Message Queue: Design Document

Overview

This document outlines the design of an in-memory message queue supporting publish/subscribe, point-to-point delivery, and consumer groups. The key architectural challenge is building a reliable async communication backbone that handles message ordering, delivery guarantees, and failure recovery while maintaining high throughput.

> 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.

Context and Problem Statement

> **Milestone(s):** All milestones (1-4) — foundational concepts that inform the entire system design

Mental Model: The Digital Postal System

Before diving into the technical details of message queues, let's build intuition using a familiar analogy: the postal system. A message queue is fundamentally like a sophisticated digital post office that handles mail between applications instead of people.

In the physical world, when you send a letter, several things happen automatically: the postal service receives your mail, sorts it by destination, delivers it to the correct mailbox, and potentially provides delivery confirmation. The recipient doesn't need to be home when you send the letter — the postal service holds it until they're ready to collect it. Multiple people can send letters to the same address, and the postal service handles the routing complexity.

Message queues provide the same asynchronous communication benefits for software systems. When Application A needs to send data to Application B, it doesn't make a direct phone call (synchronous communication). Instead, it drops a message into the queue system, trusting that the message will eventually reach its destination even if Application B is temporarily unavailable. This **decoupling** is the fundamental value proposition — senders and receivers can operate independently, at different speeds, and with different availability patterns.

The postal analogy extends further when we consider different delivery patterns. Sometimes you want to send a personal letter to one specific person (point-to-point messaging). Other times you want to send a newsletter to all subscribers (publish-subscribe messaging). The postal system handles both patterns, just as our message queue will need to support multiple messaging models.

Consider the complexity that emerges in large organizations. A company might have thousands of employees sending and receiving mail daily. Without a centralized postal system, every employee would need to personally deliver messages to every other employee — clearly unscalable. Similarly, in distributed systems with dozens or hundreds of services, direct service-to-service communication creates an unmanageable web of dependencies. A message queue acts as the central postal service, allowing services to communicate through a well-defined, scalable intermediary.

The reliability aspects of postal systems also inform message queue design. Critical mail gets delivery confirmation, tracking numbers, and insurance against loss. Similarly, our message queue will need acknowledgment protocols, message persistence, and failure recovery mechanisms. When a letter can't be delivered (wrong address, recipient moved), it goes to a dead letter office for manual handling — our system will implement the same pattern with dead letter queues.

Messaging Patterns

Understanding the different messaging patterns is crucial for designing a flexible message queue system. Let's examine the two primary patterns our system will support, continuing with postal analogies to build intuition before formalizing the technical requirements.

Publish-Subscribe Pattern (Fan-out Delivery)

The publish-subscribe pattern works like a newspaper or magazine subscription service. When a publisher creates content, it gets delivered to all current subscribers automatically. The publisher doesn't need to know who the subscribers are or how many there are — the distribution system handles that complexity.

In our message queue context, a **publisher** sends a message to a named **topic** (like "user-registration-events" or "payment-processed"). All applications that have **subscribed** to that topic receive a copy of every message. This creates a **fan-out** delivery pattern where one message becomes many deliveries.

Consider a practical example: when a user registers on an e-commerce platform, the registration service publishes a "user-registered" event. Multiple downstream services need to react: the email service sends a welcome email, the analytics service records the signup, the recommendation engine initializes user preferences, and the marketing system updates campaign metrics. With publish-subscribe, the registration service sends one message, and all interested parties receive it automatically.

Aspect	Description	Postal Analogy
Publisher	Application that sends messages to a topic	Magazine publisher
Topic	Named channel for related messages	Magazine title/subject
Subscriber	Application that receives all messages for a topic	Magazine subscriber
Message	Data payload sent to the topic	Individual magazine issue
Fan-out	One message delivered to multiple subscribers	One print run delivered to all subscribers

The key characteristics of publish-subscribe messaging include:

Temporal Decoupling: Publishers and subscribers don't need to be online simultaneously. Messages are held by the queue system until subscribers are ready to process them.

Spatial Decoupling: Publishers don't need to know the network location or identity of subscribers. The topic name provides the only coupling point.

Synchronization Decoupling: Publishers continue their work immediately after sending a message, without waiting for subscriber processing to complete.

Point-to-Point Pattern (Work Distribution)

The point-to-point pattern resembles a work distribution system where tasks are assigned to individual workers. Unlike publish-subscribe where every subscriber gets every message, point-to-point ensures that each message is processed by exactly one consumer.

Think of a customer service call center. When customers call, their calls go into a queue and are distributed among available agents. Each call is handled by exactly one agent, not broadcast to all agents. This prevents duplicate work and enables horizontal scaling — adding more agents increases the call handling capacity proportionally.

In our message queue, this pattern is implemented through **consumer groups**. Multiple consumer instances join a named group (like "payment-processor-workers"). When messages arrive for that group, the queue system distributes them among the available group members using a round-robin or other assignment strategy.

Aspect	Description	Postal Analogy
Producer	Application that sends work messages	Customer mailing support requests
Queue	Named destination for work items	Post office box for support department
Consumer Group	Set of workers processing the same type of task	Support team members
Message	Individual work item or task	Individual support request
Load Balancing	Distributing messages across group members	Distributing requests among team members

Consumer groups enable **horizontal scaling** — if message processing can't keep up with the incoming rate, administrators can launch additional consumer instances. The queue system automatically includes new members in the work distribution rotation.

Consider an image processing service. The web application receives photo uploads and sends processing requests to a queue. Multiple image processing workers (consumer group members) pull work from the queue. During high traffic periods, additional workers can be started to handle the load. During quiet periods, workers can be shut down to save resources.

Hybrid Patterns and Practical Applications

Real-world applications often combine both patterns. A user registration event might trigger both fan-out notification (publish-subscribe) and specific work tasks (point-to-point). Our message queue design must support both patterns simultaneously, often for the same underlying message flow.

For example, when an order is placed:

1. The order service publishes an "order-created" event (pub-sub) for analytics, inventory, and notification systems
2. The order service also sends a "process-payment" work item (point-to-point) to the payment processing worker pool
3. After payment completion, a "payment-processed" event (pub-sub) triggers fulfillment, confirmation emails, and accounting updates

Existing Solutions Comparison

Understanding the landscape of existing message queue solutions helps inform our design decisions and sets expectations for the capabilities we're building. Let's examine three representative systems that span the spectrum from simple pub-sub to enterprise-grade streaming platforms.

RabbitMQ: Enterprise Message Broker

RabbitMQ represents the traditional message broker approach, implementing the Advanced Message Queuing Protocol (AMQP). It provides sophisticated routing capabilities through a system of **exchanges**, **queues**, and **bindings** that determine how messages flow through the system.

Architecture Approach: RabbitMQ uses a hub-and-spoke model where producers send messages to exchanges, which route them to queues based on binding rules. Consumers attach to specific queues to receive messages. This indirection layer enables complex routing patterns like content-based routing and multi-step message transformation.

Capability	RabbitMQ Implementation	Our Queue Target
Message Persistence	Optional, with durable queues and persistent messages	Always persisted to append-only log
Delivery Guarantees	At-least-once with manual ACK, exactly-once in transactions	At-least-once with ACK/NACK support
Consumer Groups	Work queues with round-robin distribution	Consumer groups with rebalancing
Topic Patterns	Topic exchanges with wildcard routing	Hierarchical topics with * and # wildcards
Administrative Tools	Web-based management UI with detailed metrics	REST API for monitoring and administration
Clustering	Multi-node clusters with master election	Single-node (future: clustering support)
Protocol	AMQP 0.9.1 binary protocol	Custom binary TCP protocol

Strengths: RabbitMQ excels at complex routing scenarios and provides excellent tooling for operations teams. Its exchange system enables sophisticated message transformation and routing logic. The management interface provides detailed visibility into queue depths, message rates, and consumer behavior.

Limitations: The complexity of exchanges and bindings can be overwhelming for simple use cases. RabbitMQ's performance characteristics favor message delivery guarantees over raw throughput, making it less suitable for high-volume streaming scenarios.

Apache Kafka: Distributed Streaming Platform

Apache Kafka represents a fundamentally different approach, designed for high-throughput streaming and event sourcing scenarios. Rather than traditional message queues, Kafka provides distributed **logs** that clients read from specific positions.

Architecture Approach: Kafka organizes messages into **topics** divided into **partitions**. Each partition is an ordered, immutable sequence of messages. Consumers track their position in each partition, enabling replay of historical messages and parallel processing within consumer groups.

Capability	Kafka Implementation	Our Queue Target
Message Model	Immutable log with offset-based consumption	Queue with ACK-based consumption
Partitioning	Topic partitions for parallel processing	Single queue per topic (simpler model)
Consumer Groups	Partition assignment with rebalancing protocol	Round-robin message assignment
Message Retention	Time and size-based with log compaction	Time and size-based retention policies
Delivery Semantics	At-least-once, at-most-once, or exactly-once	At-least-once with duplicate detection
Throughput	Optimized for high-volume streaming (millions/sec)	Moderate throughput (thousands/sec target)
Protocol	Custom binary protocol over TCP	Custom binary protocol over TCP

Strengths: Kafka's log-based model enables extremely high throughput and supports complex stream processing scenarios. The ability to replay historical messages makes it excellent for event sourcing and building derived data systems. Consumer groups can dynamically rebalance when members join or leave.

Limitations: Kafka's complexity and operational overhead make it overkill for simple messaging scenarios. The log-based model requires careful partition key design to maintain message ordering. Setting up and tuning Kafka clusters requires significant expertise.

Redis Pub/Sub: Lightweight Messaging

Redis pub/sub provides a minimalist approach to messaging, built on top of Redis's in-memory data structure server. It offers basic publish-subscribe functionality without persistence or delivery guarantees.

Architecture Approach: Redis pub/sub uses a simple broadcast model where publishers send messages to named channels and all subscribers receive them immediately. There are no queues, persistence, or acknowledgment mechanisms — messages exist only in flight.

Capability	Redis Pub/Sub Implementation	Our Queue Target
Message Persistence	None — messages only exist in memory	Persistent append-only log
Delivery Guarantees	Fire-and-forget, no ACK mechanism	At-least-once with ACK/NACK
Consumer Groups	Pattern-based subscriptions only	Named consumer groups with load balancing
Message Ordering	No guarantees across channels	FIFO ordering within topics
Failure Recovery	No recovery — disconnected clients lose messages	Automatic redelivery and DLQ handling
Operational Complexity	Very low — single Redis instance	Low — single broker process
Protocol	Redis protocol (RESP)	Custom binary protocol

Strengths: Redis pub/sub is extremely simple to deploy and use. It provides excellent performance for scenarios where message loss is acceptable, such as real-time notifications or cache invalidation signals.

Limitations: The lack of persistence and delivery guarantees makes Redis pub/sub unsuitable for critical business data. There's no built-in support for consumer groups or load balancing among multiple consumers.

Design Implications for Our Message Queue

This analysis reveals several key design decisions for our message queue implementation:

Decision: Balanced Feature Set

- **Context**: We need to provide more reliability than Redis pub/sub but avoid the complexity of Kafka's partitioning model
- **Options Considered**:
  1. Minimal Redis-like approach with just pub/sub
  2. Full Kafka-like streaming platform with partitions
  3. RabbitMQ-like broker with exchanges and routing
  4. Balanced approach with topics, consumer groups, and persistence
- **Decision**: Balanced approach combining reliable delivery with operational simplicity
- **Rationale**: Learning objectives require understanding core messaging concepts (ACK/NACK, consumer groups, persistence) without overwhelming complexity
- **Consequences**: More sophisticated than Redis but simpler to operate than Kafka or RabbitMQ

The comparison also highlights several **non-functional requirements** our system should meet:

Reliability: Unlike Redis pub/sub, we must provide message persistence and delivery guarantees. This requires implementing acknowledgment protocols and handling failure scenarios.

Scalability: While we won't match Kafka's throughput, we should support horizontal scaling through consumer groups and handle reasonable message volumes without degradation.

Operational Simplicity: Unlike enterprise message brokers, our system should be easy to deploy and monitor, with minimal configuration required for basic usage.

Protocol Efficiency: All three systems use binary protocols for performance. Our custom protocol should be efficient while remaining simple enough for educational purposes.

Understanding these existing solutions provides context for the design choices we'll make throughout the system architecture. Each component we build addresses real-world requirements that these production systems have validated through years of operational experience.

Implementation Guidance

This foundational section establishes the context and mental models that inform all subsequent design decisions. While there's no code to implement yet, understanding the problem space and existing solutions is crucial before diving into architectural details.

Technology Stack Recommendations

Based on our analysis of existing message queue systems and the learning objectives, here are the recommended technology choices for implementing our message queue:

Component	Simple Option	Advanced Option	Rationale
Language	Go with standard library	Go with specialized libraries	Go's excellent concurrency primitives and standard library support for TCP and binary protocols
Protocol	Custom binary over TCP (net package)	Protocol Buffers over gRPC	Binary protocols are more efficient; custom design teaches protocol fundamentals
Persistence	Direct file I/O (os package)	Embedded database (BadgerDB/BoltDB)	Direct file control teaches persistence concepts; databases abstract too much
Serialization	encoding/gob for simplicity	JSON for debugging, MessagePack for efficiency	Built-in gob is sufficient for learning; avoids external dependencies
Concurrency	sync package primitives	Channel-based actor model	Standard mutexes and channels teach fundamental concurrency patterns
Testing	Standard testing package	Testify for assertions	Standard library encourages understanding test fundamentals

Project Structure Foundation

Even though we haven't started coding yet, establishing a clear project structure from the beginning prevents architectural debt and supports the learning process. Here's the recommended directory layout that will accommodate all four milestones:

```
message-queue/
  cmd/
    broker/
      main.go                 ← Broker server entry point
    client/
      main.go                 ← Example client for testing
  internal/
    protocol/
      protocol.go             ← Binary protocol definitions (Milestone 1)
      connection.go           ← Connection management
    topic/
      manager.go              ← Topic management and routing (Milestone 1)
      subscriber.go           ← Subscriber tracking
    consumer/
      coordinator.go          ← Consumer group coordination (Milestone 2)
      rebalancer.go           ← Rebalancing algorithm
    ack/
      tracker.go              ← Acknowledgment and redelivery (Milestone 2)
      timeout.go              ← Timeout management
    persistence/
      wal.go                  ← Write-ahead log (Milestone 3)
      recovery.go             ← Crash recovery
    backpressure/
      controller.go           ← Flow control (Milestone 3)
    dlq/
      manager.go              ← Dead letter queue (Milestone 4)
    monitoring/
      metrics.go              ← Metrics collection (Milestone 4)
      api.go                  ← REST API endpoints
  pkg/
    client/
      client.go               ← Client library for applications
  configs/
    broker.yaml               ← Configuration file
  docs/
    protocol.md               ← Protocol specification
    api.md                    ← REST API documentation
  scripts/
    start-broker.sh           ← Development convenience scripts
  test/
    integration/              ← End-to-end tests
  go.mod
  go.sum
  README.md
```

This structure separates concerns cleanly:

- `cmd/` contains executable entry points
- `internal/` contains broker implementation details (not importable by external packages)
- `pkg/` contains reusable client libraries
- Each major component gets its own package under `internal/`
- Configuration, documentation, and testing infrastructure have dedicated locations

Learning Path Overview

Understanding where this project fits in the broader landscape of distributed systems concepts helps contextualize the learning journey:

Prerequisites Validation: Before starting implementation, ensure comfort with these foundational concepts:

- TCP socket programming: understanding of connection lifecycle, read/write operations, and handling network partitions
- Go concurrency: goroutines, channels, sync.Mutex, sync.RWMutex for managing shared state
- Data structures: implementing queues, hash maps, and understanding their performance characteristics

Conceptual Learning Progression: Each milestone builds conceptual complexity:

1. **Network Protocol Design** (Milestone 1): Learn how distributed systems communicate reliably over unreliable networks
2. **Coordination Algorithms** (Milestone 2): Understand how multiple processes can cooperate effectively
3. **Durability and Recovery** (Milestone 3): Learn how systems survive failures and maintain data integrity
4. **Observability and Operations** (Milestone 4): Understand how production systems are monitored and debugged

Connections to Broader Distributed Systems: The patterns learned here apply to many other systems:

- **Database replication** uses similar acknowledgment and consensus patterns
- **Microservice coordination** relies heavily on asynchronous messaging concepts
- **Stream processing systems** build on the foundation of reliable message delivery
- **Cache invalidation** often uses pub/sub patterns for coordination

Common Learning Pitfalls

Based on experience teaching message queue concepts, here are the most frequent misconceptions and how to avoid them:

⚠️ **Pitfall: Underestimating Network Unreliability** Many learners assume TCP provides more guarantees than it actually does. TCP ensures ordered, reliable delivery of bytes between two endpoints, but doesn't handle application-level concerns like partial message reads, connection failures during processing, or ensuring the remote application actually processed the data.

How to avoid: Always implement proper message framing (length prefixes), heartbeat mechanisms, and application-level acknowledgments. Never assume a successful TCP write means the remote application processed your message.

⚠️ **Pitfall: Ignoring Message Ordering Constraints**
It's tempting to use concurrent processing everywhere for performance, but message ordering requirements often limit parallelization options. Understanding when ordering matters and designing appropriate concurrency patterns is crucial.

How to avoid: Clearly define ordering requirements for each message type. Use per-topic sequential processing when ordering matters, and consumer groups only when messages can be processed in parallel.

⚠️ **Pitfall: Oversimplifying Failure Scenarios** Message queues operate in a world of partial failures — network timeouts, process crashes, disk errors, and slow consumers. Designing only for the happy path creates brittle systems.

How to avoid: For each component, explicitly list what can fail, how failures are detected, and what recovery actions are taken. Test failure scenarios regularly, not just successful message delivery.

⚠️ **Pitfall: Conflating Different Delivery Semantics** "At-least-once", "at-most-once", and "exactly-once" delivery have precise meanings with significant implementation implications. Mixing these concepts leads to systems that make inconsistent guarantees.

How to avoid: Choose one delivery semantic for your initial implementation and implement it correctly. Understand the trade-offs — at-least-once requires deduplication handling, at-most-once risks message loss, exactly-once requires distributed transactions.

Success Metrics and Validation

To validate understanding as you progress through the implementation:

Conceptual Understanding Checkpoints:

- Can you explain the difference between pub/sub and point-to-point messaging using examples from your own experience?
- Can you describe why acknowledgments are necessary and what happens without them?
- Can you identify the trade-offs between different delivery semantics in concrete scenarios?

Implementation Milestones:

- After Milestone 1: Two clients can exchange messages through your broker with proper connection handling
- After Milestone 2: Multiple consumers can join a group and receive balanced work distribution
- After Milestone 3: The broker survives crashes and recovers all unacknowledged messages
- After Milestone 4: You have visibility into system behavior and can diagnose common problems

Real-World Application:

- Can you identify where message queues would improve an existing system you've worked on?

- Can you design a messaging architecture for a specific use case (e.g., e-commerce order processing)?
- Can you explain the trade-offs between using your message queue versus Redis pub/sub or a managed service like AWS SQS?

The goal isn't just to build a working message queue, but to deeply understand the distributed systems principles that make reliable asynchronous communication possible. These concepts will serve you well in any system that needs to coordinate work across multiple processes or machines.

Goals and Non-Goals

> **Milestone(s):** All milestones (1-4) — defines the fundamental scope and boundaries that guide implementation decisions across the entire project

Mental Model: The Project Charter

Think of this section as your **project charter** — a formal document that prevents scope creep and ensures everyone understands what success looks like. Just as a construction project needs blueprints that specify exactly what will be built (and what won't), our message queue needs clear boundaries. Without explicit goals, you might spend weeks implementing advanced clustering features when the real learning objective is understanding basic acknowledgment patterns. This charter becomes your North Star when you face implementation decisions: "Should I add this feature?" Check the charter. "Is this performance level acceptable?" Check the charter.

The goals section serves three critical purposes: it defines **functional requirements** (what the system must do), **non-functional requirements** (how well it must do it), and **explicit non-goals** (what we deliberately won't build). This tri-part structure prevents both under-engineering (missing essential capabilities) and over-engineering (building unnecessary complexity).

Functional Requirements

Functional requirements define the **core capabilities** our message queue must provide. These map directly to the four milestone deliverables, ensuring our scope aligns with the learning objectives. Each requirement includes specific **acceptance criteria** that determine when the feature is complete.

Requirement ID	Capability	Acceptance Criteria	Milestone
FR-1	**Topic-based Messaging**	Support named topics for message organization. Publishers send to topics, subscribers receive from topics. Topic names are hierarchical (e.g., `orders.payments.completed`)	1
FR-2	**Publish-Subscribe Pattern**	Single message published to topic reaches all active subscribers (fan-out delivery). Messages delivered in FIFO order per topic	1
FR-3	**Binary Wire Protocol**	TCP-based binary protocol supporting PUBLISH, SUBSCRIBE, ACK, NACK commands. Handle partial TCP reads and connection lifecycle	1
FR-4	**Consumer Groups**	Named consumer groups distribute messages round-robin across group members. Only one member receives each message	2
FR-5	**Message Acknowledgment**	Messages require explicit ACK for completion. Unacknowledged messages redelivered after configurable timeout	2
FR-6	**Consumer Group Rebalancing**	Automatic work redistribution when consumers join or leave groups. No message loss during rebalancing	2
FR-7	**Message Persistence**	Messages persisted to append-only log before delivery confirmation. Support crash recovery and replay	3
FR-8	**Backpressure Control**	Producer throttling when consumer lag exceeds threshold. Prevents memory exhaustion from slow consumers	3
FR-9	**Dead Letter Queue**	Messages exceeding max retry count moved to DLQ. Support inspection and replay of poison messages	4
FR-10	**Monitoring API**	REST endpoints exposing queue depth, consumer lag, throughput, and error rates per topic	4

> **Design Insight**: Notice how requirements build progressively — you can't implement consumer group rebalancing (FR-6) without first having consumer groups (FR-4) and acknowledgments (FR-5). This dependency structure mirrors the milestone progression, ensuring each stage provides a solid foundation for the next.

Message Delivery Guarantees represent our most critical functional requirement. The system must provide **at-least-once delivery** semantics, meaning messages are delivered one or more times but never lost. This choice reflects real-world messaging system requirements where duplicate handling is easier than data loss recovery.

Delivery Scenario	Guarantee	Implementation Mechanism
Normal Operation	Exactly-once delivery	Message delivered once, ACK received, message marked complete
Consumer Crash	At-least-once delivery	Redelivery after ACK timeout, consumer may receive duplicate
Broker Crash	At-least-once delivery	WAL replay on restart ensures unacknowledged messages redelivered
Network Partition	At-least-once delivery	Timeout triggers redelivery when ACK lost in network

Wildcard Subscription Support enables powerful routing patterns essential for real-world messaging systems. Subscribers can register for topic patterns rather than specific topic names, enabling flexible message routing without tight coupling between producers and consumers.

Wildcard Type	Symbol	Example Pattern	Matches	Does Not Match
Single Level	`*`	`orders.*.created`	`orders.payments.created`, `orders.shipping.created`	`orders.created`, `orders.payments.items.created`
Multi Level	`#`	`orders.#`	`orders.created`, `orders.payments.created`, `orders.payments.items.created`	`inventory.created`

Non-Functional Requirements

Non-functional requirements define **how well** the system must perform its functional capabilities. These requirements establish measurable targets that guide implementation decisions and testing criteria. Unlike functional requirements which define behavior, non-functional requirements define qualities like performance, reliability, and usability.

> **Performance Context**: Our targets reflect single-node, in-memory operation optimized for learning rather than production scale. These numbers provide concrete goals while remaining achievable on typical development hardware.

Performance Requirements establish minimum acceptable throughput and latency characteristics. These targets ensure the system demonstrates efficient concurrent processing while remaining realistic for educational implementation.

Metric	Target	Measurement Conditions	Rationale
Message Throughput	10,000 messages/second	1KB messages, 10 concurrent publishers, single topic	Demonstrates efficient concurrent processing
End-to-End Latency	< 10ms (p99)	Publisher to subscriber, normal operation	Ensures responsive real-time messaging
Consumer Group Rebalancing	< 5 seconds	10-member group, single member departure	Minimizes message delivery disruption
Memory Efficiency	1MB per 10,000 queued messages	Average 1KB message size, minimal overhead	Prevents excessive memory usage

Reliability Requirements ensure the system handles failures gracefully and maintains data integrity. These requirements guide error handling logic and recovery procedures throughout the implementation.

Reliability Aspect	Requirement	Success Criteria	Implementation Focus
Crash Recovery	Full state restoration within 30 seconds	All unacknowledged messages restored from WAL	Append-only log design and recovery procedures
Message Ordering	FIFO delivery per topic	Messages delivered in publish order within single topic	Queue-based storage and single-threaded delivery
Connection Handling	Graceful cleanup on disconnect	No memory leaks, pending messages requeued	Connection lifecycle management
Error Detection	Consumer failure detection within 60 seconds	Dead consumers removed from groups automatically	Heartbeat protocol and timeout monitoring

Operational Requirements define characteristics that make the system manageable and debuggable in practice. These requirements ensure the implementation supports effective troubleshooting and system observation.

Operational Aspect	Requirement	Implementation Approach
Configuration	File-based configuration for all tunable parameters	YAML/JSON config file with environment variable overrides
Logging	Structured logging at INFO, WARN, ERROR levels	JSON-formatted logs with correlation IDs
Monitoring	HTTP metrics endpoint exposing Prometheus format	Built-in metrics collection with `/metrics` endpoint
Administrative Interface	REST API for system inspection and management	HTTP API for topic listing, consumer groups, DLQ management

Explicit Non-Goals

Explicit non-goals prevent scope creep by clearly stating functionality we **deliberately exclude** from this implementation. These exclusions focus the learning experience on core messaging concepts rather than peripheral complexity.

> **Educational Focus**: Each non-goal represents a conscious trade-off prioritizing learning objectives over production completeness. Understanding what we won't build is as important as understanding what we will build.

Clustering and Distribution features require significant additional complexity that would overshadow the core messaging concepts. Multi-node coordination introduces consensus algorithms, partition management, and network failure handling that merit separate study.

Excluded Feature	Rationale	Alternative Approach
Multi-Broker Clustering	Requires consensus algorithms, leader election, partition replication	Single-node deployment with clear extension points for future clustering
Horizontal Scaling	Adds complexity of work distribution across nodes	Vertical scaling within single node, design patterns supporting future clustering
Cross-Datacenter Replication	Network partition handling, conflict resolution complexity	Single datacenter deployment model

Advanced Security Features represent important production concerns but add implementation complexity without advancing messaging system understanding. Basic connection security provides foundation for future enhancement.

