)	The set of replicas (leader and followers) for a partition that are fully caught up with the leader, defined as replicas whose `lastFetchOffset` is within a configured lag threshold (`replica.lag.time.max.ms`) of the leader's log end offset.
ISR Manager	A component running on the partition leader that tracks the progress (`followerStatus`) of each follower replica and periodically evaluates which replicas belong in the ISR, removing those that have fallen too far behind.
ISR Shrinkage	The process of removing followers from the in-sync replica set when they fail to keep up with the leader's write rate (exceeding the `replica.lag.time.max.ms` threshold). Excessive shrinkage can reduce replication factor and durability.
Leader Epoch	A monotonically increasing number (`LeaderEpoch`) associated with each leadership term for a partition. It is used by followers to detect and recover from stale or duplicate data after a leader change, serving as a fencing mechanism.
Leader-Follower Replication	The primary replication model where each partition has one designated leader that handles all produce/fetch requests, and one or more followers that asynchronously replicate data from the leader.
Length-prefixed	A message format where the first field is an integer (typically 4 bytes) specifying the total size of the remaining message bytes. This allows the receiver to read the complete message without parsing its internal structure first.
Log Compaction	A background process (`LogCleaner`) that removes older records for the same key from a log, retaining only the latest value. This reduces storage usage for keyed topics where only the current state matters (e.g., database change logs).
Log End Offset (LEO)	The offset of the next message that will be appended to the log (i.e., one greater than the offset of the last written record). For followers, this is their local `fetchOffset` .
Metadata Coordinator	The component (often the same as the **Controller**) responsible for maintaining and distributing cluster metadata—broker registrations, topic configurations, partition leader assignments, and ISR sets.
Multiplexing	Carrying multiple logical request/response streams over a single physical TCP connection. In this system, a producer or consumer may have one connection to a broker but send many independent requests with different `CorrelationID` s.
Nullable String	A string encoding in the wire protocol where a length prefix of `-1` indicates a `null` value, while non-negative lengths indicate a valid string of that length. This is distinct from zero-length strings.
Offset	A monotonically increasing integer assigned to each record within a partition, representing its immutable position in the ordered sequence. Offsets start at 0 and are contiguous.

Term	Definition
Offset Commit	The action by a consumer to persistently store its current read position (offset) for each assigned partition, typically to an internal topic (`__consumer_offsets`). This allows the consumer to resume from that point after a restart.
Partition	An ordered, immutable sequence of records that is a subset of a topic. Partitions enable horizontal scaling—different partitions of the same topic can be hosted on different brokers and consumed in parallel.
Partition Assignment	The mapping of partitions to consumers within a consumer group, determined by the group coordinator using a strategy like **Range** or **RoundRobin**. Each consumer receives an `Assignment` describing the partitions it should consume.
Partitioner	A component (`HashPartitioner` , `RoundRobinAssigner`) that selects the target partition for a record based on its key (if present) or a round-robin scheme (if key is `nil`). Ensures records with the same key go to the same partition.
Producer	A client that publishes (writes) records to topics. It batches records, selects target partitions, handles retries, and respects acknowledgment configurations.
Quota Enforcement	Limiting resource usage (e.g., bytes per second) per client using algorithms like the **Token Bucket** to prevent a single misbehaving client from overwhelming the broker.
Rebalance	The process of redistributing partitions among the members of a consumer group when membership changes (a consumer joins, leaves, or fails). During a rebalance, the group coordinator reassigns partitions and informs all members.
Rebalance Storm	A pathological condition where consumer group members frequently join and leave, triggering continuous rebalances and preventing the group from settling into a stable state. Often caused by misconfigured `session.timeout.ms` or network issues.
Record	The fundamental unit of data, consisting of a key, value, headers, and metadata (timestamp, attributes). Records are grouped into **RecordBatch**es for efficient storage and transmission.
Record Batch	A group of records written together as a single unit on disk and over the network. Batches include metadata like `BaseOffset` , `FirstTimestamp` , and `CRC` for integrity. They are the atomic unit of writing and replication.
Replication Plane	The network path dedicated to data replication between brokers (leader to follower via `FetchReplica` requests). This traffic is separate from client produce/fetch traffic to avoid interference.
Segment (Log Segment)	A physical file (`DataFile`) containing a contiguous range of log records, along with its accompanying index file (`IndexFile`). Segments are rolled when they reach a configured maximum size (`SegmentMaxBytes`).
Sender	A producer component that manages network connections to brokers, sends accumulated batches, handles retries with exponential backoff, and processes acknowledgment responses.
Session Timeout	The time (`session.timeout.ms`) after which the group coordinator considers a consumer dead if it hasn't received a heartbeat. This triggers a rebalance to reassign the dead consumer's partitions.
Sparse Index	See **Index**.
Tiered Storage	A hierarchical storage architecture where older log segments are moved from fast, expensive local storage (`StorageTierLocal`) to slower, cheaper remote storage (`StorageTierRemote`) like object storage, managed by a `TierManager` .
Token Bucket	An algorithm (`TokenBucket`) used for rate limiting (**Quota Enforcement**). Tokens are added at a fixed rate (`refillRate`) up to a `capacity` . Each request consumes tokens; if insufficient tokens are available, the request is delayed.
Topic	A named stream of records, divided into one or more partitions. Topics are the primary abstraction for categorization—producers write to a topic, and consumers subscribe to one or more topics.

Term	Definition
Two-phase Commit	A distributed transaction protocol used by `TransactionCoordinator` to ensure atomic commit across multiple partitions. It involves a prepare phase (where participants vote) and a commit/abort phase.
Unclean Leader Election	Electing a leader from outside the current ISR (when `allow.unclean.leader.election` is enabled). This risks data loss because the new leader may not have all committed records, but it improves availability when the ISR is empty.
Watch Pattern	A callback-based notification mechanism where components can register interest (`WatchBrokers`, `WatchTopic`) in changes to cluster metadata. The coordination service invokes callbacks when the watched data changes.
Wire Protocol	The binary format for network communication between components (clients and brokers, brokers and brokers). It defines how requests and responses are serialized into bytes for transmission over TCP.
Zero-copy	A technique for data transfer where data is moved between buffers (e.g., from disk to network) without CPU copying, often using OS-level features like `sendfile`. This reduces CPU overhead for high-throughput systems.
Zombie Consumer	A consumer that is considered dead by the group coordinator (due to heartbeat timeout) but is still alive and processing messages. This can lead to duplicate processing if partitions are reassigned while the zombie is still active.