Security Feature	Inclusion Status	Rationale
TLS/SSL Encryption	❌ Not Included	Focus on messaging logic, plaintext sufficient for learning
Authentication	❌ Not Included	Adds user management complexity beyond messaging concepts
Authorization	❌ Not Included	Topic-level permissions require separate access control system
Message Encryption	❌ Not Included	End-to-end encryption belongs in application layer

Decision: Single-Node In-Memory Architecture

- **Context**: Need to balance educational value with implementation complexity while demonstrating core messaging patterns
- **Options Considered**:
  - Distributed architecture with clustering
  - Persistent-only storage with disk-based queues
  - In-memory with optional persistence
- **Decision**: Single-node in-memory with WAL persistence for durability
- **Rationale**: Eliminates network coordination complexity while teaching persistence patterns. WAL provides crash recovery without distributed systems complexity
- **Consequences**: Enables focus on messaging patterns, limits scalability but provides clear extension path for clustering

Performance Optimization Features beyond basic efficiency would complicate the implementation without significantly advancing messaging system understanding. The goal is correctness and clarity over maximum performance.

Optimization	Inclusion Status	Learning Trade-off
Message Batching	❌ Not Included	Adds protocol complexity, masks individual message lifecycle
Compression	❌ Not Included	Focuses on encoding rather than messaging patterns
Zero-Copy Operations	❌ Not Included	Low-level optimization obscures high-level design
Custom Serialization	❌ Not Included	Standard formats adequate for demonstrating concepts

Enterprise Features common in production messaging systems require substantial additional infrastructure that would shift focus from core messaging concepts to enterprise integration patterns.

Enterprise Feature	Rationale for Exclusion	Core Concept Addressed Instead
Schema Registry	Requires separate versioning system	Focus on message routing and delivery
Message Transformation	Belongs in application logic, not broker	Emphasize clean producer/consumer separation
Complex Routing Rules	Content-based routing adds rule engine complexity	Topic-based routing with wildcards sufficient
Multi-Protocol Support	AMQP, STOMP, MQTT add protocol translation complexity	Single binary protocol demonstrates concepts clearly

Boundary Insight: These non-goals aren't "lesser" features — they're different learning objectives. A production messaging system absolutely needs security, clustering, and enterprise features. But studying those features would require a different project with different learning goals.

Compatibility Requirements with existing messaging systems would require protocol translation layers and format conversion logic that adds complexity without teaching messaging fundamentals.

Compatibility Target	Exclusion Rationale
AMQP Protocol	Standard protocol compliance requires extensive specification implementation
Kafka Wire Protocol	Existing protocol formats optimized for specific use cases, not learning
Redis Pub/Sub Commands	Command compatibility constrains design choices for educational clarity
RabbitMQ Exchange Types	Advanced routing patterns merit separate study after mastering basics

Implementation Guidance

This section provides practical guidance for translating the goals and requirements into working code, with specific recommendations for technology choices and project organization.

Technology Recommendations for implementing each requirement category:

Requirement Category	Simple Option	Advanced Option	Recommended for Learning
Wire Protocol	Text-based protocol with JSON	Binary protocol with custom framing	Binary (more realistic)
Persistence	Single log file with JSON records	Custom binary format with indexing	JSON initially, binary later
Concurrency	Mutex-based synchronization	Lock-free data structures	Mutex-based (clearer reasoning)
Configuration	Hardcoded constants	YAML file with validation	YAML file (operational realism)
Logging	Printf debugging	Structured logging with levels	Structured logging (debugging essential)
Monitoring	Manual state inspection	Metrics collection system	Metrics collection (observability crucial)

Project Structure supporting progressive milestone implementation:

```
message-queue/
├── cmd/broker/                 ← main entry point
│   └── main.go                ← server startup, config loading
├── internal/protocol/          ← Milestone 1: Wire protocol
│   ├── handler.go             ← TCP server and connection management
│   ├── messages.go            ← Command parsing and serialization
│   └── connection.go          ← Individual client connection handling
├── internal/topic/             ← Milestone 1: Topic management
│   ├── manager.go             ← Topic creation and message routing
│   ├── subscriber.go          ← Subscription management and fanout
│   └── wildcards.go           ← Pattern matching for topic subscriptions
├── internal/consumer/          ← Milestone 2: Consumer groups
│   ├── group.go               ← Consumer group coordination
│   ├── rebalancer.go          ← Work assignment and rebalancing
│   └── assignment.go          ← Round-robin and sticky assignment strategies
├── internal/ack/               ← Milestone 2: Acknowledgment tracking
│   ├── tracker.go             ← Message acknowledgment and timeout handling
│   ├── redelivery.go          ← Timeout detection and message reassignment
│   └── dlq.go                 ← Dead letter queue management
├── internal/persistence/       ← Milestone 3: WAL and recovery
│   ├── wal.go                 ← Write-ahead log implementation
│   ├── recovery.go            ← Crash recovery procedures
│   └── retention.go           ← Log cleanup and compaction
├── internal/backpressure/      ← Milestone 3: Flow control
│   ├── controller.go          ← Consumer lag monitoring and throttling
│   └── monitor.go             ← Lag calculation and threshold checking
├── internal/monitoring/        ← Milestone 4: Observability
│   ├── metrics.go             ← Metrics collection and HTTP endpoint
│   ├── admin.go               ← Administrative API for inspection
│   └── health.go              ← Health checks and consumer heartbeat
├── config/
│   └── broker.yaml            ← Configuration file with all tunable parameters
└── test/
    ├── integration/           ← End-to-end test scenarios
    └── benchmarks/            ← Performance verification
```

Core Configuration Structure enabling requirement-driven parameter tuning:

```go
// Configuration struct supporting all non-functional requirements
type BrokerConfig struct {
    // Performance tuning
    MaxConcurrentConnections int           `yaml:"max_connections"`
    MessageBufferSize       int           `yaml:"message_buffer_size"`

    // Reliability parameters
    AckTimeoutSeconds      int           `yaml:"ack_timeout_seconds"`
    ConsumerTimeoutSeconds int           `yaml:"consumer_timeout_seconds"`
    MaxRetryCount          int           `yaml:"max_retry_count"`

    // Persistence settings
    WALFilePath            string        `yaml:"wal_file_path"`
    SyncEveryNMessages     int           `yaml:"sync_every_n_messages"`
    RetentionHours         int           `yaml:"retention_hours"`

    // Backpressure thresholds
    LagWarningThreshold    int           `yaml:"lag_warning_threshold"`
    LagThrottleThreshold   int           `yaml:"lag_throttle_threshold"`

    // Operational settings
    LogLevel               string        `yaml:"log_level"`
    MetricsPort            int           `yaml:"metrics_port"`
    AdminPort              int           `yaml:"admin_port"`
}
```

Milestone Verification Checkpoints ensuring requirements satisfaction:

Milestone 1 Checkpoint — Basic Pub/Sub:

```bash
# Start broker
go run cmd/broker/main.go

# Test basic publishing (different terminal)
echo "PUBLISH orders.created hello-world" | nc localhost 9092

# Test subscription and delivery
echo "SUBSCRIBE orders.created" | nc localhost 9092
# Should receive: "MESSAGE orders.created hello-world"
```

Milestone 2 Checkpoint — Consumer Groups:

```bash
# Test consumer group load balancing

# Terminal 1: Join group as consumer-1

echo -e "JOIN_GROUP order-processors consumer-1\nSUBSCRIBE orders.created" | nc localhost 9092


# Terminal 2: Join group as consumer-2

echo -e "JOIN_GROUP order-processors consumer-2\nSUBSCRIBE orders.created" | nc localhost 9092


# Terminal 3: Publish multiple messages

for i in {1..10}; do echo "PUBLISH orders.created message-$i" | nc localhost 9092; done


# Verify: Each message appears in only ONE consumer terminal (round-robin distribution)
```

Milestone 3 Checkpoint — Persistence:

```bash
# Publish message requiring acknowledgment

echo "PUBLISH orders.created test-persistence" | nc localhost 9092


# Kill broker process (Ctrl+C)

# Restart broker

go run cmd/broker/main.go


# Verify: Unacknowledged message redelivered after restart
```

Milestone 4 Checkpoint — Monitoring:

```bash
# Check monitoring endpoint

curl http://localhost:8080/metrics

# Should show: queue_depth, consumer_lag, message_throughput per topic


# Check admin API

curl http://localhost:8080/admin/topics

# Should return: JSON list of topics with subscriber counts


curl http://localhost:8080/admin/groups

# Should return: JSON list of consumer groups with member counts
```

Language-Specific Implementation Hints for Go:

Requirement Area	Go-Specific Approach	Key Packages
TCP Server	`net.Listen()` with goroutine per connection	`net`, `bufio` for protocol parsing
Concurrency	`sync.RWMutex` for data structure protection	`sync`, channels for coordination
Persistence	`os.OpenFile()` with `O_APPEND`, `file.Sync()` for durability	`os`, `encoding/json` for records
Configuration	`yaml.Unmarshal()` with struct tags	`gopkg.in/yaml.v2`
Monitoring	`http.HandleFunc()` for metrics endpoint	`net/http`, `encoding/json`
Logging	`log/slog` for structured logging	`log/slog` (Go 1.21+) or `logrus`

Common Requirements Pitfalls:

⚠️ **Pitfall: Vague Acceptance Criteria** Many implementations fail because "support consumer groups" isn't specific enough. Requirements like FR-4 specify "round-robin distribution" and "only one member receives each message" — these can be tested and verified objectively.

⚠️ **Pitfall: Ignoring Non-Functional Requirements**
Focusing only on functional behavior while ignoring performance and reliability requirements leads to systems that "work" in simple tests but fail under realistic load. The throughput targets (10,000 messages/second) and latency targets (<10ms p99) provide concrete optimization goals.

⚠️ **Pitfall: Scope Creep from Non-Goals** The most dangerous pitfall is implementing excluded features because they "seem important." Adding authentication or clustering might feel like improvements, but they distract from the core learning objectives around message delivery patterns.

High-Level Architecture

> **Milestone(s):** All milestones (1-4) — the architectural foundation that guides component design and implementation throughout the project

Mental Model: The Digital Post Office

Think of our message queue broker as a sophisticated post office that operates at digital speed. Just as a post office has specialized departments working together to handle mail delivery, our broker consists of five core components that collaborate to ensure reliable message delivery.

The **Protocol Handler** acts like the front desk clerks who interact with customers — they understand the language customers speak (our binary protocol) and translate requests into internal operations. The **Topic Manager** functions like the sorting facility that organizes mail by destination addresses — it maintains topic-based message queues and ensures every subscriber receives their copy. The **Consumer Group Coordinator** operates like the distribution center that divides bulk deliveries among multiple delivery trucks — it assigns messages to consumer group members to balance the workload.

The **Acknowledgment Tracker** serves as the certified mail department, keeping detailed records of which messages were delivered and following up on undelivered items. Finally, the **Persistence Layer** acts like the post office's permanent records vault, ensuring that important transactions survive even if the building loses power.

Component Overview

Our message queue broker is architected around five core components, each with distinct responsibilities that together provide the complete messaging functionality outlined in our milestones.

Component	Primary Responsibility	Key Data Managed	Milestone Focus
Protocol Handler	TCP connection management and binary protocol parsing	Active connections, command parsing state	Milestone 1
Topic Manager	Topic-based message routing and subscriber fanout	Topics, subscribers, message queues	Milestone 1
Consumer Group Coordinator	Consumer group membership and message assignment	Consumer groups, member assignments, rebalancing state	Milestone 2
Acknowledgment Tracker	Message delivery tracking and redelivery management	Pending messages, timeouts, retry counts	Milestone 2
Persistence Layer	Message durability and crash recovery	Write-ahead log, checkpoints, retention metadata	Milestone 3

The **Protocol Handler** serves as the system's communication gateway, accepting TCP connections from clients and translating between our binary wire protocol and internal component operations. This component owns the complexity of network I/O, partial reads, connection lifecycle management, and protocol frame parsing. It maintains connection state for heartbeats and graceful shutdown while delegating actual message processing to specialized components.

The **Topic Manager** implements the core message routing logic that enables both publish/subscribe and point-to-point messaging patterns. It maintains in-memory topic structures with subscriber lists and message queues, orchestrates fan-out delivery to multiple subscribers, and supports wildcard topic subscriptions. This component encapsulates all topic-level concerns including topic lifecycle, subscription management, and message ordering guarantees.

The **Consumer Group Coordinator** adds sophisticated load balancing capabilities on top of basic topic routing. It manages consumer group membership, implements round-robin message assignment across group members, and handles the complex rebalancing process when consumers join or leave groups. This component ensures that work is distributed fairly while maintaining delivery guarantees during membership changes.

The **Acknowledgment Tracker** provides reliability guarantees by implementing message delivery confirmation and automatic redelivery. It tracks pending messages for each consumer, monitors acknowledgment timeouts, implements exponential backoff for retries, and ultimately routes poison messages to dead letter queues. This component bridges the gap between at-most-once and at-least-once delivery semantics.

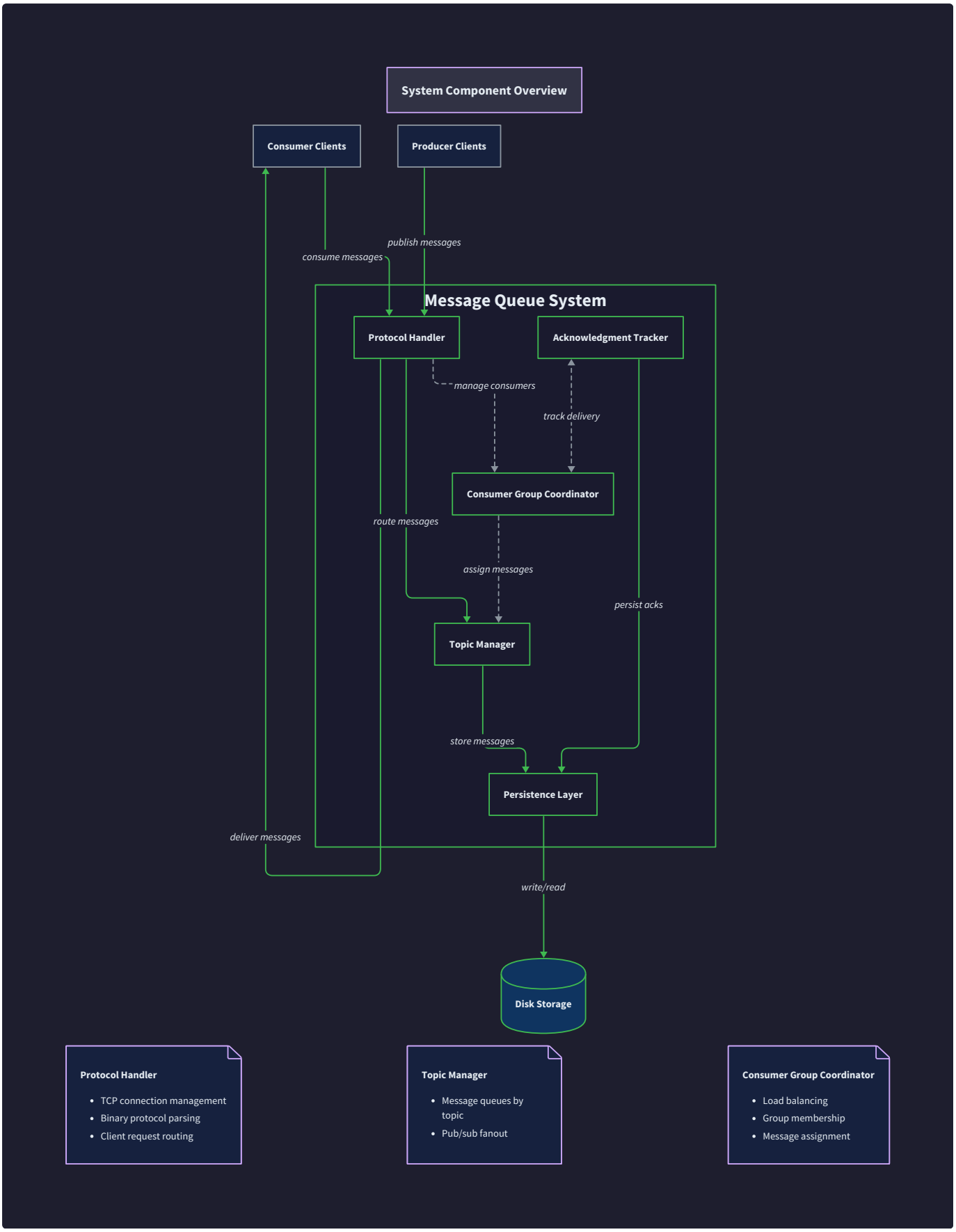
The **Persistence Layer** ensures message durability and system recoverability through an append-only write-ahead log. It handles message persistence before delivery confirmation, implements crash recovery by replaying uncommitted operations, manages log retention policies, and provides the foundation for exactly-once delivery semantics in conjunction with the acknowledgment tracker.

> **Key Architectural Insight**: Each component owns a specific concern and communicates through well-defined interfaces rather than shared state. This separation allows components to evolve independently and makes the system easier to test and debug. The Protocol Handler acts as the orchestrator, but actual message processing flows through specialized components based on the operation type.

The components interact through a clear data flow pattern. Incoming client requests flow through the Protocol Handler to the appropriate component based on operation type — `PUBLISH` commands route to the Topic Manager, `JOIN_GROUP` commands route to the Consumer Group Coordinator, and `ACK` / `NACK` commands route to the Acknowledgment Tracker. The Persistence Layer acts as a cross-cutting concern that each component can utilize for durability guarantees.

Decision: Component Granularity

- **Context**: We need to balance component complexity with interface overhead and decide how finely to decompose the system
- **Options Considered**:
  1. Monolithic broker with internal functions
  2. Five focused components with clear responsibilities
  3. Ten+ micro-components with very narrow concerns
- **Decision**: Five focused components (Protocol Handler, Topic Manager, Consumer Group Coordinator, Acknowledgment Tracker, Persistence Layer)
- **Rationale**: This granularity aligns with our milestone progression, creates testable units without excessive interface overhead, and allows each component to own a complete domain of responsibility. Five components are manageable for intermediate developers while providing clear separation of concerns.
- **Consequences**: Each component can be developed and tested independently, but we need well-defined interfaces and careful data flow design to prevent tight coupling.



Recommended Module Structure

A clean module structure is essential for managing complexity as the system grows across milestones. Our organization follows Go's standard project layout while grouping related functionality and maintaining clear dependency boundaries.

```
message-queue/
├── cmd/
│   ├── broker/                    ← Main broker executable
│   │   └── main.go                ← Server startup and configuration
│   └── client/                    ← Example client for testing
│       └── main.go                ← Client CLI tool
├── internal/                      ← Private broker implementation
│   ├── protocol/                  ← Protocol Handler component (Milestone 1)
│   │   ├── handler.go             ← TCP server and connection management
│   │   ├── parser.go              ← Binary protocol parsing
│   │   ├── commands.go            ← Command type definitions
│   │   └── protocol_test.go       ← Protocol parsing tests
│   ├── topic/                     ← Topic Manager component (Milestone 1)
│   │   ├── manager.go             ← Topic creation and subscription management
│   │   ├── fanout.go              ← Message fanout algorithm
│   │   ├── wildcard.go            ← Wildcard subscription matching
│   │   └── topic_test.go          ← Topic routing tests
│   ├── consumer/                  ← Consumer Group Coordinator (Milestone 2)
│   │   ├── coordinator.go         ← Consumer group management
│   │   ├── rebalancing.go         ← Group rebalancing algorithm
│   │   ├── assignment.go          ← Message assignment strategies
│   │   └── consumer_test.go       ← Consumer group tests
│   ├── ack/                       ← Acknowledgment Tracker (Milestone 2)
│   │   ├── tracker.go             ← Message tracking and timeout management
│   │   ├── redelivery.go          ← Redelivery and poison message handling
│   │   └── ack_test.go            ← Acknowledgment tests
│   ├── persistence/               ← Persistence Layer (Milestone 3)
│   │   ├── wal.go                 ← Write-ahead log implementation
│   │   ├── recovery.go            ← Crash recovery logic
│   │   ├── retention.go           ← Log retention and cleanup
│   │   └── persistence_test.go    ← Persistence tests
│   ├── monitoring/                ← Monitoring and observability (Milestone 4)
│   │   ├── metrics.go             ← Metrics collection
│   │   ├── health.go              ← Health checks and heartbeat
│   │   └── api.go                 ← Administrative REST API
│   └── types/                     ← Shared type definitions
│       ├── message.go             ← Message, Topic, Consumer types
│       ├── config.go              ← BrokerConfig and component configs
│       └── errors.go              ← Custom error types
├── pkg/                           ← Public client libraries
│   └── client/                    ← Go client SDK
│       ├── client.go              ← High-level client interface
│       └── protocol.go            ← Protocol implementation
├── configs/                       ← Configuration files
│   ├── broker.yaml                ← Broker configuration
│   └── logging.yaml               ← Logging configuration
├── docs/                          ← Documentation
│   ├── wire-protocol.md           ← Binary protocol specification
│   └── api.md                     ← REST API documentation
└── scripts/                       ← Build and deployment scripts
    ├── build.sh                   ← Build script
    └── run-tests.sh               ← Test execution script
```

The `internal/` directory contains all broker implementation code that should not be imported by external applications. Each component lives in its own package with focused responsibilities — this structure supports our milestone-based development where each component can be built and tested incrementally.

Directory	Purpose	Dependencies	Testing Strategy
`internal/protocol/`	TCP and protocol handling	`net`, `bufio`, internal types	Mock connections, protocol parsing
`internal/topic/`	Topic and subscription management	internal types only	In-memory topic operations
`internal/consumer/`	Consumer group coordination	internal types, topic manager	Mock consumer connections
`internal/ack/`	Message acknowledgment tracking	internal types, persistence	Timeout simulation, redelivery
`internal/persistence/`	Write-ahead log and recovery	`os`, `encoding/json`	File I/O mocking, corruption tests
`internal/monitoring/`	Metrics and health checks	`net/http`, `encoding/json`	HTTP endpoint testing

The `pkg/` directory exposes client libraries that external applications can import. This separation ensures that internal broker implementation details remain encapsulated while providing clean APIs for client applications.

Decision: Package Organization Strategy

- **Context**: Go projects can organize code by feature, by layer, or by component — each approach has trade-offs for maintainability and testing
- **Options Considered**:
  1. Feature-based packages (pubsub/, queuing/, persistence/)
  2. Layer-based packages (transport/, domain/, storage/)
  3. Component-based packages aligned with architectural components
- **Decision**: Component-based packages where each package corresponds to one architectural component
- **Rationale**: This organization matches our milestone structure, allows incremental development, creates clear ownership boundaries, and makes testing easier since each component has focused responsibilities. Dependencies flow in one direction from protocol handler to specialized components.
- **Consequences**: Each component can be developed independently, but we need careful interface design to prevent circular dependencies. The structure clearly maps to our architectural diagram and milestone progression.

Deployment Model

Our message queue broker is designed as a **single-node, in-memory system** that prioritizes simplicity and learning over distributed systems complexity. This deployment model aligns with the intermediate difficulty level and allows focus on core messaging concepts without the operational overhead of clustering.

Deployment Aspect	Current Design	Future Considerations
Node Architecture	Single broker process	Leader-follower with automatic failover
State Storage	In-memory with WAL backup	Distributed consensus (Raft/PBFT)
Client Connections	Direct TCP to broker	Load balancer with broker discovery
Failover Strategy	Process restart with WAL recovery	Hot standby with state replication
Horizontal Scaling	Vertical scaling only	Topic partitioning across nodes
Network Topology	Star topology (clients to broker)	Mesh or hub-and-spoke cluster

The **single-node model** simplifies several architectural concerns that would otherwise dominate the design. There is no need for distributed consensus protocols, cross-node message replication, or cluster membership management. Consumer group rebalancing only occurs when consumers join or leave groups, not when broker nodes fail. Message ordering is naturally guaranteed within each topic since all messages flow through a single broker instance.

Resource Management in a single-node deployment focuses on efficient memory usage and preventing resource exhaustion. The broker maintains in-memory topic queues, subscriber lists, and consumer group assignments. The Persistence Layer provides durability through write-

ahead logging but does not replicate data to other nodes. Memory consumption grows with the number of topics, active subscriptions, and unacknowledged messages — the system implements backpressure and retention policies to prevent unbounded growth.

High Availability is achieved through process-level recovery rather than distributed failover. When the broker process terminates unexpectedly, it can be restarted and will recover its state by replaying the write-ahead log. Clients experience a brief interruption during recovery but can reconnect once the broker completes its startup sequence. This approach provides reasonable availability for development and testing environments while keeping operational complexity minimal.

> **Key Design Insight**: The single-node constraint is a feature, not a limitation, for this learning project. It allows us to focus on message queue fundamentals — delivery guarantees, consumer groups, acknowledgment protocols, and persistence — without being distracted by distributed systems challenges like split-brain scenarios, network partitions, or consensus algorithms.

Operational Characteristics of the single-node deployment include predictable performance characteristics since there is no network communication between broker nodes. Latency is determined primarily by TCP round-trip time, message serialization overhead, and disk I/O for persistence. Throughput scales with the broker's CPU and memory resources, making performance tuning straightforward.

The broker process runs as a single executable that accepts configuration through YAML files or environment variables. Standard deployment patterns include running as a systemd service on Linux, as a Docker container, or directly as a development server. The simple deployment model makes it suitable for development environments, testing infrastructure, and learning scenarios.

Deployment Environment	Recommended Configuration	Use Case
Development	Direct process execution with file-based WAL	Local development and debugging
Testing/CI	Docker container with ephemeral storage	Integration tests and CI pipelines
Production-like	Systemd service with persistent storage	Demo environments and learning

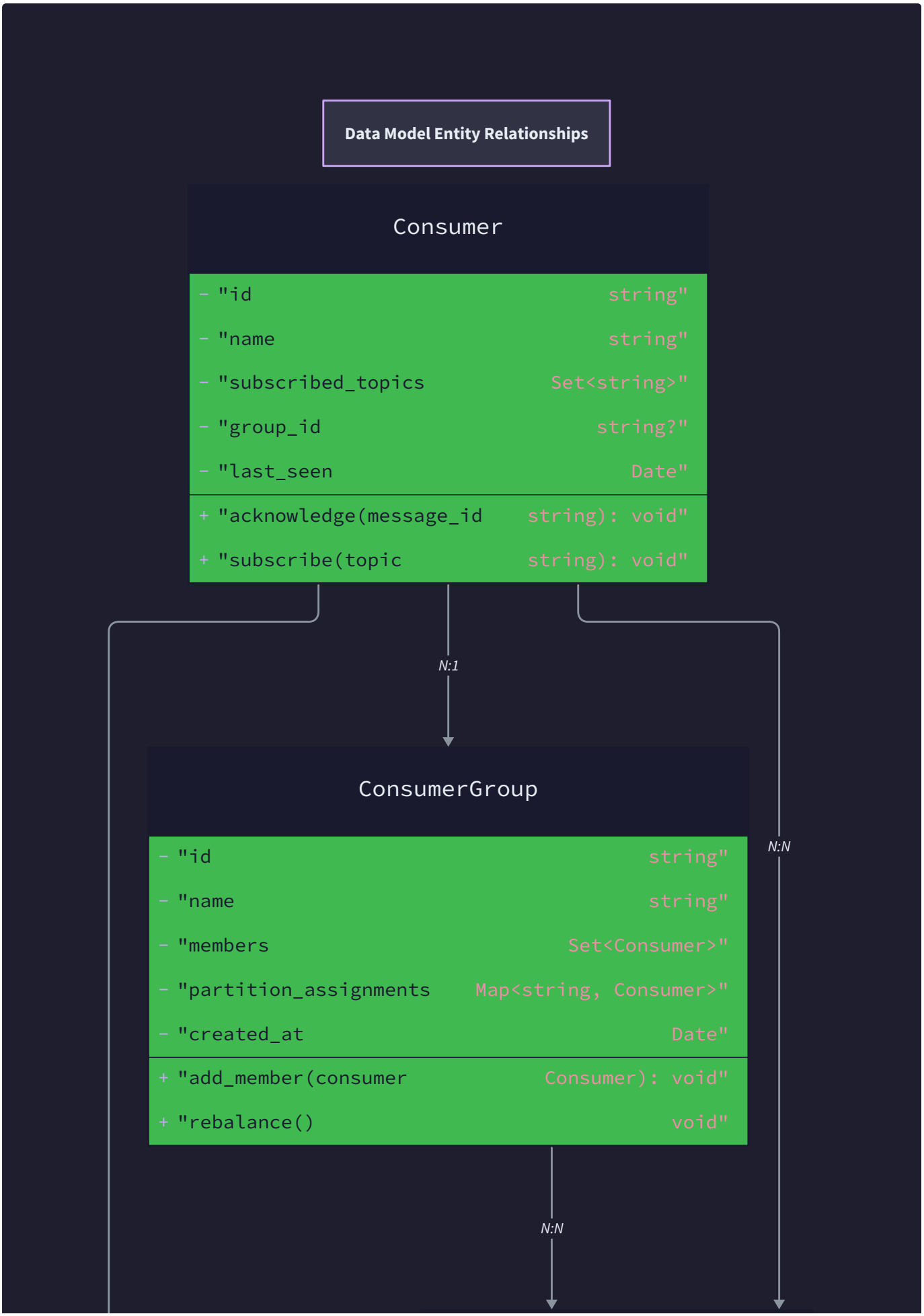
Future Clustering Considerations are built into the architectural design even though they are not implemented. Each component interface is designed to be stateless or to maintain state that could be distributed across nodes. The Topic Manager could partition topics across multiple nodes, the Consumer Group Coordinator could distribute groups across a cluster, and the Persistence Layer could replicate write-ahead logs to follower nodes.

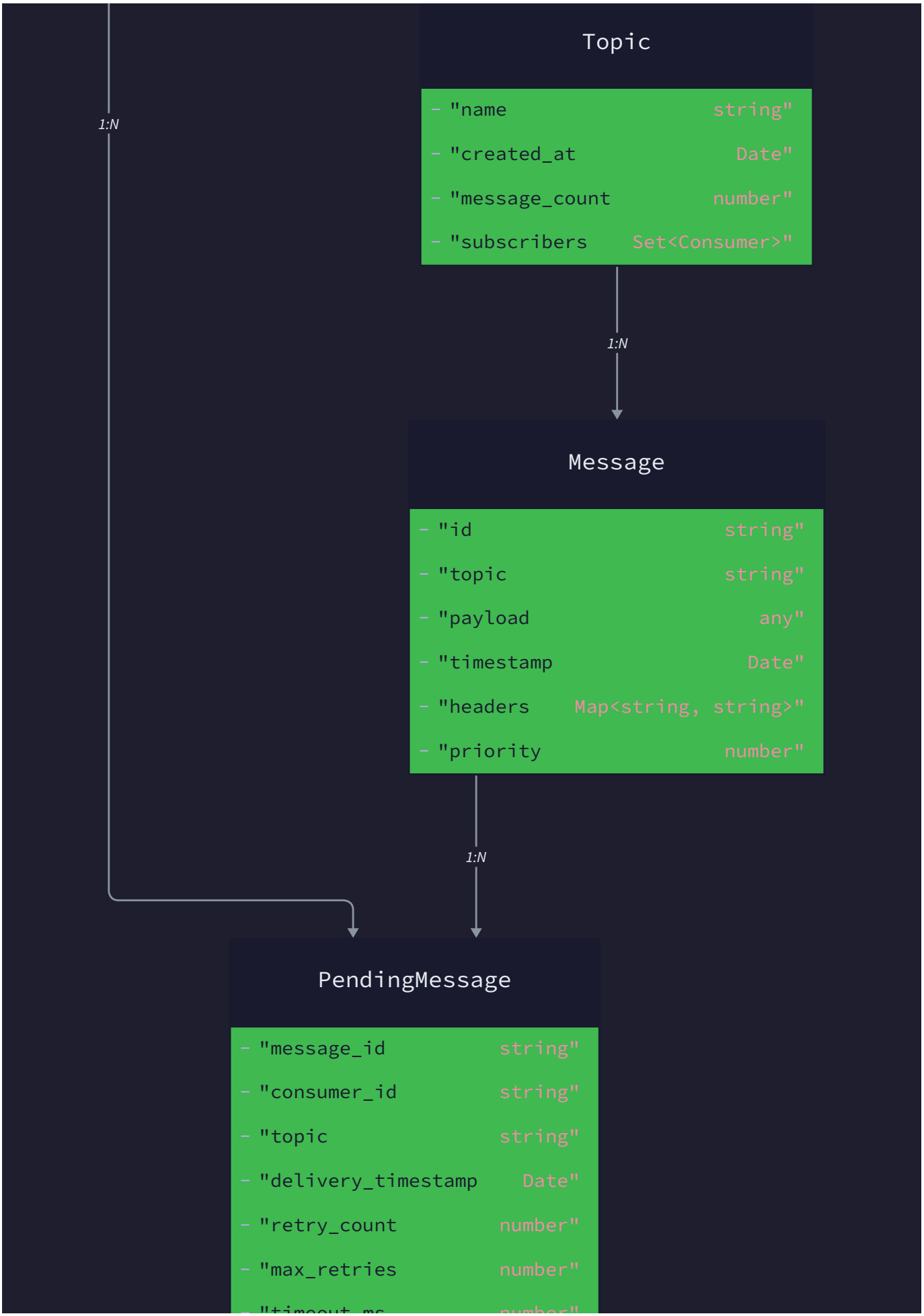
The Protocol Handler's connection-oriented design would extend naturally to a clustered environment where clients connect to any broker node and requests are internally routed to the appropriate node based on topic partitioning. The binary protocol includes fields that could support node addressing and request routing in a distributed deployment.

> **Decision: Single-Node vs Distributed Architecture**
>
> - **Context**: Message queues can be architected as single-node systems (like Redis) or distributed systems (like Kafka) — each approach has different complexity and learning trade-offs
> - **Options Considered**:
>   1. Single-node in-memory broker with WAL persistence
>   2. Multi-node cluster with distributed consensus
>   3. Hybrid approach with primary/secondary replication
> - **Decision**: Single-node in-memory broker with write-ahead log persistence
> - **Rationale**: This approach focuses learning on core message queue concepts rather than distributed systems complexity. Students can understand delivery guarantees, consumer groups, and persistence without getting lost in consensus protocols or cluster management. The architecture can be extended to clustering later.
> - **Consequences**: The system is simpler to develop, test, and operate, but cannot provide high availability through node redundancy. Recovery depends on process restart rather than automatic failover.

Data Model

Milestone(s): All milestones (1-4) — the core data structures that evolve throughout the project, from basic message handling in Milestone 1 to persistence and dead letter queues in Milestones 3-4


Data Model Entity Relationships
Consumer
- "id string"
- "name string"
- "subscribed_topics Set<string>"
- "group_id string?"
- "last_seen Date"
+ "acknowledge(message_id string): void"
+ "subscribe(topic string): void"
N:1
N:N
ConsumerGroup
- "id string"
- "name string"
- "members Set<Consumer>"
- "partition_assignments Map<string, Consumer>"
- "created_at Date"
+ "add_member(consumer Consumer): void"
+ "rebalance() void"
N:N


Topic
- "name string"
- "created_at Date"
- "message_count number"
- "subscribers Set<Consumer>"
1:N
1:N
Message
- "id string"
- "topic string"
- "payload any"
- "timestamp Date"
- "headers Map<string, string>"
- "priority number"
1:N
PendingMessage
- "message_id string"
- "consumer_id string"
- "topic string"
- "delivery_timestamp Date"
- "retry_count number"
- "max_retries number"

Mental Model: The Library Card Catalog System

Before diving into the technical data structures, let's build intuition using a familiar analogy. Think of our message queue's data model as a sophisticated library card catalog system from the pre-digital era.

In this mental model, **messages** are like books that patrons want to borrow. Each book has a unique identification number, contains specific content, and carries metadata about who requested it, when it arrived, and where it should be delivered. The **topics** are like different library sections (Fiction, Science, History) where books are organized by subject matter. When someone wants to read science books, they don't need to know about every romance novel that arrives.

Consumers are like library patrons who have signed up to receive notifications when new books arrive in their areas of interest. Some patrons prefer to receive every book individually (like subscribers in pub/sub), while others form **consumer groups** — study groups that share the workload, where only one member needs to read each book and summarize it for the others.

The library maintains detailed **records** of every transaction: which books were requested, who received them, whether they were returned properly, and what happened to books that couldn't be delivered. Books that repeatedly cause problems (damaged, inappropriate content, or repeatedly rejected) end up in a **special collection** (our dead letter queue) for manual review.

Just as a library needs robust record-keeping to track thousands of books and hundreds of patrons, our message queue requires carefully designed data structures to handle thousands of messages flowing between dozens of producers and consumers, all while maintaining order, preventing loss, and enabling recovery from failures.

Core Types

The foundation of our message queue rests on four primary data structures that capture the essential entities in any messaging system. These types evolve throughout the project milestones, starting simple in Milestone 1 and gaining complexity as we add consumer groups, persistence, and observability features.

Message Structure

The `Message` represents the fundamental unit of communication in our system. Unlike simple string payloads, our messages carry rich metadata that enables advanced features like routing, acknowledgment tracking, and forensic analysis.

Field	Type	Description
`ID`	string	Unique identifier generated by the broker, used for deduplication and acknowledgment tracking
`Topic`	string	Destination topic name where this message should be delivered
`Payload`	[]byte	Raw message content as provided by the producer, stored as bytes for language neutrality
`Headers`	map[string]string	Key-value metadata for routing, tracing, and application-specific context
`Timestamp`	int64	Unix timestamp when the message was accepted by the broker, used for ordering and TTL
`ProducerID`	string	Identifier of the client that published this message, used for quotas and debugging
`RetryCount`	int	Number of delivery attempts, incremented on each redelivery for poison message detection
`MaxRetries`	int	Maximum retry attempts before sending to dead letter queue, configurable per message
`AckDeadline`	int64	Unix timestamp when acknowledgment is required, after which message becomes eligible for redelivery
`AssignedConsumer`	string	Consumer ID currently responsible for processing this message, empty if unassigned

Topic Structure

The `Topic` acts as a named channel that organizes messages by subject and maintains the list of interested subscribers. Topics are created automatically when first referenced, but persist until explicitly deleted or garbage collected.

Field	Type	Description
`Name`	string	Hierarchical topic identifier supporting wildcards, e.g., "orders.created", "events.user.*"
`Subscribers`	[]Subscriber	List of direct subscribers that receive fan-out copies of every message
`ConsumerGroups`	map[string]*ConsumerGroup	Named consumer groups registered for this topic, indexed by group name
`MessageQueue`	[]Message	In-memory FIFO queue holding pending messages for this topic
`MessageIndex`	map[string]int	Maps message IDs to queue positions for fast lookup during acknowledgment
`RetentionPolicy`	RetentionPolicy	Rules governing when messages can be deleted (time-based, size-based, or acknowledge-based)
`CreatedAt`	int64	Unix timestamp when this topic was first created, used for lifecycle management
`LastActivity`	int64	Unix timestamp of most recent publish or subscribe activity, used for garbage collection
`TotalMessages`	int64	Cumulative count of messages published to this topic since creation, used for monitoring
`ActiveMessages`	int64	Current count of unacknowledged messages, used for backpressure calculations

Consumer Structure

The `Consumer` represents an individual client connection that processes messages from one or more topics. Consumers can operate independently (direct subscribers) or as part of a consumer group for load balancing.

Field	Type	Description
`ID`	string	Unique identifier for this consumer connection, typically connection-based
`ClientAddress`	string	Network address of the consumer client, used for debugging and monitoring
`SubscribedTopics`	[]string	List of topic patterns this consumer is interested in receiving
`GroupID`	string	Consumer group name if this consumer is part of a group, empty for direct subscribers
`LastHeartbeat`	int64	Unix timestamp of most recent heartbeat or message acknowledgment from this consumer
`MaxInflight`	int	Maximum number of unacknowledged messages this consumer can handle concurrently
`CurrentInflight`	int	Current count of messages assigned to this consumer but not yet acknowledged
`PendingMessages`	map[string]Message	Messages currently assigned to this consumer, indexed by message ID
`ProcessedCount`	int64	Total number of messages successfully processed by this consumer since connection
`ErrorCount`	int64	Total number of processing errors (NACKs) from this consumer, used for health assessment

ConsumerGroup Structure

The `ConsumerGroup` coordinates multiple consumers to distribute workload fairly while maintaining message ordering guarantees. Groups automatically rebalance when membership changes.

Field	Type	Description
`Name`	string	Unique identifier for this consumer group within the broker
`Topic`	string	Topic that this consumer group is subscribed to, one group per topic
`Members`	map[string]*Consumer	Active consumers in this group, indexed by consumer ID
`AssignmentStrategy`	string	Algorithm for distributing messages among members: "round-robin" or "sticky"
`RebalanceInProgress`	bool	Flag indicating that group membership is changing and assignments are being redistributed
`RebalanceGeneration`	int64	Incrementing counter for rebalance operations, used to detect stale assignments
`MessageAssignments`	map[string]string	Maps message IDs to consumer IDs showing current work distribution
`NextAssignment`	int	Round-robin counter for the next consumer to receive a message
`CreatedAt`	int64	Unix timestamp when this group was first created
`LastRebalance`	int64	Unix timestamp of most recent rebalancing operation
`TotalRebalances`	int64	Cumulative count of rebalancing operations since group creation

Decision: Rich Message Metadata

- **Context**: Messages could be simple payloads or carry extensive metadata for advanced features
- **Options Considered**:
  1. Simple string/byte payload only
  2. Message with basic routing metadata (topic, timestamp)
  3. Rich message structure with full lifecycle tracking
- **Decision**: Rich message structure with comprehensive metadata
- **Rationale**: Enables acknowledgment tracking, poison message detection, consumer assignment, and operational observability without requiring protocol changes later
- **Consequences**: Higher memory overhead per message, but enables all advanced features in subsequent milestones

RetentionPolicy Structure

The `RetentionPolicy` defines when messages can be safely deleted from topics, balancing memory usage against message availability for late-joining consumers.

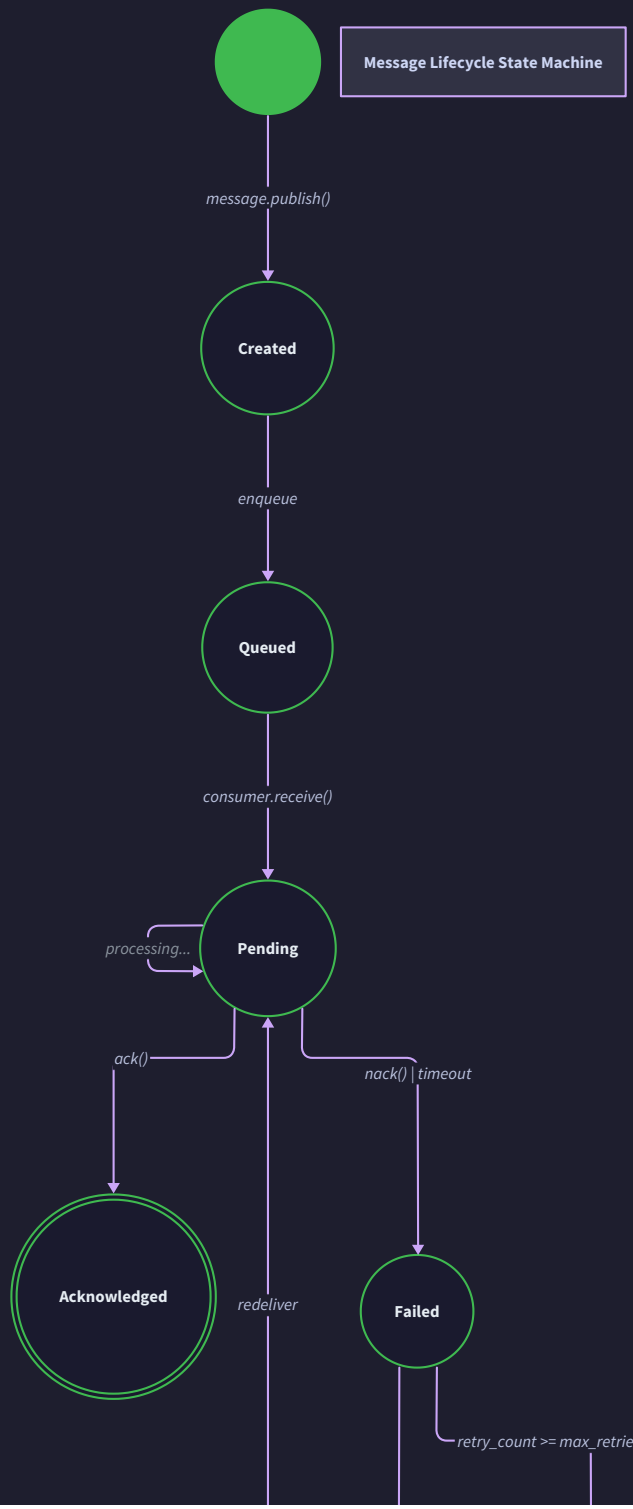
Field	Type	Description
`MaxAge`	int64	Maximum message age in seconds before eligible for deletion, 0 means no time limit
`MaxSize`	int64	Maximum total payload bytes for topic before oldest messages are evicted, 0 means no size limit
`MaxMessages`	int64	Maximum number of messages per topic before oldest are evicted, 0 means no count limit
`RequireAck`	bool	Whether messages must be acknowledged by all subscribers before deletion
`CompactionEnabled`	bool	Whether to compact repeated keys by keeping only the latest value

Message Lifecycle States

Understanding how messages transition through different states is crucial for implementing reliable delivery guarantees and debugging message flow issues. Our state machine captures the complete lifecycle from producer submission to final disposition.

Message States

- **Created**: Message instantiated by producer
- **Queued**: Message stored in topic/queue
- **Pending**: Message delivered, awaiting acknowledgment
- **Acknowledged**: Message successfully processed
- **Failed**: Processing failed or timed out
- **Redelivery**: Message queued for retry
- **Dead Letter**: Maximum retries exceeded



The message lifecycle represents a carefully orchestrated journey that ensures reliability while maintaining performance. Each state transition is triggered by specific events and may involve multiple components coordinating to maintain system consistency.

State Transition Table

Current State	Event	Next State	Actions Taken	Component Responsible
Created	Message accepted by broker	Pending	Assign message ID, record timestamp, append to WAL	Protocol Handler
Pending	Consumer available in group	Assigned	Set AssignedConsumer, update AckDeadline, send to consumer	Consumer Group Coordinator
Pending	Direct subscriber connected	Delivered	Add to subscriber's pending set, send message copy	Topic Manager
Assigned	ACK received from consumer	Acknowledged	Remove from pending sets, update consumer statistics	Acknowledgment Tracker
Assigned	NACK received from consumer	Pending	Clear AssignedConsumer, increment RetryCount	Acknowledgment Tracker
Assigned	Acknowledgment deadline exceeded	Pending	Clear AssignedConsumer, increment RetryCount, log timeout	Acknowledgment Tracker
Pending	RetryCount exceeds MaxRetries	Dead Letter	Move to DLQ topic, record original destination	DLQ Manager
Acknowledged	Retention policy satisfied	Deleted	Remove from all queues, update topic statistics	Topic Manager
Dead Letter	Admin replay command	Pending	Reset RetryCount, restore to original topic	DLQ Manager

Detailed State Descriptions

The **Created** state represents the brief moment when a message has been received by the broker but not yet persisted or made available for delivery. This state exists primarily for protocol correctness — the producer receives confirmation only after the message transitions to Pending state, ensuring durability guarantees are met.

During the **Pending** state, messages wait in topic queues for available consumers or direct delivery to subscribers. This is where message ordering is preserved — messages are delivered in the order they entered the Pending state for each topic. The Topic Manager maintains FIFO semantics during this phase, ensuring that earlier messages are not indefinitely delayed by consumer availability issues.

The **Assigned** state indicates that a specific consumer in a consumer group has been chosen to process the message. The message remains in this state until the consumer sends an explicit ACK or NACK, or until the acknowledgment deadline expires. During assignment, other consumers in the group will not receive this message, implementing the exactly-once delivery semantics within the group.

Acknowledged messages have been successfully processed by their assigned consumer or delivered to all direct subscribers. These messages remain in the system only until retention policies allow their deletion. The acknowledgment event triggers metric updates and may release backpressure if consumer lag was causing flow control.

Messages reach the **Dead Letter** state when they exceed their maximum retry count, indicating a persistent processing problem. Dead letter messages are moved to a special DLQ topic where they can be inspected, analyzed, and potentially replayed after fixing the underlying issue. This prevents poison messages from indefinitely consuming system resources.

The **Deleted** state is terminal — messages are permanently removed from all queues and indices. This transition is triggered by retention policies based on time, size, or acknowledgment requirements. Proper cleanup is essential to prevent memory leaks in long-running brokers.

> **Decision: Explicit State Tracking**
>
> - **Context**: Message states could be implicit (derived from data) or explicit (stored fields)
> - **Options Considered**:
>   1. Implicit states based on presence in different queues
>   2. Explicit state field with validation
>   3. Event-sourced state transitions with full history
> - **Decision**: Implicit states with explicit validation checkpoints
> - **Rationale**: Reduces memory overhead while maintaining correctness through validation logic; easier to debug than pure event sourcing
> - **Consequences**: Requires careful queue management to maintain state consistency, but enables efficient batch operations

State Transition Events

Understanding what triggers each state transition is crucial for implementing correct message handling logic. Events originate from various sources — network protocols, timer expirations, administrative commands, or internal policy evaluations.

Producer Events include message publication, which creates new messages in the Created state, and producer disconnections, which may trigger cleanup of unacknowledged messages if the producer was also consuming.

Consumer Events drive most state transitions: consumer connections trigger message delivery attempts, ACK/NACK responses transition messages between Assigned and Pending states, and consumer disconnections may cause rebalancing and message reassignment.

Timer Events provide system reliability: acknowledgment deadline expiration moves messages from Assigned back to Pending state, retention policy evaluation transitions Acknowledged messages to Deleted state, and heartbeat timeouts can trigger consumer removal and rebalancing.

Administrative Events enable operational control: manual message replay moves messages from Dead Letter back to Pending state, topic deletion triggers mass transition to Deleted state, and configuration changes may alter retention policies or retry limits.

Persistence Format

The persistence layer serves as the system's memory, ensuring that messages, consumer state, and broker configuration survive crashes and restarts. Our append-only log design prioritizes write performance and crash consistency over read optimization, following the principle that writes are more frequent and latency-sensitive than reads.

Write-Ahead Log (WAL) Structure

The WAL captures every state-changing operation before it's applied to in-memory structures. This ensures that we can rebuild complete system state after any failure, even if the crash occurred in the middle of a complex operation like consumer group rebalancing.

Field	Type	Size	Description
Magic	uint32	4 bytes	File format identifier (0x4D514C47) for corruption detection
Version	uint32	4 bytes	WAL format version for backward compatibility during upgrades
RecordLength	uint32	4 bytes	Total size of this record including headers, used for framing
RecordType	uint8	1 byte	Operation type: MESSAGE_PUBLISHED, ACK_RECEIVED, CONSUMER_JOINED, etc.
Timestamp	uint64	8 bytes	Unix nanoseconds when operation was initiated, for ordering and debugging
TransactionID	uint64	8 bytes	Unique identifier linking related operations, 0 for single-operation transactions
Checksum	uint32	4 bytes	CRC32 of record data for corruption detection on replay
RecordData	[]byte	Variable	JSON-encoded operation payload containing all necessary state changes

Record Type Enumeration

The RecordType field enables efficient filtering during recovery and allows different record types to have specialized handling logic.

Record Type	Value	Payload Schema	Description
MESSAGE_PUBLISHED	0x01	MessageRecord	New message accepted by broker and assigned to topic
MESSAGE_ACKNOWLEDGED	0x02	AckRecord	Consumer confirmed successful processing of assigned message
MESSAGE_NACKED	0x03	NackRecord	Consumer reported processing failure, message should be retried
MESSAGE_EXPIRED	0x04	ExpiryRecord	Message exceeded retry limit or TTL, moved to dead letter queue
CONSUMER_JOINED	0x05	ConsumerRecord	New consumer connected and joined topic or consumer group
CONSUMER_LEFT	0x06	ConsumerRecord	Consumer disconnected or was removed due to heartbeat failure
GROUP_REBALANCED	0x07	RebalanceRecord	Consumer group membership changed and work was redistributed
TOPIC_CREATED	0x08	TopicRecord	New topic was created with initial configuration
TOPIC_DELETED	0x09	TopicRecord	Topic and all its messages were permanently removed
CHECKPOINT	0xFF	CheckpointRecord	Snapshot of broker state for faster recovery

Payload Schema Definitions

Each record type has a specific JSON schema that contains all information needed to replay the operation during crash recovery. These schemas evolve carefully to maintain backward compatibility.

MessageRecord Schema:

```
{
  "messageID": string,
  "topic": string,
  "payload": base64-encoded bytes,
  "headers": object with string values,
  "producerID": string,
  "maxRetries": integer
}
```

AckRecord Schema:

```
{
  "messageID": string,
  "consumerID": string,
  "processingTime": integer milliseconds,
  "success": boolean
}
```

ConsumerRecord Schema:

```
{
  "consumerID": string,
  "clientAddress": string,
  "subscribedTopics": array of strings,
  "groupID": string or null,
  "maxInflight": integer
}
```

RebalanceRecord Schema:

```
{
  "groupName": string,
  "topic": string,
  "newGeneration": integer,
  "memberAssignments": object mapping consumer IDs to message ID arrays,
  "removedMembers": array of consumer IDs
}
```

Indexing Strategy

To enable efficient recovery and operational queries, we maintain lightweight indices that can be rebuilt from the WAL if corrupted or lost.

Message Index Structure: The message index enables fast lookup of message state during acknowledgment processing and administrative queries.

Field	Type	Description
MessageID	string	Unique message identifier as primary key
WALOffset	int64	Byte offset in WAL where the MESSAGE_PUBLISHED record is stored
CurrentState	string	Derived state: "pending", "assigned", "acknowledged", "dead_letter"
AssignedConsumer	string	Consumer ID if message is currently assigned, empty otherwise
TopicName	string	Topic where message was published, for cross-referencing
RetryCount	int	Current number of delivery attempts
LastModified	int64	WAL offset of most recent record affecting this message

Topic Index Structure: The topic index provides fast access to topic metadata and supports wildcard subscription matching.

Field	Type	Description
TopicName	string	Full topic name as specified by producers
CreationOffset	int64	WAL offset where TOPIC_CREATED record is stored
MessageCount	int64	Current number of messages in this topic's queue
SubscriberCount	int	Number of direct subscribers registered
ConsumerGroups	[]string	List of consumer group names registered for this topic
LastActivity	int64	WAL offset of most recent message or subscription activity

Consumer Group Index Structure: The consumer group index enables fast group membership queries and rebalancing decisions.

Field	Type	Description
GroupName	string	Unique consumer group identifier
TopicName	string	Topic that this group is subscribed to
CurrentGeneration	int64	Current rebalancing generation number
ActiveMembers	[]string	List of consumer IDs currently in this group
PendingRebalance	bool	Whether group is currently undergoing membership changes
LastRebalanceOffset	int64	WAL offset of most recent GROUP_REBALANCED record

Decision: Append-Only WAL with JSON Payloads

- **Context**: Persistence format could prioritize compactness, human readability, or write performance
- **Options Considered**:
  1. Binary format with custom serialization for maximum compactness
  2. JSON payloads with binary headers for debuggability
  3. Full-text logging with structured queries
- **Decision**: JSON payloads with binary headers
- **Rationale**: JSON enables easy debugging and administrative tools while binary headers provide efficient framing and metadata; append-only design maximizes write throughput
- **Consequences**: Larger file sizes than binary format but significantly easier to debug and extend

Checkpointing Strategy

To prevent unbounded recovery times, we periodically write checkpoint records that contain complete snapshots of broker state. Checkpoints enable recovery to start from a recent point rather than replaying the entire WAL from the beginning.

Checkpoint Record Structure:

Section	Content	Description
Topics	Full topic metadata and message queues	Complete state of all active topics including pending messages
Consumers	Consumer connection state and subscriptions	All active consumer connections and their configurations
Consumer Groups	Group membership and message assignments	Current consumer group state including pending rebalances
Metrics	Cumulative counters and performance statistics	Operational metrics that should survive broker restarts

Checkpointing Algorithm:

1. **Trigger Evaluation**: Checkpoints are triggered by WAL size thresholds (e.g., 100MB), time intervals (e.g., every 5 minutes), or administrative commands
2. **State Serialization**: Serialize complete broker state to temporary checkpoint file, ensuring atomic writes
3. **WAL Coordination**: Record checkpoint record in WAL containing file path and state hash for validation
4. **File Rotation**: Move temporary checkpoint file to permanent location with atomic rename operation
5. **Cleanup**: Mark older checkpoint files eligible for deletion, maintaining at least two recent checkpoints for redundancy

Recovery Algorithm:

1. **Checkpoint Location**: Scan WAL from end backwards to find most recent valid checkpoint record
2. **State Restoration**: Load checkpoint file and deserialize broker state, validating checksums
3. **Incremental Replay**: Replay WAL records from checkpoint to end of log, applying state changes
4. **Index Rebuilding**: Reconstruct in-memory indices from restored state
5. **Validation**: Verify system consistency and mark recovery complete, begin accepting new connections

Common Pitfalls

⚠️ **Pitfall: Message ID Collisions** A common mistake is using simple incrementing counters for message IDs, which can cause collisions after broker restarts if the counter is reset. This breaks message deduplication and acknowledgment tracking. Instead, use UUIDs or combine timestamp with broker ID to ensure global uniqueness.

⚠️ **Pitfall: Unbounded Message Queues** Failing to implement proper retention policies or backpressure can cause topic queues to grow without bounds, eventually exhausting memory. Always implement size-based limits and actively monitor queue depths, rejecting new messages when thresholds are exceeded.

⚠️ **Pitfall: Race Conditions in State Transitions** Message state changes can race between components — for example, a consumer ACK arriving while a timeout is being processed. Use proper locking or atomic operations around state transitions, and validate state preconditions before applying changes.

⚠️ **Pitfall: WAL Corruption Handling** When WAL records are corrupted (incomplete writes, checksum mismatches), the naive approach is to crash the broker. Instead, implement graceful degradation by stopping replay at the corruption point and continuing with partial state, logging the issue for administrative attention.

⚠️ **Pitfall: Index Inconsistencies** If indices become inconsistent with WAL data due to bugs or partial failures, queries will return incorrect results. Always validate index consistency during startup and provide administrative commands to rebuild indices from WAL when needed.

⚠️ **Pitfall: Memory Leaks from Unbounded Maps** Consumer and message tracking maps can grow unbounded if cleanup logic has bugs. Implement periodic cleanup jobs that remove stale entries based on timeouts, and monitor map sizes in production.

Implementation Guidance

This subsection provides concrete guidance for implementing the data model in Go, including complete infrastructure code and structured skeletons for core logic.

Technology Recommendations Table:

Component	Simple Option	Advanced Option
Serialization	JSON with encoding/json	Protocol Buffers with protobuf
WAL Storage	os.File with manual sync	Embedded database like BadgerDB
ID Generation	UUID with google/uuid	Snowflake ID with custom generator
Indexing	In-memory maps with sync.RWMutex	Embedded search with Bleve
Checksums	CRC32 with hash/crc32	xxHash with faster algorithm

Recommended File/Module Structure:

```
internal/data/
  types.go              ← Core type definitions (Message, Topic, Consumer, etc.)
  lifecycle.go          ← Message state machine implementation
  wal.go               ← Write-ahead log with persistence
  wal_test.go          ← WAL recovery and corruption tests
  checkpoint.go        ← State snapshotting for fast recovery
  retention.go         ← Message cleanup and retention policies
  index.go             ← In-memory indices for fast lookups
  validation.go        ← Data consistency checks and repairs
```

Core Type Definitions (Complete Implementation):

```go
// Package data contains all core data structures and persistence logic
// for the message queue broker.
package data

import (
    "encoding/json"
    "sync"
    "time"
)

// Message represents a single message in the queue with full lifecycle metadata.
// Messages are immutable after creation except for state tracking fields.
type Message struct {
    ID               string            `json:"id"`
    Topic            string            `json:"topic"`
    Payload          []byte            `json:"payload"`
    Headers          map[string]string `json:"headers"`
    Timestamp        int64             `json:"timestamp"`
    ProducerID       string            `json:"producer_id"`
    RetryCount       int               `json:"retry_count"`
    MaxRetries       int               `json:"max_retries"`
    AckDeadline      int64             `json:"ack_deadline"`
    AssignedConsumer string            `json:"assigned_consumer"`
}

// Topic manages message queues and subscriber lists for a named destination.
type Topic struct {
    Name            string                    `json:"name"`
    Subscribers     map[string]*Consumer      `json:"-"` // Not persisted, rebuilt on startup
    ConsumerGroups  map[string]*ConsumerGroup `json:"-"` // Not persisted, rebuilt on startup
    MessageQueue    []Message                 `json:"message_queue"`
    MessageIndex    map[string]int            `json:"-"` // Message ID -> queue position
    RetentionPolicy RetentionPolicy           `json:"retention_policy"`
    CreatedAt       int64                     `json:"created_at"`
    LastActivity    int64                     `json:"last_activity"`
    TotalMessages   int64                     `json:"total_messages"`
    ActiveMessages  int64                     `json:"active_messages"`
```

```
    mu sync.RWMutex `json:"-"` // Protects all fields during concurrent access
}

// Consumer represents an active client connection processing messages.
type Consumer struct {
    ID               string             `json:"id"`
    ClientAddress    string             `json:"client_address"`
    SubscribedTopics []string           `json:"subscribed_topics"`
    GroupID          string             `json:"group_id"`
    LastHeartbeat    int64              `json:"last_heartbeat"`
    MaxInflight      int                `json:"max_inflight"`
    CurrentInflight  int                `json:"current_inflight"`
    PendingMessages  map[string]Message `json:"pending_messages"`
    ProcessedCount   int64              `json:"processed_count"`
    ErrorCount       int64              `json:"error_count"`

    mu sync.Mutex `json:"-"` // Protects inflight counters and pending messages
}

// ConsumerGroup coordinates load balancing across multiple consumers.
type ConsumerGroup struct {
    Name               string               `json:"name"`
    Topic              string               `json:"topic"`
    Members            map[string]*Consumer  `json:"-"` // Not persisted, rebuilt on startup
    AssignmentStrategy string               `json:"assignment_strategy"`
    RebalanceInProgress bool                `json:"rebalance_in_progress"`
    RebalanceGeneration int64               `json:"rebalance_generation"`
    MessageAssignments map[string]string     `json:"message_assignments"` // Message ID -> Consumer ID
    NextAssignment     int                  `json:"next_assignment"`
    CreatedAt          int64                `json:"created_at"`
    LastRebalance      int64                `json:"last_rebalance"`
    TotalRebalances    int64                `json:"total_rebalances"`

    mu sync.RWMutex `json:"-"` // Protects group membership and assignments
}
```

```
// RetentionPolicy defines when messages can be deleted from topics.
type RetentionPolicy struct {
    MaxAge            int64 `json:"max_age_seconds"`
    MaxSize           int64 `json:"max_size_bytes"`
    MaxMessages       int64 `json:"max_messages"`
    RequireAck        bool  `json:"require_ack"`
    CompactionEnabled bool  `json:"compaction_enabled"`
}

// MessageState represents the current lifecycle state of a message.
type MessageState string

const (
    StateCreated     MessageState = "created"
    StatePending     MessageState = "pending"
    StateAssigned    MessageState = "assigned"
    StateAcknowledged MessageState = "acknowledged"
    StateDeadLetter  MessageState = "dead_letter"
    StateDeleted     MessageState = "deleted"
)

// WALRecord represents a single entry in the write-ahead log.
type WALRecord struct {
    Magic         uint32          `json:"-"`        // Binary header field
    Version       uint32          `json:"-"`        // Binary header field
    RecordLength  uint32          `json:"-"`        // Binary header field
    RecordType    WALRecordType   `json:"type"`
    Timestamp     int64           `json:"timestamp"`
    TransactionID uint64          `json:"transaction_id"`
    Checksum      uint32          `json:"-"`        // Binary header field
    Data          json.RawMessage `json:"data"`
}

// WALRecordType identifies different types of operations in the WAL.
type WALRecordType uint8

const (
    RecordMessagePublished WALRecordType = 0x01
```

```
    RecordMessageAcked     WALRecordType = 0x02
    RecordMessageNacked    WALRecordType = 0x03
    RecordMessageExpired   WALRecordType = 0x04
    RecordConsumerJoined   WALRecordType = 0x05
    RecordConsumerLeft     WALRecordType = 0x06
    RecordGroupRebalanced  WALRecordType = 0x07
    RecordTopicCreated     WALRecordType = 0x08
    RecordTopicDeleted     WALRecordType = 0x09
    RecordCheckpoint       WALRecordType = 0xFF
)
```

Message State Machine Implementation (Complete Infrastructure):

GO

```go
// lifecycle.go - Complete message state machine implementation

package data

import (
    "fmt"
    "time"
)

// MessageStateMachine manages state transitions for individual messages.
type MessageStateMachine struct {
    currentState MessageState
    message      *Message
    callbacks    map[MessageState][]StateCallback
}

// StateCallback is called when a message enters a specific state.
type StateCallback func(*Message, MessageState, MessageState) error

// NewMessageStateMachine creates a state machine for tracking message lifecycle.
func NewMessageStateMachine(msg *Message) *MessageStateMachine {
    return &MessageStateMachine{
        currentState: StateCreated,
        message:      msg,
        callbacks:    make(map[MessageState][]StateCallback),
    }
}

// TransitionTo attempts to move the message to a new state.
// Returns error if transition is invalid or callback fails.
func (sm *MessageStateMachine) TransitionTo(newState MessageState, event string) error {
    if !sm.isValidTransition(sm.currentState, newState) {
        return fmt.Errorf("invalid transition from %s to %s on event %s",
            sm.currentState, newState, event)
    }

    oldState := sm.currentState
    sm.currentState = newState
```

```
    // Execute registered callbacks for this state
    for _, callback := range sm.callbacks[newState] {
        if err := callback(sm.message, oldState, newState); err != nil {
            // Rollback state change on callback failure
            sm.currentState = oldState
            return fmt.Errorf("state transition callback failed: %w", err)
        }
    }

    return nil
}

// GetState returns the current state of the message.
func (sm *MessageStateMachine) GetState() MessageState {
    return sm.currentState
}

// RegisterCallback adds a function to be called when message enters specified state.
func (sm *MessageStateMachine) RegisterCallback(state MessageState, callback StateCallback) {
    sm.callbacks[state] = append(sm.callbacks[state], callback)
}

// isValidTransition checks if the state change is allowed by our state machine rules.
func (sm *MessageStateMachine) isValidTransition(from, to MessageState) bool {
    validTransitions := map[MessageState][]MessageState{
        StateCreated:      {StatePending},
        StatePending:      {StateAssigned, StateDeadLetter},
        StateAssigned:     {StateAcknowledged, StatePending, StateDeadLetter},
        StateAcknowledged: {StateDeleted},
        StateDeadLetter:   {StatePending, StateDeleted}, // Allow replay from DLQ
        StateDeleted:      {}, // Terminal state
    }

    allowedStates, exists := validTransitions[from]
    if !exists {
        return false
    }
```

```
    for _, allowedState := range allowedStates {
        if allowedState == to {
            return true
        }
    }
    return false
}
```

Core Logic Skeletons for Implementation:

```go
// Message creation and validation - student implements this
func NewMessage(topic string, payload []byte, producerID string, headers map[string]string) (*Message, error) {
    // TODO 1: Generate unique message ID using UUID or timestamp+random
    // TODO 2: Validate topic name format (no empty strings, length limits)
    // TODO 3: Set timestamp to current Unix nanoseconds
    // TODO 4: Initialize retry count to 0 and set max retries from config
    // TODO 5: Create and return Message struct with all fields populated
    // Hint: Use time.Now().UnixNano() for timestamp
    // Hint: Use google/uuid package for ID generation
    return nil, nil
}

// Topic message queue management - student implements this
func (t *Topic) EnqueueMessage(msg *Message) error {
    t.mu.Lock()
    defer t.mu.Unlock()

    // TODO 1: Check if topic has reached retention policy limits
    // TODO 2: Append message to MessageQueue slice
    // TODO 3: Add entry to MessageIndex map (message ID -> queue position)
    // TODO 4: Update ActiveMessages and TotalMessages counters
    // TODO 5: Update LastActivity timestamp
    // Hint: Check len(t.MessageQueue) against RetentionPolicy.MaxMessages
    return nil
}

// Consumer group message assignment - student implements this
func (cg *ConsumerGroup) AssignNextMessage(msg *Message) (*Consumer, error) {
    cg.mu.Lock()
    defer cg.mu.Unlock()

    // TODO 1: Check if rebalancing is in progress, return error if so
    // TODO 2: Find available consumer with capacity (CurrentInflight < MaxInflight)
    // TODO 3: Use assignment strategy (round-robin vs sticky) to select consumer
    // TODO 4: Update message's AssignedConsumer field and set AckDeadline
    // TODO 5: Add message to consumer's PendingMessages map
    // TODO 6: Increment consumer's CurrentInflight counter
```

```
    // TODO 7: Record assignment in MessageAssignments map
    // Hint: Use time.Now().Add(30*time.Second).UnixNano() for AckDeadline
    return nil, nil
}

// WAL record writing - student implements this
func (wal *WriteAheadLog) AppendRecord(recordType WALRecordType, data interface{}) (int64, error) {
    // TODO 1: Serialize data to JSON using json.Marshal
    // TODO 2: Create WALRecord with headers (magic, version, length, type, timestamp, checksum)
    // TODO 3: Calculate CRC32 checksum of serialized data
    // TODO 4: Write binary headers to file using binary.Write
    // TODO 5: Write JSON data to file
    // TODO 6: Call file.Sync() to ensure durability
    // TODO 7: Return file offset of written record
    // Hint: Use binary.LittleEndian for consistent byte order
    // Hint: Use hash/crc32 package for checksum calculation
    return 0, nil
}
```

Milestone Checkpoints:

Milestone 1 Checkpoint - Basic Data Model:

- Run: `go test ./internal/data/... -v`
- Expected: All type definitions compile and basic message creation works
- Manual verification: Create a Message, Topic, and Consumer; verify all fields are properly initialized
- Signs of problems: Panic on field access, nil pointer dereferences, missing JSON tags

Milestone 2 Checkpoint - State Machine and Consumer Groups:

- Run: `go test ./internal/data/lifecycle_test.go -v`
- Expected: State transitions work correctly and invalid transitions are rejected
- Manual verification: Create consumer group, assign messages, verify round-robin distribution
- Signs of problems: Messages stuck in wrong state, race conditions in group assignment

Milestone 3 Checkpoint - Persistence Layer:

- Run: `go test ./internal/data/wal_test.go -v`
- Expected: WAL writes/reads work and crash recovery restores state correctly
- Manual verification: Write records to WAL, restart process, verify state restoration
- Signs of problems: Corrupted WAL files, incomplete recovery, memory leaks

Language-Specific Go Hints:

- Use `sync.RWMutex` for topic and consumer group access since reads are more frequent than writes
- Use `encoding/json` with struct tags for WAL serialization; add `json:"-"` for non-persistent fields
- Use `binary.Write` and `binary.Read` with `LittleEndian` for consistent WAL headers across platforms
- Use `os.File.Sync()` after WAL writes to ensure durability; consider batching syncs for performance

- Use `time.Now().UnixNano()` for high-precision timestamps that enable message ordering
- Use channels for state machine event notifications to avoid callback complexity
- Use `context.Context` for graceful shutdown of background cleanup goroutines

Wire Protocol Design

> **Milestone(s):** Milestone 1 (Core Queue & Wire Protocol) — defines the foundational communication layer that enables all client-broker interactions

Mental Model: Request-Response Conversations

Think of the wire protocol as **structured conversations** between clients and the message broker, similar to how postal workers use standardized forms and procedures to process mail. When you walk into a post office to send a package, there's a specific conversation pattern: you fill out a shipping form with standardized fields, the clerk validates the information, processes your request, and gives you a receipt with a tracking number. The postal worker doesn't need to guess what you want or parse free-form text — the standardized forms make the interaction predictable and efficient.

Our message queue wire protocol works the same way. Instead of clients sending arbitrary text that the broker must parse and interpret, we define a **binary protocol** with specific message formats for each type of operation. When a client wants to publish a message, it constructs a standardized `PUBLISH` command frame with well-defined fields: command type, topic name, message payload, and headers. The broker receives this frame, validates the format, processes the request, and sends back a standardized response indicating success or failure.

This conversation-based approach provides several critical benefits. First, it eliminates ambiguity — both client and broker know exactly what each message means and how to process it. Second, it enables **efficient parsing** because the broker can read fixed-size headers to determine message boundaries and types before processing the payload. Third, it supports **connection multiplexing** where multiple concurrent operations can occur on the same TCP connection without interference. Finally, it provides a foundation for **protocol evolution** where new command types can be added while maintaining backward compatibility with existing clients.

The key insight is that network communication between distributed components should be as predictable and structured as possible. Just as postal workers rely on standardized forms to process millions of packages efficiently, our message broker relies on a standardized wire protocol to handle thousands of concurrent client operations reliably.

Message Framing

Binary message framing solves one of the fundamental challenges of TCP communication: **message boundary detection**. TCP provides a reliable byte stream, but it doesn't preserve message boundaries — data sent in one `write()` call might arrive as multiple `read()` calls, or multiple `write()` calls might arrive as a single `read()`. Without proper framing, the receiver cannot determine where one message ends and the next begins.

Our wire protocol uses **length-prefixed framing** with a fixed-size header that contains all the information needed to parse the complete message. This approach is similar to how HTTP/2 frames work, but optimized for our message queue use case. Each frame begins with a magic number for protocol validation, followed by version information, frame type, and payload length. This header structure allows the receiver to read a fixed number of bytes, determine the payload size, and then read exactly that many additional bytes to get the complete message.

The frame header structure provides both efficiency and reliability. The magic number enables **protocol validation** — if a client accidentally connects using HTTP or another protocol, the broker can immediately detect the mismatch and close the connection gracefully. The version field supports **protocol evolution** where future versions can extend the frame format while maintaining compatibility with older clients. The frame type field enables **command multiplexing** where different types of operations can be distinguished without parsing the payload. The length field prevents **buffer overflow attacks** by allowing the broker to allocate exactly the right amount of memory for each message.

Field	Type	Size (bytes)	Description
Magic	uint32	4	Protocol identifier (0x4D515545 - "MQUE")
Version	uint8	1	Protocol version (currently 1)
FrameType	uint8	1	Command type (PUBLISH=1, SUBSCRIBE=2, ACK=3, NACK=4, etc.)
Flags	uint8	1	Frame flags (compression, encryption, etc.)
Reserved	uint8	1	Reserved for future use (must be 0)
PayloadLength	uint32	4	Length of payload in bytes
Total Header		**12**	Fixed header size

The payload immediately follows the header and contains the command-specific data encoded as binary fields. For variable-length strings (like topic names), we use **nested length prefixes** where each string is preceded by a 2-byte length field. This allows efficient parsing without requiring null terminators or escape sequences.

Frame parsing algorithm:

1. **Read fixed header** — Read exactly 12 bytes from the TCP connection to get the complete frame header
2. **Validate magic number** — Check that the first 4 bytes match 0x4D515545; if not, close connection with protocol error
3. **Check version compatibility** — Verify the version field is supported; newer versions should be backward compatible
4. **Validate payload length** — Ensure PayloadLength is within acceptable bounds (e.g., maximum 10MB per frame)
5. **Allocate payload buffer** — Create a buffer of exactly PayloadLength bytes to prevent over-allocation
6. **Read payload** — Read exactly PayloadLength bytes from the connection to get the complete command payload
7. **Parse command** — Use FrameType to determine payload structure and deserialize command-specific fields

This framing approach handles the common TCP streaming challenges elegantly. **Partial reads** are handled by continuing to read until the expected number of bytes is received. **Frame boundaries** are preserved because each frame is self-contained with its own length prefix. **Connection reuse** is supported because frames can be processed sequentially on the same TCP connection.

> **Critical Implementation Note:** Always read the exact number of bytes specified in the frame header. TCP may deliver data in chunks smaller than requested, so the read loop must continue until all bytes are received. Similarly, always write complete frames atomically to prevent interleaving of concurrent operations on the same connection.

Command Specification

The wire protocol supports five core commands that enable all message queue operations: `PUBLISH` for sending messages to topics, `SUBSCRIBE` for registering interest in topic messages, `ACK` and `NACK` for acknowledging message processing, and `JOIN_GROUP` for participating in consumer groups. Each command has a specific payload format optimized for its use case, with fixed-size fields for efficient parsing and variable-length fields for flexibility.

PUBLISH Command

The `PUBLISH` command allows producers to send messages to specific topics. The command payload contains the topic name, message headers, and message body, along with optional parameters like message TTL and delivery requirements.

Field	Type	Size	Description
TopicLength	uint16	2	Length of topic name in bytes
TopicName	string	variable	Target topic for message delivery
HeaderCount	uint16	2	Number of key-value header pairs
Headers	[]Header	variable	Message headers (key-length, key, value-length, value)
PayloadLength	uint32	4	Length of message payload in bytes
Payload	[]byte	variable	Actual message content
TTL	uint32	4	Time-to-live in seconds (0 = no expiration)
RequireAck	uint8	1	1 if message requires acknowledgment, 0 otherwise

PUBLISH Response:

Field	Type	Size	Description
Status	uint8	1	Response code (0=success, 1=topic not found, 2=quota exceeded, etc.)
MessageID	string	variable	Unique identifier for published message (length-prefixed)
Timestamp	uint64	8	Server timestamp when message was accepted

SUBSCRIBE Command

The `SUBSCRIBE` command registers a client's interest in receiving messages from specific topics. It supports both exact topic matching and wildcard patterns for hierarchical topic structures.

Field	Type	Size	Description
TopicLength	uint16	2	Length of topic pattern in bytes
TopicPattern	string	variable	Topic name or wildcard pattern
GroupLength	uint16	2	Length of consumer group name (0 if not using groups)
GroupName	string	variable	Consumer group identifier for load balancing
MaxInflight	uint16	2	Maximum unacknowledged messages for this subscription
AutoAck	uint8	1	1 if messages should be auto-acknowledged, 0 for manual ack

SUBSCRIBE Response:

Field	Type	Size	Description
Status	uint8	1	Response code (0=success, 1=invalid pattern, 2=group error, etc.)
SubscriptionID	string	variable	Unique identifier for this subscription (length-prefixed)

Message Delivery (Server-to-Client)

When the broker delivers messages to subscribers, it uses a standardized delivery frame format that includes all the information needed for processing and acknowledgment.

Field	Type	Size	Description
SubscriptionID	string	variable	Identifies which subscription this message relates to
MessageID	string	variable	Unique message identifier for acknowledgment
TopicName	string	variable	Actual topic name (may differ from subscription pattern)
HeaderCount	uint16	2	Number of message headers
Headers	[]Header	variable	Original message headers from publisher
PayloadLength	uint32	4	Length of message content
Payload	[]byte	variable	Message content
AckDeadline	uint64	8	Timestamp when acknowledgment is required (0 if auto-ack)

ACK Command

The `ACK` command confirms successful processing of a delivered message, allowing the broker to remove it from pending queues and update consumer group state.

Field	Type	Size	Description
MessageID	string	variable	Identifier of message being acknowledged
ProcessingTime	uint32	4	Time spent processing message in milliseconds

ACK Response:

Field	Type	Size	Description
Status	uint8	1	Response code (0=success, 1=message not found, 2=ack timeout, etc.)

NACK Command

The `NACK` command indicates that a message could not be processed successfully and should be redelivered to another consumer or moved to a dead letter queue.

Field	Type	Size	Description
MessageID	string	variable	Identifier of message being rejected
ReasonCode	uint8	1	Rejection reason (1=processing error, 2=poison message, 3=timeout, etc.)
RetryRequested	uint8	1	1 if message should be retried, 0 if it should go to DLQ

NACK Response:

Field	Type	Size	Description
Status	uint8	1	Response code (0=success, 1=message not found, etc.)
Action	uint8	1	Action taken (1=requeued, 2=sent to DLQ, 3=dropped)

Protocol Architecture Decisions

Decision: Binary vs Text Protocol

> **Decision: Use Binary Wire Protocol**
>
> - **Context**: Need efficient, unambiguous communication between clients and broker with support for binary message payloads and high throughput
> - **Options Considered**:
>   1. Text-based protocol (like Redis RESP or HTTP)
>   2. Binary protocol (like Apache Kafka or RabbitMQ AMQP)
>   3. gRPC with Protocol Buffers
> - **Decision**: Implement custom binary protocol with length-prefixed framing
> - **Rationale**: Binary protocols provide better performance for high-throughput scenarios, eliminate parsing ambiguity for binary payloads, and offer more compact representation. Text protocols require escaping for binary data and consume more bandwidth. gRPC adds complexity and dependencies that aren't justified for this learning project.
> - **Consequences**: More efficient network utilization and parsing, but requires custom client libraries and more complex debugging compared to text protocols

Option	Pros	Cons
Text Protocol	Human readable, easy debugging, simple client implementation	Inefficient binary encoding, parsing complexity, escaping required
Binary Protocol	Compact representation, fast parsing, no escaping needed	Requires custom tooling, harder to debug, more complex implementation
gRPC/Protobuf	Schema evolution, code generation, battle-tested	Heavy dependencies, learning curve, overkill for simple message queue

Decision: Synchronous vs Asynchronous Command Processing

> **Decision: Synchronous Request-Response with Async Message Delivery**
>
> - **Context**: Need to balance client simplicity with broker efficiency for command processing vs message delivery
> - **Options Considered**:
>   1. Fully synchronous (client waits for response to every command)
>   2. Fully asynchronous (fire-and-forget commands, callbacks for responses)
>   3. Hybrid (sync for commands, async for message delivery)
> - **Decision**: Use synchronous request-response for client commands (`PUBLISH` , `SUBSCRIBE` , `ACK`) with asynchronous server-initiated message delivery
> - **Rationale**: Synchronous commands provide clear error handling and flow control for client operations. Asynchronous message delivery enables high-throughput fanout without blocking the broker. This hybrid approach balances simplicity and performance.
> - **Consequences**: Clients get immediate feedback on command success/failure, but must handle asynchronous message arrival. Broker can pipeline message delivery for better throughput.

Decision: Connection Management Strategy

> **Decision: Long-Lived Persistent Connections with Heartbeat**
>
> - **Context**: Need reliable client-broker communication with efficient connection reuse and failure detection
> - **Options Considered**:
>   1. Request-per-connection (like HTTP/1.0)
>   2. Connection pooling with multiplexing
>   3. Single long-lived connection per client
> - **Decision**: Use persistent TCP connections with periodic heartbeat messages and connection-level multiplexing
> - **Rationale**: Long-lived connections avoid TCP handshake overhead and enable server-initiated message delivery. Heartbeats provide timely failure detection. Connection multiplexing allows multiple concurrent operations without additional connections.
> - **Consequences**: Better performance and lower resource usage, but requires connection state management and heartbeat processing. Client reconnection logic needed for network failures.

Option	Pros	Cons
Per-Request Connections	Stateless, simple error handling	High overhead, no server push capability
Connection Pooling	Resource efficiency, load balancing	Complex pool management, connection sharing issues
Long-Lived Connections	Low overhead, server push support	State management, failure detection complexity

Decision: Message ID Generation Strategy

> **Decision: Server-Generated UUIDs for Message IDs**
>
> - **Context**: Need globally unique message identifiers for acknowledgment tracking and duplicate detection
> - **Options Considered**:
>   1. Client-generated UUIDs
>   2. Server-generated UUIDs
>   3. Server-generated sequential IDs per topic
> - **Decision**: Broker generates UUID v4 for each message upon receipt
> - **Rationale**: Server-generated IDs ensure uniqueness without coordination, prevent client ID conflicts, and enable broker-controlled message ordering. UUIDs provide global uniqueness without requiring centralized coordination.
> - **Consequences**: Slight increase in message header size, but eliminates duplicate ID conflicts and simplifies client implementation

Common Protocol Pitfalls

⚠️ Pitfall: Partial TCP Reads

Problem: TCP is a stream protocol that doesn't preserve message boundaries. A single `read()` call may return only part of a frame, or multiple frames may arrive in a single read. Many developers assume that `read(buffer, n)` will always return exactly `n` bytes, leading to frame corruption and protocol errors.

Why it happens: TCP optimizes for throughput over message preservation. Network conditions, buffer sizes, and system scheduling can cause data to arrive in arbitrary chunks. Writing `conn.Read(headerBuffer[0:12])` once and assuming you got the complete 12-byte header will fail unpredictably in production.

How to fix: Implement a **reliable frame reader** that continues reading until the exact number of bytes is received:

```
ReadExactly(conn, buffer, expectedBytes):
  totalRead = 0
  while totalRead < expectedBytes:
    bytesRead = conn.Read(buffer[totalRead:expectedBytes])
    if bytesRead == 0:
      return error "connection closed"
    totalRead += bytesRead
  return success
```

Use this for both header reading (always 12 bytes) and payload reading (PayloadLength bytes as specified in header).

⚠️ Pitfall: Frame Boundary Corruption

Problem: Attempting to read multiple frames in a single large read operation, then manually splitting the buffer to find frame boundaries. This approach seems efficient but leads to complex parsing logic and makes error recovery nearly impossible when frame corruption occurs.

Why it happens: Developers try to optimize network I/O by reading large buffers, then splitting them into frames in application code. However, this approach requires complex buffering logic and makes it difficult to handle partial frames or protocol errors.

How to fix: Always read one complete frame at a time using the length-prefixed approach. Read the header first, extract the payload length, then read exactly that many payload bytes. This keeps frame parsing simple and makes error handling straightforward.

⚠️ Pitfall: Connection Cleanup on Client Disconnect

Problem: When clients disconnect unexpectedly (network failure, crash, etc.), their subscriptions and pending acknowledgments remain active on the broker indefinitely. This causes memory leaks and messages to be delivered to dead connections.

Why it happens: TCP doesn't immediately detect when the other end disconnects, especially if the disconnection is due to network failure rather than a clean close. The broker continues to treat the connection as active and accumulates state for the dead client.

How to fix: Implement multiple layers of connection health monitoring:

1. **TCP keepalive** — Enable SO_KEEPALIVE at the socket level to detect dead connections
2. **Application-level heartbeat** — Require clients to send periodic heartbeat frames (every 30-60 seconds)
3. **Write failure detection** — When attempting to deliver a message to a client, treat write failures as connection death
4. **Connection cleanup** — When a connection dies, immediately clean up all subscriptions, pending messages, and consumer group memberships

⚠️ Pitfall: Concurrent Write Corruption

Problem: Multiple goroutines writing frames to the same TCP connection simultaneously, causing frame interleaving and protocol corruption. For example, one goroutine writes a PUBLISH frame header while another writes a message delivery frame payload.

Why it happens: The broker needs to send responses to client commands and also deliver messages asynchronously. Without proper synchronization, concurrent writes corrupt the frame stream.

How to fix: Serialize all writes to each connection using either:

1. **Per-connection write mutex** — All frame writes acquire a mutex before writing the complete frame
2. **Per-connection write channel** — Send complete frames through a channel serviced by a dedicated writer goroutine
3. **Frame assembly** — Build complete frames in memory before writing them atomically to the connection

The channel approach is often cleaner as it naturally serializes writes and enables write buffering for better performance.

⚠️ Pitfall: Unbounded Memory Growth from Large Frames

Problem: Accepting frames with arbitrarily large PayloadLength values, allowing malicious or buggy clients to exhaust broker memory by sending enormous frames.

Why it happens: The protocol specifies PayloadLength as a uint32, which theoretically allows 4GB frames. Without validation, the broker will attempt to allocate whatever memory is requested.

How to fix: Implement frame size limits at multiple levels:

1. **Maximum frame size** — Reject any frame with PayloadLength exceeding a reasonable limit (e.g., 10MB)

2. **Connection quotas** — Limit the total memory allocated for frames from any single connection
3. **Global memory limits** — Track total frame memory usage across all connections

When a frame exceeds limits, close the connection immediately with an appropriate error code rather than attempting to skip the oversized frame.

Implementation Guidance

The wire protocol implementation requires careful handling of binary data, concurrent connections, and network edge cases. This section provides starter code for infrastructure components and skeleton code for the core protocol logic.

Technology Recommendations

Component	Simple Option	Advanced Option
TCP Server	net.Listener + goroutine per connection	Event-driven I/O with epoll/kqueue
Binary Encoding	Manual byte operations + encoding/binary	Protocol Buffers or MessagePack
Connection Management	sync.Map for connection tracking	Custom connection pool with metrics
Frame Buffering	bufio.Reader/Writer	Zero-copy ring buffers

Recommended File Structure

```
internal/protocol/
  protocol.go          ← frame definitions and constants
  handler.go           ← protocol handler implementation
  connection.go        ← connection management
  commands.go          ← command parsing and execution
  protocol_test.go     ← protocol tests
  testdata/            ← test frame samples
cmd/broker/
  main.go              ← server entry point
```

Infrastructure Starter Code

Frame Definitions (protocol.go):

```go
package protocol

import (
    "encoding/binary"
    "fmt"
)

const (
    // Protocol constants
    ProtocolMagic   = 0x4D515545 // "MQUE"
    ProtocolVersion = 1
    MaxFrameSize    = 10 * 1024 * 1024 // 10MB
    HeaderSize      = 12
)

// Frame types
const (
    FramePublish    = 1
    FrameSubscribe  = 2
    FrameAck        = 3
    FrameNack       = 4
    FrameJoinGroup  = 5
    FrameDelivery   = 10
    FrameResponse   = 11
    FrameHeartbeat  = 99
)

// Response status codes
const (
    StatusSuccess     = 0
    StatusTopicNotFound = 1
    StatusQuotaExceeded = 2
    StatusInvalidFormat = 3
    StatusUnauthorized  = 4
)

// Frame header structure
type FrameHeader struct {
```

```
    Magic         uint32
    Version       uint8
    FrameType     uint8
    Flags         uint8
    Reserved      uint8
    PayloadLength uint32
}

// Serialize frame header to bytes
func (h *FrameHeader) Marshal() []byte {
    buf := make([]byte, HeaderSize)
    binary.BigEndian.PutUint32(buf[0:4], h.Magic)
    buf[4] = h.Version
    buf[5] = h.FrameType
    buf[6] = h.Flags
    buf[7] = h.Reserved
    binary.BigEndian.PutUint32(buf[8:12], h.PayloadLength)
    return buf
}

// Parse frame header from bytes
func UnmarshalFrameHeader(data []byte) (*FrameHeader, error) {
    if len(data) < HeaderSize {
        return nil, fmt.Errorf("insufficient header data: got %d bytes, need %d", len(data), HeaderSize)
    }

    header := &FrameHeader{
        Magic:         binary.BigEndian.Uint32(data[0:4]),
        Version:       data[4],
        FrameType:     data[5],
        Flags:         data[6],
        Reserved:      data[7],
        PayloadLength: binary.BigEndian.Uint32(data[8:12]),
    }

    if header.Magic != ProtocolMagic {
```

```
        return nil, fmt.Errorf("invalid protocol magic: got 0x%x, expected 0x%x", header.Magic, ProtocolMagic)
    }

    if header.PayloadLength > MaxFrameSize {
        return nil, fmt.Errorf("frame too large: %d bytes exceeds limit of %d", header.PayloadLength, MaxFrameSize)
    }

    return header, nil
}

// Helper function to read length-prefixed strings
func ReadString(data []byte, offset int) (string, int, error) {
    if offset+2 > len(data) {
        return "", offset, fmt.Errorf("insufficient data for string length")
    }

    length := binary.BigEndian.Uint16(data[offset : offset+2])
    offset += 2

    if offset+int(length) > len(data) {
        return "", offset, fmt.Errorf("insufficient data for string content")
    }

    str := string(data[offset : offset+int(length)])
    offset += int(length)

    return str, offset, nil
}

// Helper function to write length-prefixed strings
func WriteString(buf []byte, offset int, str string) int {
    binary.BigEndian.PutUint16(buf[offset:offset+2], uint16(len(str)))
    offset += 2
    copy(buf[offset:], []byte(str))
    return offset + len(str)
}
```

Connection Management (connection.go):

```go
package protocol

import (
    "bufio"
    "net"
    "sync"
    "time"
)

type Connection struct {
    conn       net.Conn
    reader     *bufio.Reader
    writer     *bufio.Writer
    writeMux   sync.Mutex
    clientID   string
    remoteAddr string
    createdAt  time.Time
    lastSeen   time.Time
}

func NewConnection(conn net.Conn) *Connection {
    return &Connection{
        conn:       conn,
        reader:     bufio.NewReader(conn),
        writer:     bufio.NewWriter(conn),
        remoteAddr: conn.RemoteAddr().String(),
        createdAt:  time.Now(),
        lastSeen:   time.Now(),
    }
}

// ReadFrame reads exactly one complete frame from the connection
func (c *Connection) ReadFrame() (*FrameHeader, []byte, error) {
    // TODO 1: Read exactly HeaderSize bytes for frame header
    // TODO 2: Parse header using UnmarshalFrameHeader
    // TODO 3: Validate header fields (magic, version, etc.)
    // TODO 4: Read exactly PayloadLength bytes for frame payload
    // TODO 5: Update lastSeen timestamp
```

```
    // TODO 6: Return header and payload
    // Hint: Use io.ReadFull to ensure complete reads
    return nil, nil, nil
}

// WriteFrame sends a complete frame atomically
func (c *Connection) WriteFrame(frameType uint8, payload []byte) error {
    c.writeMux.Lock()
    defer c.writeMux.Unlock()

    // TODO 1: Create FrameHeader with correct magic, version, type, and payload length
    // TODO 2: Marshal header to bytes
    // TODO 3: Write header bytes to buffered writer
    // TODO 4: Write payload bytes to buffered writer
    // TODO 5: Flush buffered writer to ensure frame is sent
    // TODO 6: Handle any write errors appropriately
    return nil
}

// Close cleanly shuts down the connection
func (c *Connection) Close() error {
    return c.conn.Close()
}
```

Core Logic Skeleton Code

Protocol Handler (handler.go):

```go
package protocol

import (
    "context"
    "log"
    "net"
    "sync"
)

type Handler struct {
    connections sync.Map // map[string]*Connection
    topicManager interface{} // Will be injected from topic manager component
    // TODO: Add other component dependencies
}

func NewHandler() *Handler {
    return &Handler{}
}

// HandleConnection processes all frames from a client connection
func (h *Handler) HandleConnection(conn net.Conn) {
    connection := NewConnection(conn)
    defer h.cleanupConnection(connection)


    // TODO 1: Generate unique connection ID and store in connections map
    // TODO 2: Start heartbeat goroutine for connection monitoring
    // TODO 3: Enter main frame processing loop
    // TODO 4: For each frame received:
    //   - Call ReadFrame to get header and payload
    //   - Route to appropriate command handler based on frame type
    //   - Handle any protocol errors by closing connection
    // TODO 5: Clean up connection state when loop exits
    // Hint: Use a select statement to handle both incoming frames and heartbeat timeouts
}

// ProcessPublishCommand handles PUBLISH frame from client
func (h *Handler) ProcessPublishCommand(conn *Connection, payload []byte) error {
    // TODO 1: Parse topic name from payload (length-prefixed string)
```

```
    // TODO 2: Parse header count and iterate through headers
    // TODO 3: Parse message payload length and content
    // TODO 4: Parse TTL and acknowledgment requirements
    // TODO 5: Create Message struct with parsed data
    // TODO 6: Call topic manager to enqueue message
    // TODO 7: Generate unique message ID
    // TODO 8: Send PUBLISH response with status and message ID
    // TODO 9: Handle any validation errors (invalid topic, quota exceeded, etc.)
    return nil
}

// ProcessSubscribeCommand handles SUBSCRIBE frame from client
func (h *Handler) ProcessSubscribeCommand(conn *Connection, payload []byte) error {
    // TODO 1: Parse topic pattern from payload
    // TODO 2: Parse consumer group name (if specified)
    // TODO 3: Parse max inflight and auto-ack settings
    // TODO 4: Validate topic pattern syntax
    // TODO 5: Register subscription with topic manager
    // TODO 6: Generate unique subscription ID
    // TODO 7: Store subscription mapping for this connection
    // TODO 8: Send SUBSCRIBE response with status and subscription ID
    return nil
}
```

Milestone Checkpoint

After implementing the wire protocol:

Test Command: `go test ./internal/protocol/... -v`

Expected Behavior:

- Protocol parser correctly handles all frame types without errors
- Frame boundaries are preserved even with partial TCP reads
- Multiple concurrent connections can send/receive frames without corruption
- Invalid frames (wrong magic, oversized payload) are rejected cleanly

Manual Verification:

1. Start the broker: `go run cmd/broker/main.go`
2. Connect with netcat: `nc localhost 8080`
3. Send a binary PUBLISH frame (use a hex editor or custom test client)
4. Verify the broker responds with proper PUBLISH response frame
5. Check that connection cleanup works when you disconnect netcat

Common Issues:

- **Symptom**: "Invalid protocol magic" errors with valid clients
- **Cause**: Endianness mismatch in binary encoding
- **Fix**: Ensure consistent use of BigEndian for all multi-byte fields
- **Symptom**: Connections hang during frame reads
- **Cause**: Partial TCP reads not handled correctly
- **Fix**: Use `io.ReadFull` instead of single `Read()` calls
- **Symptom**: Frame corruption under load
- **Cause**: Concurrent writes to same connection
- **Fix**: Properly acquire write mutex before sending complete frames

Topic Manager Component

> **Milestone(s):** Milestone 1 (Core Queue & Wire Protocol) — implements the foundational message routing and topic management that enables publish/subscribe messaging patterns

Mental Model: Radio Station Broadcasting

Before diving into the technical implementation of topic management, let's build an intuitive understanding using a familiar analogy. Think of the **Topic Manager** as operating like a sophisticated radio broadcasting station that manages multiple channels simultaneously.

In the radio broadcasting world, each radio station (our topic) has a unique frequency identifier like "FM 101.5" or "AM 1010" (our topic name). When a DJ (our publisher) wants to broadcast a song or announcement (our message), they select which station frequency to transmit on. The radio tower (our Topic Manager) then amplifies that signal and broadcasts it across the airwaves to reach all listeners tuned to that frequency.

The beautiful part of radio broadcasting is the **fan-out delivery pattern** — when the DJ plays a song on FM 101.5, every car radio, home stereo, and portable device tuned to that frequency receives the exact same audio signal simultaneously. The DJ doesn't need to know how many listeners are out there, where they're located, or what type of radio they're using. This is **spatial decoupling** — the broadcaster and listeners don't need to know each other's locations or identities.

Similarly, listeners (our subscribers) can tune in and out of stations (subscribe and unsubscribe from topics) at any time without affecting the broadcast or other listeners. If someone turns off their radio during a song, the broadcast continues normally for everyone else. This demonstrates **temporal decoupling** — participants don't need to be active simultaneously for the system to function.

Modern radio systems also support **hierarchical organization** through station grouping. For example, all country music stations might use frequencies in the 95-105 FM range, while talk radio occupies 570-1700 AM. Listeners can scan for "all country stations" using pattern matching rather than memorizing every individual frequency. Our Topic Manager implements similar **wildcard subscription** capabilities, allowing subscribers to receive messages from multiple related topics using pattern matching like `music/country/*` or `news/#`.

The Topic Manager also acts like a radio station's **program director**, making decisions about which shows to keep on the air (topic retention), how long to store recordings (message persistence), and when to create new specialty channels (dynamic topic creation). Just as a radio station monitors listener engagement and adjusts programming accordingly, our Topic Manager tracks subscription patterns and message activity to optimize performance.

This mental model helps us understand that topic-based messaging is fundamentally about **decoupled broadcasting** — publishers broadcast to topics without knowing subscribers, subscribers listen to topics without knowing publishers, and the Topic Manager serves as the intelligent routing infrastructure that makes this seamless communication possible.

Topic Manager Interface

The Topic Manager serves as the central nervous system for message routing and subscription management within our message queue. It maintains the mapping between topic names and their associated subscribers, handles message distribution through fan-out delivery, and manages the lifecycle of topics from creation through cleanup. The interface provides clean abstractions for the core operations while hiding the complexity of concurrent access patterns and subscription indexing.

The Topic Manager's responsibilities extend beyond simple message routing. It must enforce retention policies, track topic statistics for monitoring, manage wildcard subscription patterns, and coordinate with the persistence layer for durable message storage. The interface design prioritizes thread safety since multiple goroutines will be publishing messages and managing subscriptions concurrently.

Method Name	Parameters	Returns	Description
`CreateTopic`	`name string, retentionPolicy RetentionPolicy`	`error`	Creates a new topic with the specified retention rules. Returns error if topic already exists or name is invalid. Initializes empty subscriber maps and message queues. Thread-safe for concurrent topic creation.
`DeleteTopic`	`name string, force bool`	`error`	Removes a topic and all its messages. If force is false, returns error if active subscribers exist. If force is true, forcibly disconnects all subscribers. Cleans up all memory and persistent storage.
`PublishMessage`	`msg Message`	`error`	Accepts a message for delivery to all subscribers of the target topic. Validates message format, enforces retention policies, persists to WAL if durability is enabled, and triggers fan-out delivery. Returns error if topic doesn't exist or message is invalid.
`Subscribe`	`consumerID string, topicPattern string, conn Connection`	`error`	Registers a consumer to receive messages from topics matching the pattern. Supports exact matches and wildcard patterns using * and # syntax. Adds consumer to all matching existing topics and remembers pattern for future topic matches.
`Unsubscribe`	`consumerID string, topicPattern string`	`error`	Removes a consumer's subscription to the specified topic pattern. Cleans up consumer references from all matching topics. If this was the consumer's last subscription, removes consumer entirely from Topic Manager state.
`UnsubscribeAll`	`consumerID string`	`error`	Removes all subscriptions for a consumer, typically called during connection cleanup. Iterates through all topics and removes consumer references. More efficient than individual unsubscribe calls during disconnect.
`GetTopicInfo`	`name string`	`TopicInfo, error`	Returns metadata about a topic including subscriber count, message count, retention settings, and creation time. Used for monitoring and administrative purposes. Returns error if topic doesn't exist.
`ListTopics`	`pattern string`	`[]string`	Returns names of all topics matching the optional pattern. If pattern is empty, returns all topics. Supports same wildcard syntax as subscriptions. Used for discovery and administrative tools.
`GetSubscribers`	`topicName string`	`[]Consumer, error`	Returns list of all consumers subscribed to a specific topic. Includes both direct subscribers and wildcard pattern matches. Used for debugging and monitoring subscriber distribution.

The interface methods handle various edge cases and error conditions gracefully. For example, `PublishMessage` validates that the target topic exists before attempting delivery, while `Subscribe` automatically creates topics if they don't exist (depending on configuration). All methods are designed to be thread-safe and can be called concurrently from multiple goroutines handling different client connections.

> **Key Design Insight**: The interface separates **topic lifecycle management** (create/delete) from **message operations** (publish/subscribe). This separation allows different authorization policies — perhaps any client can subscribe to existing topics, but only administrators can create or delete topics.

The Topic Manager maintains several internal data structures to support these operations efficiently:

Data Structure	Type	Purpose
`topics`	`map[string]*Topic`	Primary index mapping topic names to Topic objects. Protected by RWMutex for concurrent access.
`wildcardSubscriptions`	`map[string][]WildcardSubscription`	Index of active wildcard patterns and their associated consumers. Used for efficient pattern matching when new topics are created.
`consumerTopics`	`map[string][]string`	Reverse index mapping consumer IDs to their subscribed topics. Enables efficient cleanup during consumer disconnection.
`topicStats`	`map[string]*TopicStatistics`	Performance metrics for each topic including message rates, subscriber counts, and error rates. Updated atomically during message operations.

Message Fanout Algorithm

The **message fanout algorithm** is the heart of publish/subscribe messaging, responsible for delivering a single published message to all interested subscribers efficiently and reliably. When a publisher sends a message to a topic, the Topic Manager must identify all subscribers (both direct subscribers and wildcard pattern matches), create individual delivery tasks, and coordinate the delivery process while maintaining message ordering guarantees.

The fanout process must handle several complex scenarios: subscribers might be slow or unresponsive, new subscribers could join during fanout, and the system must track delivery status for reliability guarantees. The algorithm is designed to be non-blocking for the publisher — once the message is accepted and persisted, the publisher receives confirmation while fanout happens asynchronously.

Here's the step-by-step fanout algorithm that executes when `PublishMessage` is called:

1. **Message Validation and Enrichment**: The Topic Manager first validates the incoming message structure, ensuring required fields like Topic and Payload are present and valid. It generates a unique message ID if not provided, sets the timestamp to current time, and applies any topic-level message transformation rules.
2. **Topic Resolution and Creation**: The algorithm resolves the target topic name, checking if the topic exists in the `topics` map. If the topic doesn't exist and auto-creation is enabled in `BrokerConfig`, a new topic is created with default retention policies. If auto-creation is disabled, the algorithm returns an error immediately.
3. **Retention Policy Enforcement**: Before accepting the message, the algorithm checks the topic's `RetentionPolicy` to ensure adding this message wouldn't violate size or count limits. If the topic is at capacity, it may need to evict old messages first, depending on the retention strategy (FIFO eviction vs. reject new messages).
4. **Persistence Checkpoint**: If the topic has persistence enabled, the algorithm writes the message to the Write-Ahead Log using `AppendRecord(RecordMessagePublished, messageData)`. The algorithm waits for the fsync to complete before proceeding, ensuring durability. This is a synchronous step that affects publisher latency but guarantees message durability.
5. **Subscriber Discovery**: The algorithm builds a complete list of delivery targets by combining direct subscribers from the topic's `Subscribers` map with consumers that have wildcard subscriptions matching this topic. For wildcard matching, it iterates through `wildcardSubscriptions` and tests each pattern against the topic name using the pattern matching rules.
6. **Message Queuing**: For each identified subscriber, the algorithm creates a delivery task containing the message copy, target consumer information, and delivery metadata (attempt count, deadline, etc.). These tasks are added to the appropriate delivery queues — either individual consumer queues for guaranteed delivery or a shared queue for best-effort delivery.
7. **Asynchronous Delivery Coordination**: The algorithm hands off the delivery tasks to worker goroutines that handle the actual message transmission to consumers. The number of delivery workers is configurable and affects both throughput and resource usage. The Topic Manager tracks in-flight deliveries for monitoring and flow control purposes.
8. **Delivery Status Tracking**: As delivery workers complete their tasks (successfully or with failures), they report back to the Topic Manager. Successful deliveries are logged for monitoring, while failed deliveries may trigger retry logic or dead letter queue routing depending on the consumer's acknowledgment settings.

The fanout algorithm must handle several challenging edge cases:

> **Concurrency Challenge**: New subscribers might join the topic after step 5 (subscriber discovery) but before delivery completes. The algorithm handles this by taking a snapshot of subscribers at the time of publish — new subscribers won't receive this message but will receive all subsequent messages.

⚠️ **Pitfall: Blocking Fanout on Slow Consumers** A common mistake is implementing synchronous fanout where the publisher waits for all deliveries to complete. This creates a **cascading performance problem** — if any subscriber is slow or unresponsive, it delays message confirmation for the publisher and blocks the Topic Manager thread. Always implement asynchronous fanout with delivery confirmation happening independently of publisher confirmation.

⚠️ **Pitfall: Memory Explosion with Large Fan-out** When a message must be delivered to thousands of subscribers, naive implementations create individual message copies for each subscriber, potentially using gigabytes of memory for a single message. Instead, use **copy-on-write semantics** where all delivery tasks initially reference the same message object, only creating copies if the message needs consumer-specific modifications.

The fanout algorithm also supports **delivery prioritization** where certain subscribers (like monitoring systems or critical applications) receive messages before others. This is implemented through priority queues in the delivery coordination step, ensuring that high-priority deliveries happen first even under load.

Fanout Metric	Purpose	Calculation
`FanoutSize`	Number of delivery targets for this message	Count of direct subscribers + wildcard matches
`FanoutLatency`	Time from publish to last delivery completion	Max delivery time across all consumers
`FanoutErrors`	Number of failed deliveries for this message	Count of delivery tasks that exhausted retries
`FanoutEfficiency`	Percentage of successful deliveries	(Successful deliveries / Total deliveries) * 100

Wildcard Subscription Matching

Wildcard subscription matching enables subscribers to receive messages from multiple related topics using pattern-based subscriptions instead of explicitly subscribing to each individual topic. This feature is essential for building scalable systems where topics are created dynamically or follow hierarchical naming conventions. The matching system must be both flexible enough to support complex patterns and efficient enough to evaluate thousands of patterns against each published message without introducing significant latency.

Our wildcard implementation follows the **hierarchical topic naming convention** used by MQTT and other messaging systems, where topic names are structured as path-like hierarchies using forward slashes as separators (e.g., `sensors/temperature/building1/floor2`). This structure enables powerful pattern matching while keeping the syntax intuitive for developers.

The wildcard syntax supports two special characters with distinct behaviors:

Wildcard Character	Behavior	Example Pattern	Matches	Doesn't Match
`*` (single-level)	Matches exactly one level in the hierarchy	`sensors/*/building1`	`sensors/temperature/building1`, `sensors/humidity/building1`	`sensors/building1`, `sensors/temp/room/building1`
`#` (multi-level)	Matches zero or more levels at the end of pattern	`sensors/temperature/#`	`sensors/temperature`, `sensors/temperature/building1`, `sensors/temperature/building1/floor2`	`sensors/humidity/building1`

The pattern matching algorithm processes subscriptions and published topics through several phases:

Pattern Normalization and Validation: When a client subscribes with a wildcard pattern, the Topic Manager first normalizes the pattern by removing redundant slashes, validating the wildcard placement rules (# can only appear at the end), and converting it to a canonical form for

efficient matching. Invalid patterns like `sensors/#/building1` or `sensors/*/` are rejected with descriptive error messages.

Subscription Indexing Strategy: Rather than testing every wildcard pattern against every published message, the Topic Manager uses an **intelligent indexing strategy** to minimize pattern evaluations. Patterns are indexed by their non-wildcard prefixes, so patterns like `sensors/temperature/#` and `sensors/humidity/*` are grouped under the `sensors/` prefix. When a message is published to `sensors/temperature/building1`, only patterns with matching prefixes need evaluation.

Hierarchical Pattern Matching Algorithm: The core matching algorithm splits both the pattern and the topic name into level segments and compares them level by level:

1. **Segment Splitting**: Both pattern and topic are split on `/` delimiters into arrays of segments. Empty segments from duplicate slashes are filtered out during normalization.
2. **Level-by-Level Comparison**: The algorithm walks through segments simultaneously. Exact string matches advance both pattern and topic pointers. Single-level wildcards (`*`) match any single topic segment and advance both pointers.
3. **Multi-Level Wildcard Handling**: When encountering `#`, the algorithm immediately returns a match since `#` consumes all remaining topic segments. The `#` wildcard can only appear as the final pattern segment.
4. **Termination Conditions**: A match succeeds if both pattern and topic segments are exhausted simultaneously, or if the pattern ends with `#`. A match fails if either has remaining segments when the other is exhausted (unless the pattern remainder is only `#`).

Here's a concrete example of the matching process for pattern `sensors/*/building1/#` against topic `sensors/temperature/building1/floor2/room5`:

Step	Pattern Segment	Topic Segment	Action	Result
1	`sensors`	`sensors`	Exact match	Continue
2	`*`	`temperature`	Wildcard match	Continue
3	`building1`	`building1`	Exact match	Continue
4	`#`	`floor2`	Multi-level wildcard	**MATCH**

The Topic Manager maintains an optimized data structure for wildcard subscriptions:

```
wildcardIndex := map[string][]WildcardSubscription{
    "sensors/":     [{Pattern: "sensors/#", ConsumerID: "monitor1"},
                     {Pattern: "sensors/*/alerts", ConsumerID: "alerter"}],
    "logs/":        [{Pattern: "logs/*/error", ConsumerID: "errorHandler"}],
    "events/user/": [{Pattern: "events/user/#", ConsumerID: "analytics"}],
}
```

This indexing strategy reduces the number of pattern evaluations from O(total patterns) to O(patterns with matching prefix), significantly improving performance when hundreds or thousands of wildcard subscriptions are active.

> **Performance Optimization**: For topics with many wildcard subscribers, the pattern matching can become a bottleneck. The Topic Manager implements **pattern compilation** where frequently used patterns are pre-compiled into optimized matcher functions, similar to how regular expression engines optimize patterns.

⚠️ **Pitfall: Overly Broad Wildcard Patterns** Subscribers using patterns like `#` or `*/#` will receive messages from ALL topics, potentially overwhelming the consumer and degrading broker performance. Implement **subscription validation** that warns about or rejects overly broad patterns, and monitor subscription patterns to detect misconfigured clients.

⚠️ **Pitfall: Incorrect Multi-Level Wildcard Placement** The `#` wildcard has strict placement rules — it must be the final segment and must be preceded by a separator or be the entire pattern. Patterns like `sensors/#/building1` or `#/temperature` are invalid. The subscription validation must catch these errors and provide clear feedback about correct wildcard usage.

The wildcard matching system also supports **subscription optimization** where the Topic Manager detects redundant patterns (e.g., if a consumer subscribes to both `sensors/#` and `sensors/temperature/*`, the latter is redundant) and **pattern inheritance** where new topics automatically notify all matching wildcard subscribers without requiring explicit subscription management.

Topic Manager Architecture Decisions

The Topic Manager's design involves several critical architectural decisions that impact performance, scalability, and operational characteristics. Each decision represents a trade-off between competing concerns, and understanding these trade-offs is essential for both implementation and future system evolution.

Decision: In-Memory Topic Storage vs. Disk-Backed Storage

- **Context**: Topics and their subscriber lists need persistent storage that survives broker restarts, but performance is critical for message routing operations.
- **Options Considered**: Pure in-memory with periodic snapshots, direct disk storage with caching layer, hybrid approach with memory primary and disk backup
- **Decision**: Hybrid approach with in-memory primary storage and asynchronous disk persistence
- **Rationale**: In-memory storage provides sub-millisecond topic lookup and subscriber enumeration required for high-throughput message routing. Disk persistence through WAL ensures durability without blocking message operations. This combination offers both performance and reliability.
- **Consequences**: Requires careful memory management as topic count grows, adds complexity for managing memory/disk consistency, but enables horizontal scaling through topic partitioning

Option	Pros	Cons	Performance Impact
Pure In-Memory	Fastest lookup (O(1) topic access)	Lost on restart without complex snapshots	10-100x faster than disk
Direct Disk Storage	Durable, handles unlimited topics	High latency (10-100ms per operation)	Severe bottleneck for routing
Hybrid (Chosen)	Fast routing + durability	Complex consistency management	Fast routing, slow recovery

Decision: Wildcard Subscription Indexing Strategy

- **Context**: Supporting thousands of wildcard patterns against high-frequency message publishing requires efficient pattern matching that doesn't become a system bottleneck.
- **Options Considered**: Linear scan of all patterns, trie-based indexing by pattern prefixes, compiled pattern matching with caching
- **Decision**: Trie-based indexing with compiled pattern optimization for frequently used patterns
- **Rationale**: Linear scanning becomes O(n) bottleneck with many patterns. Trie indexing reduces average case to O(log depth) while maintaining simplicity. Pattern compilation provides additional optimization for hot paths without excessive complexity.
- **Consequences**: More complex subscription management code, memory overhead for trie structures, but enables scaling to thousands of concurrent wildcard subscriptions

Option	Complexity	Memory Usage	Match Performance	Chosen
Linear Scan	Low	Minimal	O(patterns)	No
Trie Index	Medium	Moderate	O(log depth)	**Yes**
Compiled Patterns	High	High	O(1) cached	Optimization

Decision: Topic Auto-Creation Policy

- **Context**: Publishers may reference topics that don't exist yet. The system must decide whether to create topics automatically or require explicit topic creation.
- **Options Considered**: Always auto-create with default settings, never auto-create (explicit creation required), configurable auto-creation with validation rules
- **Decision**: Configurable auto-creation with topic name validation and default retention policies
- **Rationale**: Auto-creation reduces operational overhead and enables dynamic systems, but uncontrolled topic creation can lead to resource exhaustion. Configurable policy with validation provides flexibility while maintaining control.
- **Consequences**: Requires careful default policies to prevent abuse, adds configuration complexity, but enables both strict control (production) and ease of use (development)

Decision: Subscription Management Threading Model

- **Context**: Subscribe/unsubscribe operations must coordinate with concurrent message publishing and fanout operations without causing deadlocks or performance degradation.
- **Options Considered**: Single-threaded with global lock, per-topic locks with careful ordering, lock-free with atomic operations and copy-on-write
- **Decision**: Per-topic reader-writer locks with hierarchical lock ordering to prevent deadlocks
- **Rationale**: Single global lock creates unnecessary contention between unrelated topics. Per-topic locks allow concurrent operations on different topics while maintaining consistency. Reader-writer semantics optimize for the common case of many concurrent message publishes with occasional subscription changes.
- **Consequences**: More complex lock management with potential deadlock risks, requires careful lock ordering discipline, but enables high concurrency for topic operations

The threading model uses a specific lock ordering to prevent deadlocks:

Lock Type	Scope	Hold Duration	Ordering Priority
`topicMapLock`	Global topic map	Brief (map operations only)	1 (highest)
`topic.subscriberLock`	Per-topic subscriber list	Brief (list modifications)	2
`wildcardIndexLock`	Wildcard pattern index	Brief (pattern updates)	3 (lowest)

Decision: Message Delivery Semantics for New Subscribers

- **Context**: When a new subscriber joins a topic, the system must decide whether they receive messages published before their subscription or only receive new messages going forward.
- **Options Considered**: Immediate delivery (new messages only), replay from beginning of topic, configurable replay depth with timestamp or offset
- **Decision**: Immediate delivery with optional replay capability through administrative API
- **Rationale**: Most pub/sub use cases expect real-time message delivery without historical replay. Historical replay is complex and resource-intensive. Providing replay as an optional administrative feature separates concerns and keeps the common path simple.
- **Consequences**: Simpler subscription logic and memory management, some use cases require external message history storage, but provides clear and predictable behavior for most applications

The Topic Manager's configuration reflects these architectural decisions:

Configuration Option	Default Value	Purpose	Impact
`AutoCreateTopics`	`true`	Enable automatic topic creation on publish	Development ease vs. production control
`DefaultRetentionMessages`	`10000`	Maximum messages per auto-created topic	Memory usage vs. message history
`WildcardIndexingEnabled`	`true`	Enable optimized wildcard pattern indexing	Performance vs. memory usage
`MaxWildcardPatterns`	`1000`	Limit wildcard subscriptions per consumer	Resource protection vs. flexibility
`TopicCleanupInterval`	`300s`	Frequency of unused topic cleanup	Resource efficiency vs. recreation overhead

These decisions create a Topic Manager that balances performance, scalability, and operational simplicity while providing clear extension points for future enhancements like topic partitioning, cross-datacenter replication, or content-based routing.

Common Topic Management Pitfalls

Topic management appears straightforward in simple scenarios, but several subtle pitfalls commonly trap developers implementing production message queue systems. These issues often manifest only under load or in edge cases, making them particularly dangerous since they may not surface during initial development and testing.

⚠️ **Pitfall: Memory Leaks from Abandoned Topics Problem Description**: Topics that no longer have active subscribers or publishers continue consuming memory indefinitely. This happens when applications create topics dynamically (like per-user notification topics) but don't explicitly clean them up when they're no longer needed. Over time, these "ghost topics" accumulate and can exhaust broker memory.

Why It's Wrong: In-memory topic storage assumes that topics are relatively long-lived and actively used. When topics accumulate without bounds, the broker eventually runs out of memory and crashes. Even worse, the memory consumption grows slowly, so the problem may not be noticed until the broker has accumulated thousands of unused topics over weeks or months.

How to Detect: Monitor topic count growth over time and track per-topic activity metrics. Topics that haven't received messages or subscription activity for extended periods are candidates for cleanup. Warning signs include steady memory growth despite stable application usage, and topic counts that grow much faster than actual application needs.

How to Fix: Implement **topic lifecycle management** with configurable retention policies. Topics should automatically be marked for cleanup after a configurable period of inactivity (e.g., no messages published and no active subscribers for 24 hours). Provide administrative APIs for manual topic cleanup and bulk operations. Consider implementing topic expiration times that applications can set when creating topics.

⚠️ **Pitfall: Race Conditions in Fanout Delivery Problem Description**: When messages are being delivered to subscribers concurrently, race conditions can occur if new subscribers join or existing subscribers disconnect during the fanout process. This can result in some subscribers missing messages, duplicate deliveries, or attempts to deliver to disconnected clients.

Why It's Wrong: Fanout delivery must be atomic from the perspective of topic state — either a subscriber exists during message publication and receives the message, or they don't exist and don't receive it. Race conditions violate this atomicity and create unpredictable message delivery behavior that's extremely difficult to debug in production.

How to Detect: Look for inconsistent delivery patterns where some subscribers occasionally miss messages despite being connected, or where disconnected clients show up in delivery error logs. Message delivery metrics that don't add up correctly (e.g., fanout size doesn't match actual deliveries) indicate race conditions.

How to Fix: Use **snapshot-based delivery** where the fanout algorithm takes a consistent snapshot of all subscribers at the moment of message publication. All delivery operations use this snapshot, ensuring that subscriber list changes during delivery don't affect the current message. Implement proper reader-writer locking where message publication takes a read lock and subscription changes take a write lock.

⚠️ **Pitfall: Wildcard Pattern Performance Degradation Problem Description**: As the number of wildcard subscriptions increases, the pattern matching overhead for each published message grows significantly. Naive implementations test every wildcard pattern against every published topic, creating O(patterns × messages) computational complexity that can overwhelm the broker under load.

Why It's Wrong: Wildcard pattern matching should scale sublinearly with the number of patterns through intelligent indexing. Linear growth means that doubling the number of wildcard subscriptions doubles the CPU cost of message publishing, eventually making the system unusable as subscription counts grow.

How to Detect: Monitor CPU usage during message publishing operations. If CPU utilization increases significantly as wildcard subscription count grows (even with constant message rates), pattern matching is becoming a bottleneck. Profile the code to confirm that pattern matching functions

consume disproportionate CPU time.

How to Fix: Implement **prefix-based pattern indexing** where patterns are grouped by their non-wildcard prefixes, so only relevant patterns are evaluated for each topic. For frequently used patterns, compile them into optimized matcher functions. Consider pattern hierarchy optimization where broad patterns (like `#`) are checked before more specific patterns to short-circuit evaluation.

⚠️ **Pitfall: Topic Name Injection and Security Vulnerabilities Problem Description**: Allowing unvalidated topic names from clients can lead to various security and operational issues. Malicious clients might create topics with names containing special characters, extremely long names, or names that exploit filesystem limitations if topics are persisted to disk.

Why It's Wrong: Topic names often become part of file paths, monitoring metric names, or administrative interfaces. Unvalidated names can break these integrations or create security vulnerabilities through path traversal attacks (e.g., topic names containing `../` sequences).

How to Detect: Monitor for topics with unusual names containing special characters, very long names, or suspicious patterns like path traversal sequences. File system errors or monitoring system errors related to metric names often indicate problematic topic names.

How to Fix: Implement **strict topic name validation** using regular expressions or allowlists. Reject names containing special characters outside a safe set (alphanumeric, hyphens, underscores, forward slashes for hierarchy). Enforce maximum name length limits (e.g., 255 characters). Sanitize topic names before using them in file paths or metric names.

⚠️ **Pitfall: Inefficient Subscription Cleanup During Mass Disconnections Problem Description**: When many clients disconnect simultaneously (e.g., during network partitions or rolling application deployments), the subscription cleanup process can become a bottleneck if each disconnect triggers individual cleanup operations for every subscribed topic.

Why It's Wrong: Individual subscription cleanup operations require acquiring locks and updating data structures for each topic. During mass disconnections, this creates a thundering herd effect where hundreds of concurrent cleanup operations compete for the same locks, degrading broker performance exactly when it needs to be most responsive.

How to Detect: Monitor broker latency during periods of high client disconnection. If response times spike when multiple clients disconnect simultaneously, subscription cleanup is becoming a bottleneck. Look for lock contention in profiling data during these periods.

How to Fix: Implement **batched subscription cleanup** where disconnection events are queued and processed in batches rather than individually. Use background cleanup goroutines that process disconnection queues periodically, reducing lock contention. Consider lazy cleanup where disconnected clients are marked as inactive but not immediately removed, then cleaned up during background maintenance cycles.

⚠️ **Pitfall: Unbounded Topic Creation Attack Problem Description**: With auto-topic creation enabled, malicious or misconfigured clients can create unlimited numbers of topics, exhausting broker memory or hitting filesystem limits. This is especially problematic when topic names are generated programmatically with unique suffixes.

Why It's Wrong: Topic creation consumes memory and other resources. Without limits, a single client can effectively perform a denial-of-service attack by creating millions of topics rapidly. Even non-malicious clients with bugs (like using timestamps or UUIDs in topic names) can accidentally create huge numbers of topics.

How to Detect: Monitor topic creation rates and total topic counts. Rapid topic creation (especially from single clients) or exponential topic count growth indicates potential abuse. Check for topic naming patterns that suggest programmatic generation with unique identifiers.

How to Fix: Implement **topic creation rate limiting** per client and globally. Set maximum limits on total topic count and per-client topic creation rates. Require authentication and authorization for topic creation in production environments. Monitor and alert on unusual topic creation patterns.

Pitfall Category	Detection Method	Prevention Strategy	Recovery Approach
Memory Leaks	Monitor memory growth + topic count	Automatic topic cleanup policies	Manual cleanup + restart if severe
Race Conditions	Delivery inconsistencies + error logs	Snapshot-based delivery + proper locking	Fix code + message replay if needed
Performance Issues	CPU profiling + response time monitoring	Optimized indexing + pattern compilation	Add indexing + scale horizontally
Security Issues	Topic name auditing + error monitoring	Strict validation + sanitization	Rename/delete bad topics + patch validation
Mass Disconnect Issues	Latency spikes during disconnections	Batched cleanup + background processing	Restart broker if cleanup queue backs up
Resource Exhaustion	Creation rate monitoring + limits	Rate limiting + authentication	Block creation + cleanup unused topics

Implementation Guidance

The Topic Manager bridges the gap between the wire protocol's command processing and the actual message storage and delivery infrastructure. For junior developers, understanding how to structure this component properly is crucial since it handles both high-frequency operations (message publishing) and low-frequency administrative operations (subscription management) with different performance characteristics.

Technology Recommendations

Component	Simple Option	Advanced Option
Topic Storage	`map[string]*Topic` with `sync.RWMutex`	Concurrent hash map with fine-grained locking
Pattern Matching	Regular expression with `regexp.Compile()`	Custom trie-based matcher with compiled patterns
Message Delivery	Goroutine per subscriber with channels	Worker pool with job queue and priorities
Cleanup Management	`time.Ticker` with periodic cleanup	Heap-based priority queue for precise timing
Configuration	YAML config file with `gopkg.in/yaml.v3`	JSON schema validation with hot reload

Start with the simple options for Milestone 1, then upgrade to advanced options as performance requirements increase. The simple approach using Go's built-in `map` and `sync.RWMutex` can easily handle thousands of topics and subscribers before becoming a bottleneck.

Recommended File Structure

```
project-root/
  internal/
    topic/
      manager.go              ← main TopicManager implementation
      manager_test.go         ← unit tests for core functionality
      wildcard.go             ← pattern matching algorithms
      wildcard_test.go        ← pattern matching test cases
      cleanup.go              ← background cleanup and maintenance
      types.go                ← Topic, Subscription data structures
      metrics.go              ← performance monitoring and stats
    protocol/
      handler.go              ← calls into topic manager
    storage/
      wal.go                  ← persistence layer integration
```

GO

This structure separates concerns cleanly — pattern matching logic is isolated in `wildcard.go`, background maintenance in `cleanup.go`, and core routing logic in `manager.go`. This makes testing easier and enables parallel development of different features.

Infrastructure Starter Code (Complete)

Here's a complete configuration management system for the Topic Manager that you can use as-is:

```go
// internal/topic/config.go

package topic

import (
    "fmt"
    "time"
    "gopkg.in/yaml.v3"
    "os"
)

type Config struct {
    // Topic creation policies
    AutoCreateTopics      bool          `yaml:"auto_create_topics"`
    MaxTopicsPerBroker    int           `yaml:"max_topics_per_broker"`
    TopicNameMaxLength    int           `yaml:"topic_name_max_length"`

    // Default retention policies for auto-created topics
    DefaultRetention struct {
        MaxMessages int64         `yaml:"max_messages"`
        MaxAge      time.Duration `yaml:"max_age"`
        MaxSize     int64         `yaml:"max_size"`
    } `yaml:"default_retention"`

    // Wildcard subscription settings
    MaxWildcardPatterns  int `yaml:"max_wildcard_patterns"`
    WildcardIndexing     bool `yaml:"wildcard_indexing_enabled"`

    // Cleanup and maintenance
    CleanupInterval      time.Duration `yaml:"cleanup_interval"`
    InactivityThreshold  time.Duration `yaml:"inactivity_threshold"`

    // Performance tuning
    DeliveryWorkers      int `yaml:"delivery_workers"`
    MaxConcurrentFanout  int `yaml:"max_concurrent_fanout"`
}

func LoadConfig(filename string) (*Config, error) {
```

```
    data, err := os.ReadFile(filename)
    if err != nil {
        return nil, fmt.Errorf("reading config file: %w", err)
    }

    var config Config
    if err := yaml.Unmarshal(data, &config); err != nil {
        return nil, fmt.Errorf("parsing config: %w", err)
    }

    // Set defaults for missing values
    setDefaults(&config)

    // Validate configuration
    if err := validateConfig(&config); err != nil {
        return nil, fmt.Errorf("invalid config: %w", err)
    }

    return &config, nil
}

func setDefaults(config *Config) {
    if config.MaxTopicsPerBroker == 0 {
        config.MaxTopicsPerBroker = 10000
    }
    if config.TopicNameMaxLength == 0 {
        config.TopicNameMaxLength = 255
    }
    if config.DefaultRetention.MaxMessages == 0 {
        config.DefaultRetention.MaxMessages = 10000
    }
    if config.DefaultRetention.MaxAge == 0 {
        config.DefaultRetention.MaxAge = 24 * time.Hour
    }
    if config.CleanupInterval == 0 {
        config.CleanupInterval = 5 * time.Minute
```

```
    }
    if config.DeliveryWorkers == 0 {
        config.DeliveryWorkers = 10
    }
}

func validateConfig(config *Config) error {
    if config.MaxTopicsPerBroker <= 0 {
        return fmt.Errorf("max_topics_per_broker must be positive")
    }
    if config.DeliveryWorkers <= 0 {
        return fmt.Errorf("delivery_workers must be positive")
    }
    if config.CleanupInterval < time.Second {
        return fmt.Errorf("cleanup_interval must be at least 1 second")
    }
    return nil
}
```

Core Logic Skeleton Code

Here's the skeleton for the main TopicManager with detailed TODOs mapping to the algorithms described above:

```go
// internal/topic/manager.go

package topic

import (
    "fmt"
    "sync"
    "time"
)

type TopicManager struct {
    // Core data structures
    topics     map[string]*Topic
    topicsLock sync.RWMutex

    // Wildcard subscription management
    wildcardSubscriptions map[string][]*WildcardSubscription
    wildcardLock         sync.RWMutex

    // Consumer tracking for cleanup
    consumerTopics map[string][]string
    consumerLock   sync.RWMutex

    // Configuration and dependencies
    config     *Config
    walWriter  WALWriter  // For persistence
    metrics    *Metrics   // For monitoring

    // Background processing
    cleanupTicker *time.Ticker
    stopChan      chan struct{}
}

func NewTopicManager(config *Config, walWriter WALWriter) *TopicManager {
    tm := &TopicManager{
        topics:               make(map[string]*Topic),
        wildcardSubscriptions: make(map[string][]*WildcardSubscription),
        consumerTopics:       make(map[string][]string),
```

```
        config:             config,
        walWriter:          walWriter,
        metrics:            NewMetrics(),
        stopChan:           make(chan struct{}),
    }

    // Start background cleanup if enabled
    if config.CleanupInterval > 0 {
        tm.cleanupTicker = time.NewTicker(config.CleanupInterval)
        go tm.backgroundCleanup()
    }

    return tm
}

// PublishMessage implements the core fanout algorithm described above.
func (tm *TopicManager) PublishMessage(msg Message) error {
    // TODO 1: Validate message structure and required fields
    //   - Check that msg.Topic is not empty and passes topic name validation
    //   - Check that msg.Payload is not nil and within size limits
    //   - Generate unique msg.ID if not provided using UUID or similar
    //   - Set msg.Timestamp to current Unix timestamp

    // TODO 2: Resolve target topic and handle auto-creation
    //   - Lock topics map for reading: tm.topicsLock.RLock()
    //   - Look up topic in tm.topics map
    //   - If topic doesn't exist and tm.config.AutoCreateTopics is true:
    //     - Upgrade to write lock, create new topic with default retention
    //     - Add to tm.topics map
    //   - If topic doesn't exist and auto-creation disabled, return error

    // TODO 3: Enforce retention policies before accepting message
    //   - Check if adding this message would exceed topic's MaxMessages limit
    //   - Check if adding this message would exceed topic's MaxSize limit
    //   - If at capacity, either evict old messages (FIFO) or reject new message
    //   - Update topic.TotalMessages and topic.MessageQueue
```

```
    // TODO 4: Write to persistent storage if durability enabled
    //   - If topic has persistence enabled, call tm.walWriter.AppendRecord()
    //   - Use RecordMessagePublished record type with message data
    //   - Wait for fsync to complete before proceeding (ensures durability)
    //   - If write fails, return error without proceeding to delivery

    // TODO 5: Build complete subscriber list (direct + wildcard matches)
    //   - Get direct subscribers from topic.Subscribers map
    //   - Call tm.findWildcardMatches(msg.Topic) to get wildcard subscribers
    //   - Combine into single list, removing duplicates if same consumer
    //   - If no subscribers found, still log message but skip delivery

    // TODO 6: Create delivery tasks for asynchronous fanout
    //   - For each subscriber, create DeliveryTask with message copy and target
    //   - Add delivery metadata: attempt count=1, deadline=now+timeout
    //   - Queue delivery task to delivery worker pool using channels
    //   - Don't wait for delivery completion - return success immediately

    // TODO 7: Update metrics and monitoring counters
    //   - Increment topic.TotalMessages counter atomically
    //   - Record fanout size in tm.metrics.FanoutSize histogram
    //   - Update topic.LastActivity timestamp for cleanup tracking
    //   - Log successful message acceptance for debugging

    return nil
}

// Subscribe registers a consumer for topic pattern (exact or wildcard).
func (tm *TopicManager) Subscribe(consumerID string, pattern string, conn Connection) error {
    // TODO 1: Validate subscription parameters
    //   - Check that consumerID is not empty and valid format
    //   - Validate topic pattern using tm.validateTopicPattern(pattern)
    //   - Check that consumer doesn't exceed max subscription limits
    //   - Ensure connection is active and authenticated
```

```go
    // TODO 2: Handle wildcard vs exact topic subscriptions differently
    //   - If pattern contains '*' or '#', process as wildcard subscription
    //   - For wildcard: add to tm.wildcardSubscriptions with pattern indexing
    //   - For exact topics: add directly to topic's Subscribers map
    //   - Create topic if it doesn't exist (if auto-creation enabled)

    // TODO 3: Update consumer tracking for efficient cleanup
    //   - Add topic/pattern to tm.consumerTopics[consumerID] list
    //   - This enables fast cleanup when consumer disconnects
    //   - Use proper locking to avoid race conditions

    // TODO 4: Handle existing topic matching for wildcard subscriptions
    //   - If this is wildcard pattern, find all existing topics that match
    //   - Add consumer as subscriber to all matching existing topics
    //   - This ensures consumer gets messages from topics created before subscription

    // TODO 5: Send subscription confirmation and setup delivery channel
    //   - Create consumer entry with connection details and preferences
    //   - Setup delivery channel/queue for this consumer
    //   - Send SUBSCRIBE_OK response to client with subscription details
    //   - Log successful subscription for monitoring and debugging

    return nil
}

// Unsubscribe removes consumer from topic pattern.
func (tm *TopicManager) Unsubscribe(consumerID string, pattern string) error {
    // TODO 1: Locate and validate existing subscription
    //   - Find consumer in tm.consumerTopics map
    //   - Verify that consumer is actually subscribed to this pattern
    //   - Return error if subscription doesn't exist

    // TODO 2: Remove from appropriate subscription index
    //   - For wildcard patterns: remove from tm.wildcardSubscriptions
    //   - For exact topics: remove from topic.Subscribers map
    //   - Handle case where topic might have been deleted already
```

```
    // TODO 3: Clean up consumer tracking data
    //   - Remove pattern from tm.consumerTopics[consumerID] list
    //   - If consumer has no remaining subscriptions, remove consumer entirely
    //   - Update any delivery queues to stop sending to this consumer

    // TODO 4: Send unsubscribe confirmation
    //   - Send UNSUBSCRIBE_OK response to client
    //   - Log unsubscription for monitoring and debugging
    //   - Update subscription metrics counters

    return nil
}

// findWildcardMatches returns all wildcard subscriptions matching the topic name.
func (tm *TopicManager) findWildcardMatches(topicName string) []*Consumer {
    // TODO 1: Get topic segments for matching
    //   - Split topicName on '/' delimiter into segments
    //   - Remove any empty segments from duplicate slashes
    //   - This creates the segment array for pattern matching

    // TODO 2: Check each wildcard pattern using efficient indexing
    //   - Use prefix indexing to only check patterns with matching prefixes
    //   - For each candidate pattern, call tm.matchPattern(pattern, segments)
    //   - Collect all consumers from matching patterns

    // TODO 3: Remove duplicate consumers
    //   - Same consumer might match multiple patterns for same topic
    //   - Use map[string]*Consumer to deduplicate by consumer ID
    //   - Return final list of unique consumers

    return nil
}

// matchPattern implements the hierarchical wildcard matching algorithm.
func (tm *TopicManager) matchPattern(pattern string, topicSegments []string) bool {
```

```
    // TODO 1: Parse pattern into segments
    //   - Split pattern on '/' delimiter, handle empty segments
    //   - Validate pattern format (e.g., # only at end)

    // TODO 2: Implement level-by-level matching
    //   - Walk through pattern and topic segments simultaneously
    //   - Handle exact matches, single-level wildcards (*), multi-level wildcards (#)
    //   - Return true if match succeeds, false otherwise
    //   - See detailed algorithm description in design section above

    return false
}

// backgroundCleanup runs periodically to remove unused topics and clean up state.
func (tm *TopicManager) backgroundCleanup() {
    for {
        select {
        case <-tm.cleanupTicker.C:
            tm.performCleanup()
        case <-tm.stopChan:
            return
        }
    }
}

func (tm *TopicManager) performCleanup() {
    // TODO 1: Find inactive topics candidates
    //   - Scan tm.topics for topics with no recent activity
    //   - Check LastActivity timestamp against InactivityThreshold
    //   - Topics with no subscribers and no recent messages are candidates

    // TODO 2: Verify topics are safe to delete
    //   - Double-check no active subscribers (including wildcard matches)
    //   - Ensure no pending messages in delivery queues
    //   - Check if topic has persistence requirements
```

```
    // TODO 3: Perform cleanup operations
    //   - Remove topics from tm.topics map
    //   - Clean up any associated persistent storage
    //   - Update metrics and log cleanup operations
    //   - Be careful with locking order to avoid deadlocks
}

func (tm *TopicManager) Stop() {
    if tm.cleanupTicker != nil {
        tm.cleanupTicker.Stop()
    }
    close(tm.stopChan)
}
```

Milestone Checkpoint

After implementing the Topic Manager for Milestone 1, verify the functionality with these concrete tests:

Command to Run: `go test ./internal/topic/... -v`

Expected Test Behavior:

- Topic creation and deletion operations complete without errors
- Message publishing to non-existent topics either auto-creates them or returns appropriate errors based on configuration
- Subscription operations correctly add consumers to topic subscriber lists
- Wildcard pattern matching correctly identifies topics (test with patterns like `sensors/*` and `sensors/#`)
- Concurrent publish and subscribe operations don't cause data races (run with `-race` flag)

Manual Verification Steps:

1. Start the broker with topic manager enabled
2. Use a test client to subscribe to pattern `test/*`
3. Publish messages to `test/temperature` and `test/humidity`
4. Verify subscriber receives both messages
5. Publish to `test/room/sensor1` and verify it's NOT received (single-level wildcard)
6. Subscribe to `test/#` and verify it receives all messages including `test/room/sensor1`

Signs Something is Wrong:

- **Race conditions**: Run tests with `go test -race` - any race conditions indicate locking problems
- **Memory leaks**: If topic count grows without bound during testing, cleanup logic isn't working
- **Missing messages**: If subscribers don't receive expected messages, fanout algorithm has bugs
- **Wildcard failures**: If wildcard patterns don't match expected topics, pattern matching logic is incorrect

Performance Expectations: The topic manager should handle 1000+ topics and 100+ concurrent subscribers without significant latency increase. Message publishing should complete in under 1ms for simple fanout scenarios.

Consumer Groups Component

> **Milestone(s):** Milestone 2 (Consumer Groups & Acknowledgment) — implements consumer groups with round-robin distribution, acknowledgment tracking, and automatic rebalancing when consumers join or leave groups

Mental Model: Work Distribution Teams

Think of consumer groups as specialized work teams in a busy restaurant kitchen. When orders come in, they need to be distributed among the available line cooks so that each order is prepared exactly once, but multiple cooks can work simultaneously to handle high volume. This is fundamentally different from the radio station broadcasting model we saw with topics.

In a restaurant, when a new cook joins the team, the head chef reassigns stations to balance the workload. If a cook leaves during service, their unfinished orders must be quickly redistributed to other team members. The chef maintains a list of who's working on what, tracks when orders are completed, and ensures nothing falls through the cracks. If an order sits too long without being marked complete, the chef assumes something went wrong and reassigns it.

Consumer groups implement this same coordination pattern for message processing. Within a consumer group, each message is delivered to exactly one consumer (point-to-point delivery), but the group can have multiple consumers working in parallel. When consumers join or leave the group, the system automatically rebalances work assignments. When a consumer fails to acknowledge a message within the timeout window, the message is automatically reassigned to another group member.

This differs from the pub/sub pattern where every subscriber receives a copy of each message. Consumer groups enable **horizontal scaling** of message processing — you can add more consumers to handle increased load, and the system automatically distributes the work among them.

Group Coordinator Interface

The Group Coordinator manages consumer group membership, message assignment, and rebalancing operations. It acts as the "head chef" in our restaurant analogy, maintaining awareness of all group members and coordinating work distribution.

Method Name	Parameters	Returns	Description
`JoinGroup`	groupName string, consumerID string, conn *Connection	error	Adds consumer to group and triggers rebalancing if needed
`LeaveGroup`	groupName string, consumerID string	error	Removes consumer from group and redistributes assigned messages
`AssignMessage`	msg *Message, groupName string	(*Consumer, error)	Selects next consumer for message delivery using assignment strategy
`GetGroupState`	groupName string	(*ConsumerGroup, error)	Returns current group membership and assignment state
`TriggerRebalance`	groupName string, reason string	error	Forces immediate rebalancing due to membership changes
`CheckHeartbeats`	groupName string	[]string	Returns list of consumers that missed recent heartbeats
`ProcessHeartbeat`	groupName string, consumerID string	error	Updates consumer's last heartbeat timestamp
`GetAssignmentStrategy`	groupName string	string	Returns current assignment strategy for the group
`SetAssignmentStrategy`	groupName string, strategy string	error	Changes assignment strategy and triggers rebalancing

The Group Coordinator maintains state about each consumer group, tracking membership, assignment strategies, and rebalancing progress. It coordinates with the Topic Manager to receive messages and with the Acknowledgment Tracker to handle delivery confirmations.

> **Design Principle: Centralized Coordination**
>
> The Group Coordinator uses a centralized approach where one component manages all group state and assignment decisions. While this creates a single point of coordination, it simplifies consistency guarantees and makes rebalancing algorithms easier to implement correctly. Distributed coordination approaches (like Kafka's consumer-side coordination) are more complex but can scale to larger group sizes.

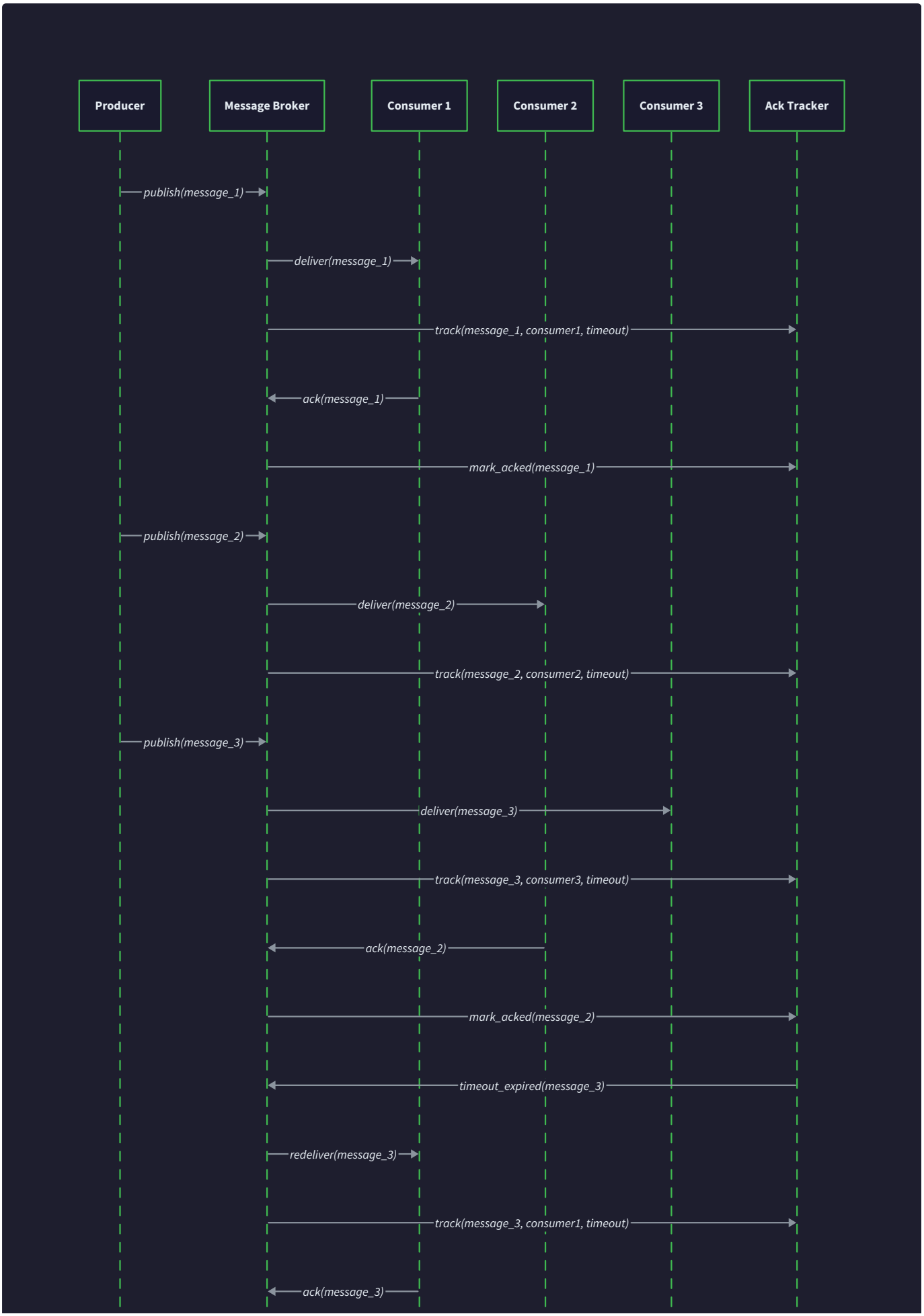
Rebalancing Algorithm

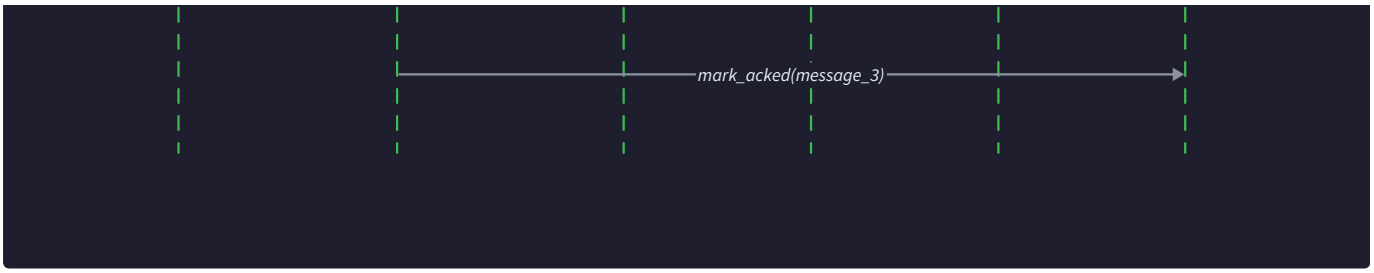
Consumer group rebalancing redistributes message assignments when group membership changes. This ensures work remains evenly distributed and that leaving consumers don't cause message loss. The rebalancing process must handle both voluntary departures (graceful shutdown) and involuntary departures (network failures, crashes).

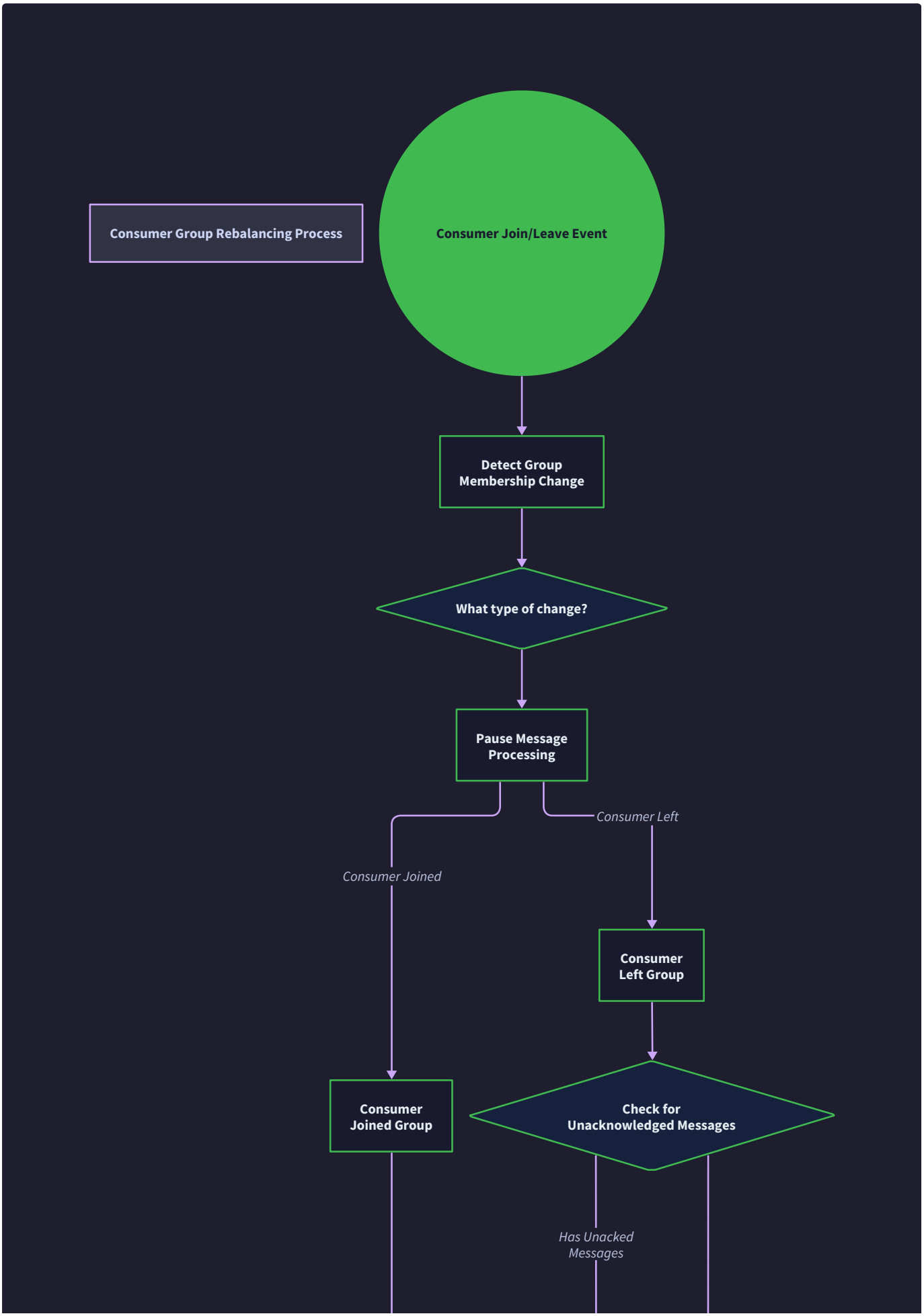
The rebalancing algorithm follows these detailed steps:

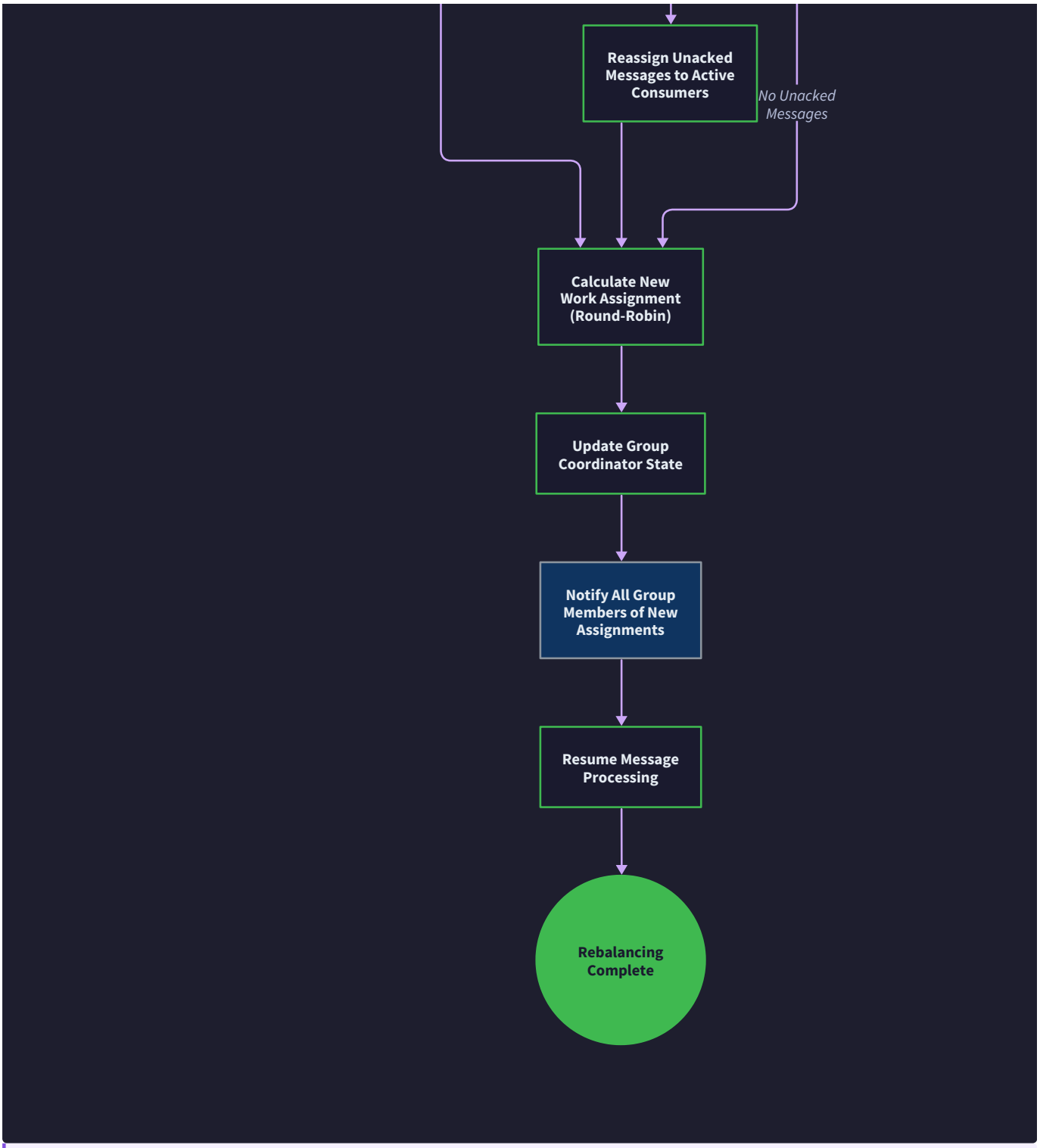
1. **Trigger Detection**: The coordinator detects a rebalancing trigger (new consumer joining, existing consumer leaving, or heartbeat timeout). It immediately sets the `RebalanceInProgress` flag to true and increments the `RebalanceGeneration` counter.
2. **Membership Snapshot**: The coordinator creates a snapshot of current group membership by iterating through the `Members` map in the `ConsumerGroup` struct. It excludes any consumers that have missed more than three consecutive heartbeats.
3. **Assignment Pause**: The coordinator temporarily pauses new message assignments to the group by checking the `RebalanceInProgress` flag in the `AssignMessage` method. Messages continue to be enqueued but are not delivered until rebalancing completes.
4. **Inflight Message Collection**: The coordinator scans each consumer's `PendingMessages` map to identify messages that are currently assigned but not yet acknowledged. These messages must be reassigned during rebalancing to prevent loss.
5. **Assignment Strategy Execution**: Based on the group's `AssignmentStrategy` field (either "round-robin" or "sticky"), the coordinator calculates new message assignments. Round-robin distributes messages evenly across all consumers, while sticky assignment tries to minimize reassignment of inflight messages.
6. **Assignment Distribution**: The coordinator updates the `MessageAssignments` map with the new consumer-to-message mappings. It sends assignment notifications to each consumer over their connection, including the new rebalance generation number.
7. **Consumer Acknowledgment**: Each consumer must acknowledge the rebalancing by responding with the correct generation number. Consumers that fail to acknowledge within 30 seconds are removed from the group and their assignments redistributed.
8. **Completion**: Once all consumers acknowledge, the coordinator clears the `RebalanceInProgress` flag, updates the `LastRebalance` timestamp, and increments the `TotalRebalances` counter. Normal message assignment resumes.

Current State	Trigger Event	Next State	Actions Taken
Stable	Consumer joins	Rebalancing	Set flag, increment generation, pause assignments
Stable	Consumer leaves	Rebalancing	Remove member, collect inflight messages, start rebalancing
Stable	Heartbeat timeout	Rebalancing	Mark consumer dead, trigger membership change
Rebalancing	All consumers ack	Stable	Clear flag, resume assignments, update timestamps
Rebalancing	Consumer timeout	Rebalancing	Remove unresponsive consumer, recalculate assignments
Rebalancing	New join request	Rebalancing	Queue join until current rebalance completes


Producer
Message Broker
Consumer 1
Consumer 2
Consumer 3
Ack Tracker
publish(message_1)
deliver(message_1)
track(message_1, consumer1, timeout)
ack(message_1)
mark_acked(message_1)
publish(message_2)
deliver(message_2)
track(message_2, consumer2, timeout)
publish(message_3)
deliver(message_3)
track(message_3, consumer3, timeout)
ack(message_2)
mark_acked(message_2)
timeout_expired(message_3)
redeliver(message_3)
track(message_3, consumer1, timeout)
ack(message_3)


mark_acked(message_3)


Consumer Group Rebalancing Process
Consumer Join/Leave Event
Detect Group Membership Change
What type of change?
Pause Message Processing
Consumer Left
Consumer Joined
Consumer Left Group
Consumer Joined Group
Check for Unacknowledged Messages
Has Unacked Messages



Critical Insight: The Generation Number

The rebalance generation number is crucial for preventing split-brain scenarios. When a consumer receives a message assignment, it includes the generation number. If the consumer tries to acknowledge a message with an outdated generation, the coordinator rejects it. This ensures that consumers operating with stale assignment information cannot interfere with the new assignment state.

Assignment Strategies

The consumer group coordinator supports two primary assignment strategies that determine how messages are distributed among group members. The choice of strategy significantly impacts performance characteristics, especially during rebalancing operations.

Round-Robin Assignment

Round-robin assignment distributes messages sequentially across all available consumers in the group. The coordinator maintains a `NextAssignment` counter that increments with each message assignment, wrapping around when it reaches the group size.

Round-Robin Algorithm:

1. The coordinator receives a message for assignment to a consumer group
2. It calculates the target consumer index using `NextAssignment % len(group.Members)`
3. It retrieves the consumer at that index from the members list (sorted by consumer ID for consistency)
4. It increments `NextAssignment` for the next message
5. If the selected consumer has reached its `MaxInflight` limit, the coordinator tries the next consumer in sequence

Round-Robin Trade-offs:

Aspect	Pros	Cons
Load Distribution	Perfect balance when consumers have equal processing speed	Imbalanced when consumers have different capabilities
Rebalancing Impact	Simple to recalculate assignments	All assignments change, causing maximum disruption
Implementation	Straightforward counter-based logic	Requires sorted member list for consistency
Message Ordering	Preserves relative ordering across consumers	No ordering guarantees within individual consumers

Sticky Assignment

Sticky assignment attempts to minimize reassignment during rebalancing by keeping existing consumer-message pairs intact whenever possible. This strategy maintains a persistent mapping in the `MessageAssignments` map that survives rebalancing operations.

Sticky Assignment Algorithm:

1. During normal operation, messages are assigned using round-robin but the assignment is recorded in `MessageAssignments`
2. When rebalancing begins, the coordinator first attempts to reassign inflight messages to their current consumers
3. If a consumer has left the group, only their messages are redistributed to remaining consumers
4. New messages are assigned to consumers with the lowest current load (fewest inflight messages)
5. The assignment history is preserved to inform future rebalancing decisions

Sticky Assignment Trade-offs:

Aspect	Pros	Cons
Rebalancing Efficiency	Minimizes message reassignment and duplicate processing	More complex algorithm with edge cases
Consumer Affinity	Maintains consumer-message relationships for caching benefits	Can create hotspots with uneven message processing
Memory Usage	Requires persistent assignment tracking	Higher memory overhead for assignment state
Fault Tolerance	Faster recovery when consumers rejoin	Complex cleanup when assignments become stale

Strategy Selection Considerations

The choice between round-robin and sticky assignment depends on specific application requirements and operational characteristics:

Choose Round-Robin When:

- Consumers are homogeneous (same processing capability)
- Message processing is stateless
- Simple operational model is preferred
- Rebalancing frequency is low

Choose Sticky Assignment When:

- Consumers maintain message-specific state or caches
- Message processing costs vary significantly
- Rebalancing happens frequently
- Minimizing duplicate processing is critical

> **Decision: Default Assignment Strategy**
>
> - **Context**: Consumer groups need a default assignment strategy that works well for most use cases while being simple to understand and debug
> - **Options Considered**:
>   - Round-robin as default with sticky as advanced option
>   - Sticky as default with round-robin as simple option
>   - Configurable per-group with no system default
> - **Decision**: Round-robin as system default with per-group configuration support
> - **Rationale**: Round-robin provides predictable behavior and perfect load balance for the common case of homogeneous consumers. It's easier to reason about during debugging and has fewer edge cases. Sticky assignment can be enabled for specific groups that need it.
> - **Consequences**: Simple default behavior for new users, with the option to optimize for specific workloads. Documentation must clearly explain when to choose each strategy.

Consumer Group Architecture Decisions

Assignment Strategy Storage

Decision: Per-Group Assignment Strategy Configuration

- **Context**: Different consumer groups may have different performance characteristics and requirements that favor different assignment strategies
- **Options Considered**:
  - Global system-wide assignment strategy setting
  - Per-topic assignment strategy configuration
  - Per-group assignment strategy with inheritance from topic defaults
- **Decision**: Per-group assignment strategy stored in `ConsumerGroup.AssignmentStrategy` field
- **Rationale**: Groups processing the same topic may have different requirements (e.g., one group does fast stateless processing while another maintains expensive per-message state). Per-group configuration provides maximum flexibility without adding complexity to topic management.
- **Consequences**: Enables optimization for specific workloads but requires administrators to understand the trade-offs. Default strategy must be carefully chosen for new groups.

Configuration Level	Pros	Cons	Chosen?
Global system-wide	Simple administration, consistent behavior	Cannot optimize for different workloads	No
Per-topic	Balances simplicity with customization	Groups on same topic may have different needs	No
Per-group	Maximum flexibility for optimization	More complex configuration management	**Yes**

Rebalancing Trigger Conditions

Decision: Immediate Rebalancing on Membership Changes

- **Context**: Consumer group membership can change due to planned scaling, unplanned failures, or network partitions. The timing of rebalancing affects both performance and consistency.
- **Options Considered**:
  - Immediate rebalancing on any membership change
  - Batched rebalancing with configurable delay to handle flapping
  - Lazy rebalancing only when assignment conflicts occur
- **Decision**: Immediate rebalancing with exponential backoff for repeated membership changes
- **Rationale**: Immediate rebalancing ensures consistent work distribution and prevents message loss when consumers fail. Exponential backoff prevents thrashing during network instability while still maintaining responsiveness for legitimate scaling events.
- **Consequences**: Enables fast failover and scaling but may cause performance spikes during network instability. Requires careful tuning of backoff parameters.

Trigger Strategy	Pros	Cons	Chosen?
Immediate	Fast failover, consistent assignments	May thrash during instability	**Yes** (with backoff)
Batched/delayed	Reduces thrashing, more stable	Slower failover, delayed scaling	No
Lazy/on-conflict	Minimal overhead	Inconsistent assignments, complex conflict resolution	No

Consumer Heartbeat Mechanism

Decision: Coordinator-Initiated Heartbeat Tracking

- **Context**: Consumer group coordination requires distinguishing between healthy consumers and those that have failed or become unresponsive. The heartbeat mechanism affects both failure detection speed and system overhead.
- **Options Considered**:
  - Consumer-initiated heartbeats sent periodically to coordinator
  - Coordinator-initiated health checks polling all consumers
  - Piggyback heartbeats on message acknowledgments
- **Decision**: Consumer-initiated heartbeats with coordinator tracking `LastHeartbeat` timestamps
- **Rationale**: Consumer-initiated heartbeats provide the best balance of timely failure detection and low overhead. Consumers already maintain connections to the coordinator, so heartbeat messages use existing infrastructure. Piggyback approaches fail when consumers have no messages to acknowledge.
- **Consequences**: Requires consumers to implement heartbeat logic but provides reliable failure detection. Heartbeat frequency must be tuned to balance detection speed with network overhead.

Heartbeat Approach	Pros	Cons	Chosen?
Consumer-initiated	Low overhead, timely detection	Requires consumer implementation	**Yes**
Coordinator-polling	Simple consumer logic	Higher network overhead	No
Piggyback on ACKs	Zero additional overhead	Fails with idle consumers	No

Common Consumer Group Pitfalls

⚠️ Pitfall: Duplicate Delivery During Rebalancing

During consumer group rebalancing, messages that are inflight (delivered but not yet acknowledged) can be delivered to multiple consumers if the rebalancing algorithm is not carefully implemented. This happens when the coordinator reassigns an inflight message to a new consumer before the original consumer has had a chance to acknowledge it.

Why This Happens: The window between message delivery and acknowledgment creates a race condition during rebalancing. If Consumer A receives Message M1 at time T1, and a rebalancing begins at time T2 before the acknowledgment arrives, the coordinator may reassign M1 to Consumer B. If Consumer A's acknowledgment arrives after the reassignment, both consumers end up processing the same message.

How to Detect: Monitor for duplicate message IDs being acknowledged by different consumers within the same consumer group. Log entries showing the same message being assigned to multiple consumers during rebalancing periods indicate this issue.

How to Fix: Implement generation-based message tracking where each message assignment includes the current rebalance generation number. Reject acknowledgments that reference outdated generations. During rebalancing, wait for a grace period to allow inflight acknowledgments to arrive before reassigning messages, or implement proper lease expiration to ensure exclusivity.

⚠️ Pitfall: Thundering Herd on Rebalancing

When a consumer group undergoes rebalancing, especially with large numbers of consumers, all group members may simultaneously attempt to rejoin or reestablish their assignments. This can overwhelm the coordinator and cause cascading failures or extremely long rebalancing times.

Why This Happens: Rebalancing notifications are typically broadcast to all group members simultaneously. If consumers implement aggressive retry logic, they can overwhelm the coordinator with concurrent requests. This is especially problematic when the rebalancing was triggered by coordinator overload in the first place.

How to Detect: Monitor coordinator CPU and memory usage during rebalancing operations. Look for connection timeouts, rejected requests, or rebalancing operations that take significantly longer than expected. Multiple rapid-fire rebalancing operations may indicate the coordinator is struggling to handle the load.

How to Fix: Implement randomized backoff and jittered retry timing for consumer responses to rebalancing notifications. Add per-consumer rate limiting on rebalancing participation. Consider implementing a coordinator queue system that processes rebalancing requests serially rather than concurrently, or partition large consumer groups into smaller sub-groups.

⚠️ Pitfall: Inconsistent Assignment Generation Tracking

When consumers or the coordinator fail to properly track rebalance generation numbers, split-brain scenarios can occur where different consumers operate with incompatible views of message assignments. This leads to either duplicate processing or message loss.

Why This Happens: Generation numbers must be consistently updated across all group members and the coordinator during every rebalancing operation. If a consumer misses a rebalancing notification due to network issues or if the coordinator fails to update generation numbers atomically, parts of the system operate with stale state.

How to Detect: Log and monitor generation number mismatches in acknowledgment rejection events. Look for consumers that repeatedly have their acknowledgments rejected due to outdated generations, or messages that never get acknowledged because they're assigned to consumers with mismatched generations.

How to Fix: Implement persistent generation tracking in the coordinator that survives restarts. Require consumers to include generation numbers in all operations (not just acknowledgments). Add generation validation to all coordinator operations and reject requests with incorrect generations. Implement a coordinator recovery mechanism that can reconstruct current generation state from the persistent log.

⚠️ Pitfall: Memory Leaks from Dead Consumer State

Consumer groups can accumulate state for consumers that have permanently left but were not properly cleaned up. This includes entries in the `Members` map, orphaned `PendingMessages`, and stale `MessageAssignments` that consume memory indefinitely.

Why This Happens: When consumers crash or experience network partitions, they may not send proper leave notifications to the coordinator. If heartbeat timeout detection is not properly implemented or has overly generous timeouts, dead consumer state persists indefinitely. Assignment maps may also retain references to messages that were reassigned but not properly cleaned up.

How to Detect: Monitor memory usage trends in the coordinator, especially the size of consumer group maps. Look for consumer group members that have not sent heartbeats for extended periods but are still present in membership lists. Check for messages in assignment maps that reference consumers no longer in the members list.

How to Fix: Implement aggressive heartbeat timeout detection with automatic cleanup of dead consumers. Add background cleanup tasks that scan for orphaned state (assignments referencing non-existent consumers, pending messages for consumers not in groups). Use weak references or periodic garbage collection for consumer state, and implement proper cleanup in all consumer removal code paths.

⚠️ Pitfall: Head-of-Line Blocking in Round-Robin Assignment

In round-robin assignment, when one consumer becomes slow or unresponsive, it can block message processing for the entire group if the assignment algorithm doesn't account for consumer capacity and response times.

Why This Happens: Pure round-robin assignment assigns messages to consumers based solely on position in the rotation, without considering whether the target consumer has capacity to handle more messages. If Consumer A is processing slowly and reaches its `MaxInflight` limit, the assignment algorithm may continue trying to assign messages to it, causing messages to queue up rather than being distributed to available consumers.

How to Detect: Monitor per-consumer inflight message counts and processing times. Look for situations where some consumers in a group are idle while others are at maximum capacity. Check for increasing message queue depths despite having available consumers in the group.

How to Fix: Modify the round-robin assignment algorithm to skip consumers that have reached their `MaxInflight` limit and select the next available consumer in the rotation. Implement capacity-aware assignment that considers both consumer position and current load. Add load balancing logic that can temporarily deviate from strict round-robin when consumers have significantly different processing rates.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Consumer Tracking	In-memory Go maps with sync.RWMutex	Distributed consensus with etcd/Consul
Heartbeat Transport	TCP keepalive + custom messages	gRPC health checking service
Assignment Storage	JSON serialization to disk	Protocol Buffers with WAL integration
Rebalancing Coordination	Single-threaded state machine	Multi-phase distributed consensus
Membership Management	Timeout-based failure detection	Gossip protocol with phi accrual

Recommended File Structure

```
internal/coordinator/
  coordinator.go           ← Main GroupCoordinator implementation
  coordinator_test.go      ← Unit tests for core logic
  assignment_strategies.go ← Round-robin and sticky assignment implementations
  rebalancing.go           ← Rebalancing state machine and algorithms
  heartbeat.go             ← Consumer heartbeat tracking and cleanup
  membership.go            ← Consumer group membership management
  types.go                 ← ConsumerGroup, Consumer, and related types
  errors.go                ← Consumer group specific error types
```

Infrastructure Starter Code

Complete heartbeat tracking infrastructure that handles the non-core complexity:

```go
// Package coordinator provides consumer group coordination and rebalancing
package coordinator

import (
    "sync"
    "time"
    "context"
)

// HeartbeatTracker manages consumer liveness detection with automatic cleanup
type HeartbeatTracker struct {
    mu              sync.RWMutex
    consumers       map[string]*ConsumerHeartbeat
    heartbeatTTL    time.Duration
    cleanupInterval time.Duration
    stopCh          chan struct{}
}

type ConsumerHeartbeat struct {
    ConsumerID    string
    GroupName     string
    LastHeartbeat time.Time
    MissedCount   int
}

// NewHeartbeatTracker creates a tracker with background cleanup
func NewHeartbeatTracker(heartbeatTTL, cleanupInterval time.Duration) *HeartbeatTracker {
    tracker := &HeartbeatTracker{
        consumers:       make(map[string]*ConsumerHeartbeat),
        heartbeatTTL:    heartbeatTTL,
        cleanupInterval: cleanupInterval,
        stopCh:          make(chan struct{}),
    }

    go tracker.backgroundCleanup()
    return tracker
}
```

```go
// ProcessHeartbeat updates the last heartbeat time for a consumer
func (h *HeartbeatTracker) ProcessHeartbeat(groupName, consumerID string) {
    h.mu.Lock()
    defer h.mu.Unlock()

    key := groupName + ":" + consumerID
    if hb, exists := h.consumers[key]; exists {
        hb.LastHeartbeat = time.Now()
        hb.MissedCount = 0
    } else {
        h.consumers[key] = &ConsumerHeartbeat{
            ConsumerID:    consumerID,
            GroupName:     groupName,
            LastHeartbeat: time.Now(),
            MissedCount:   0,
        }
    }
}

// GetDeadConsumers returns consumers that have exceeded the heartbeat TTL
func (h *HeartbeatTracker) GetDeadConsumers(groupName string) []string {
    h.mu.RLock()
    defer h.mu.RUnlock()

    var deadConsumers []string
    now := time.Now()

    for key, hb := range h.consumers {
        if hb.GroupName == groupName {
            if now.Sub(hb.LastHeartbeat) > h.heartbeatTTL {
                deadConsumers = append(deadConsumers, hb.ConsumerID)
            }
        }
    }

    return deadConsumers
```

```go
}

// RemoveConsumer removes tracking for a consumer
func (h *HeartbeatTracker) RemoveConsumer(groupName, consumerID string) {
    h.mu.Lock()
    defer h.mu.Unlock()
    delete(h.consumers, groupName+":"+consumerID)
}

// backgroundCleanup periodically removes stale consumer heartbeat records
func (h *HeartbeatTracker) backgroundCleanup() {
    ticker := time.NewTicker(h.cleanupInterval)
    defer ticker.Stop()

    for {
        select {
        case <-ticker.C:
            h.cleanup()
        case <-h.stopCh:
            return
        }
    }
}

func (h *HeartbeatTracker) cleanup() {
    h.mu.Lock()
    defer h.mu.Unlock()

    now := time.Now()
    for key, hb := range h.consumers {
        // Remove consumers that have been dead for more than 2x heartbeat TTL
        if now.Sub(hb.LastHeartbeat) > h.heartbeatTTL*2 {
            delete(h.consumers, key)
        }
    }
}
```

```
// Stop shuts down the background cleanup goroutine

func (h *HeartbeatTracker) Stop() {

    close(h.stopCh)

}
```

Assignment strategy infrastructure with pluggable implementations:

```go
// AssignmentStrategy defines how messages are distributed among group members
type AssignmentStrategy interface {
    // AssignMessage selects a consumer for the given message
    AssignMessage(msg *Message, group *ConsumerGroup) (*Consumer, error)

    // OnRebalance recalculates assignments when membership changes
    OnRebalance(group *ConsumerGroup, newMembers []*Consumer) error

    // Name returns the strategy identifier
    Name() string
}

// StrategyRegistry manages available assignment strategies
type StrategyRegistry struct {
    strategies map[string]AssignmentStrategy
}

func NewStrategyRegistry() *StrategyRegistry {
    registry := &StrategyRegistry{
        strategies: make(map[string]AssignmentStrategy),
    }

    // Register built-in strategies
    registry.Register(NewRoundRobinStrategy())
    registry.Register(NewStickyStrategy())

    return registry
}

func (r *StrategyRegistry) Register(strategy AssignmentStrategy) {
    r.strategies[strategy.Name()] = strategy
}

func (r *StrategyRegistry) Get(name string) (AssignmentStrategy, bool) {
    strategy, exists := r.strategies[name]
    return strategy, exists
}
```

Core Logic Skeleton

Main coordinator implementation with detailed TODOs:

```go
// GroupCoordinator manages consumer groups, membership, and message assignment
type GroupCoordinator struct {
    mu                sync.RWMutex
    groups            map[string]*ConsumerGroup
    heartbeatTracker  *HeartbeatTracker
    strategyRegistry  *StrategyRegistry
    rebalanceTimeout  time.Duration
    maxRebalanceRetries int
}

// JoinGroup adds a consumer to the specified group and triggers rebalancing
func (gc *GroupCoordinator) JoinGroup(groupName, consumerID string, conn *Connection) error {
    // TODO 1: Acquire write lock to ensure consistent group state during modification
    // TODO 2: Check if group exists, create new ConsumerGroup if not (set CreatedAt timestamp)
    // TODO 3: Check if consumer is already in group - if so, update connection and return success
    // TODO 4: Create new Consumer struct with ID, connection, current timestamp for LastHeartbeat
    // TODO 5: Add consumer to group.Members map using consumerID as key
    // TODO 6: If this is the first member, set group.AssignmentStrategy to default ("round-robin")
    // TODO 7: Call TriggerRebalance with reason "consumer-joined" to redistribute work
    // TODO 8: Update group.LastRebalance timestamp and increment TotalRebalances counter
    // Hint: Use defer to unlock mutex even if errors occur
    // Hint: Initialize Consumer with MaxInflight=10 as default, empty PendingMessages map
}

// TriggerRebalance initiates the rebalancing process for a consumer group
func (gc *GroupCoordinator) TriggerRebalance(groupName, reason string) error {
    // TODO 1: Acquire write lock and locate the target group in groups map
    // TODO 2: Check if rebalancing is already in progress - if so, return error to prevent concurrent rebalancing
    // TODO 3: Set group.RebalanceInProgress = true and increment group.RebalanceGeneration
    // TODO 4: Create snapshot of current group members by copying group.Members map
    // TODO 5: Remove any consumers that failed heartbeat check using heartbeatTracker.GetDeadConsumers()
    // TODO 6: Collect inflight messages from all remaining consumers' PendingMessages maps
    // TODO 7: Get the assignment strategy from strategyRegistry using group.AssignmentStrategy
    // TODO 8: Call strategy.OnRebalance() to calculate new message assignments
    // TODO 9: Send rebalance notifications to all consumers with new generation number
    // TODO 10: Start timeout goroutine to handle consumers that don't acknowledge rebalancing
    // Hint: Use context.WithTimeout for rebalance timeout handling
```

```
    // Hint: If no consumers remain after heartbeat check, clean up the group
}

// AssignMessage selects a consumer for message delivery using the group's assignment strategy
func (gc *GroupCoordinator) AssignMessage(msg *Message, groupName string) (*Consumer, error) {
    // TODO 1: Acquire read lock for concurrent-safe access to group state
    // TODO 2: Look up the consumer group in groups map, return error if not found
    // TODO 3: Check if rebalancing is in progress - if so, queue message and return nil consumer
    // TODO 4: Get the assignment strategy from strategyRegistry using group.AssignmentStrategy
    // TODO 5: Call strategy.AssignMessage() to select target consumer
    // TODO 6: Verify selected consumer hasn't exceeded MaxInflight limit
    // TODO 7: Add message to consumer's PendingMessages map with current timestamp
    // TODO 8: Update message.AssignedConsumer field and set message.AckDeadline
    // TODO 9: Increment consumer.CurrentInflight counter
    // TODO 10: Return the selected consumer for message delivery
    // Hint: AckDeadline should be current time + 30 seconds
    // Hint: If strategy returns consumer at MaxInflight, try next consumer in round-robin order
}
```

Round-robin assignment strategy implementation:

```go
// RoundRobinStrategy implements simple round-robin message assignment
type RoundRobinStrategy struct{}

func NewRoundRobinStrategy() *RoundRobinStrategy {
    return &RoundRobinStrategy{}
}

func (r *RoundRobinStrategy) Name() string {
    return "round-robin"
}

// AssignMessage selects the next consumer in round-robin order
func (r *RoundRobinStrategy) AssignMessage(msg *Message, group *ConsumerGroup) (*Consumer, error) {
    // TODO 1: Check if group has any members - return error if empty
    // TODO 2: Create sorted slice of consumer IDs from group.Members for consistent ordering
    // TODO 3: Calculate target index using group.NextAssignment % len(members)
    // TODO 4: Get consumer at target index, check if it has capacity (CurrentInflight < MaxInflight)
    // TODO 5: If consumer has capacity, increment group.NextAssignment and return consumer
    // TODO 6: If consumer at capacity, try next consumer in sequence (up to len(members) attempts)
    // TODO 7: If all consumers are at capacity, return error indicating group overload
    // Hint: Sort consumer IDs alphabetically to ensure consistent assignment order
    // Hint: Use modulo arithmetic to wrap around when reaching end of consumer list
}

// OnRebalance recalculates round-robin position when membership changes
func (r *RoundRobinStrategy) OnRebalance(group *ConsumerGroup, newMembers []*Consumer) error {
    // TODO 1: Update group.Members map by clearing old members and adding new members
    // TODO 2: Redistribute inflight messages from group.MessageAssignments to new members
    // TODO 3: Reset group.NextAssignment to 0 to restart round-robin from first member
    // TODO 4: Clear any assignments for consumers no longer in the group
    // TODO 5: Notify all new members of their assigned messages with current generation number
    // Hint: During rebalancing, messages should be redistributed as evenly as possible
    // Hint: Preserve message ordering within each consumer's assignment
}
```

Language-Specific Hints

For Go implementation:

- Use `sync.RWMutex` for consumer group state to allow concurrent reads during normal operation
- Use `time.NewTicker()` for heartbeat checking with proper cleanup via `ticker.Stop()`

- Use `context.WithTimeout()` for rebalancing operations to prevent infinite blocking
- Use `sort.Strings()` on consumer IDs to ensure consistent assignment ordering
- Use `atomic` package for counters like `TotalRebalances` to avoid race conditions
- Implement proper connection cleanup with `defer conn.Close()` patterns

Milestone Checkpoint

After implementing the consumer group coordinator:

Test Command:

```bash
go test ./internal/coordinator/... -v -race
```

Expected Behavior:

- Creating a consumer group and joining consumers should trigger rebalancing
- Round-robin message assignment should distribute messages evenly across group members
- Consumer failures (simulated by stopping heartbeats) should trigger automatic rebalancing
- Message acknowledgments should properly remove messages from pending queues

Manual Verification:

1. Start the message broker server
2. Create a topic: `curl -X POST localhost:8080/topics/test-topic`
3. Join multiple consumers to the same group: `curl -X POST localhost:8080/groups/test-group/join -d '{"consumer_id": "consumer1"}'`
4. Publish messages and verify they're distributed round-robin: `curl -X POST localhost:8080/topics/test-topic/publish -d '{"message": "test"}'`
5. Stop one consumer and verify rebalancing redistributes its pending messages

Signs of Problems:

- Messages delivered to multiple consumers in the same group (duplicate processing)
- Messages never delivered despite available consumers (assignment failures)
- Rebalancing operations that never complete (deadlock or timeout issues)
- Memory usage growing over time (consumer state leaks)

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Messages delivered to multiple consumers	Race condition during rebalancing	Check logs for overlapping assignments during rebalancing	Implement proper generation number validation
Rebalancing never completes	Consumer not acknowledging rebalance notification	Monitor consumer heartbeats and rebalance timeouts	Add timeout handling and dead consumer removal
Uneven message distribution	Round-robin counter not updating correctly	Log NextAssignment value and consumer selection	Fix modulo arithmetic and counter synchronization
Memory usage growing constantly	Dead consumer state not cleaned up	Monitor size of group.Members map over time	Implement heartbeat timeout and background cleanup
Consumer group stuck in rebalancing state	Exception during assignment calculation	Check assignment strategy error handling	Add proper error recovery and state reset
Messages lost during rebalancing	Inflight messages not properly redistributed	Trace message assignments during rebalancing	Ensure all pending messages are collected and reassigned

Message Acknowledgment Component

> **Milestone(s):** Milestone 2 (Consumer Groups & Acknowledgment) — implements message acknowledgment, tracking, and redelivery mechanisms that enable reliable message processing and automatic failure recovery

Mental Model: Certified Mail Receipts

Think of message acknowledgment like the postal service's certified mail system. When you send an important document via certified mail, several things happen: the postal service tracks the package at each step, requires the recipient to sign for delivery, and provides you with a receipt proving successful delivery. If the recipient isn't home, they leave a notice and attempt redelivery later. If multiple attempts fail, the package goes to a special handling facility.

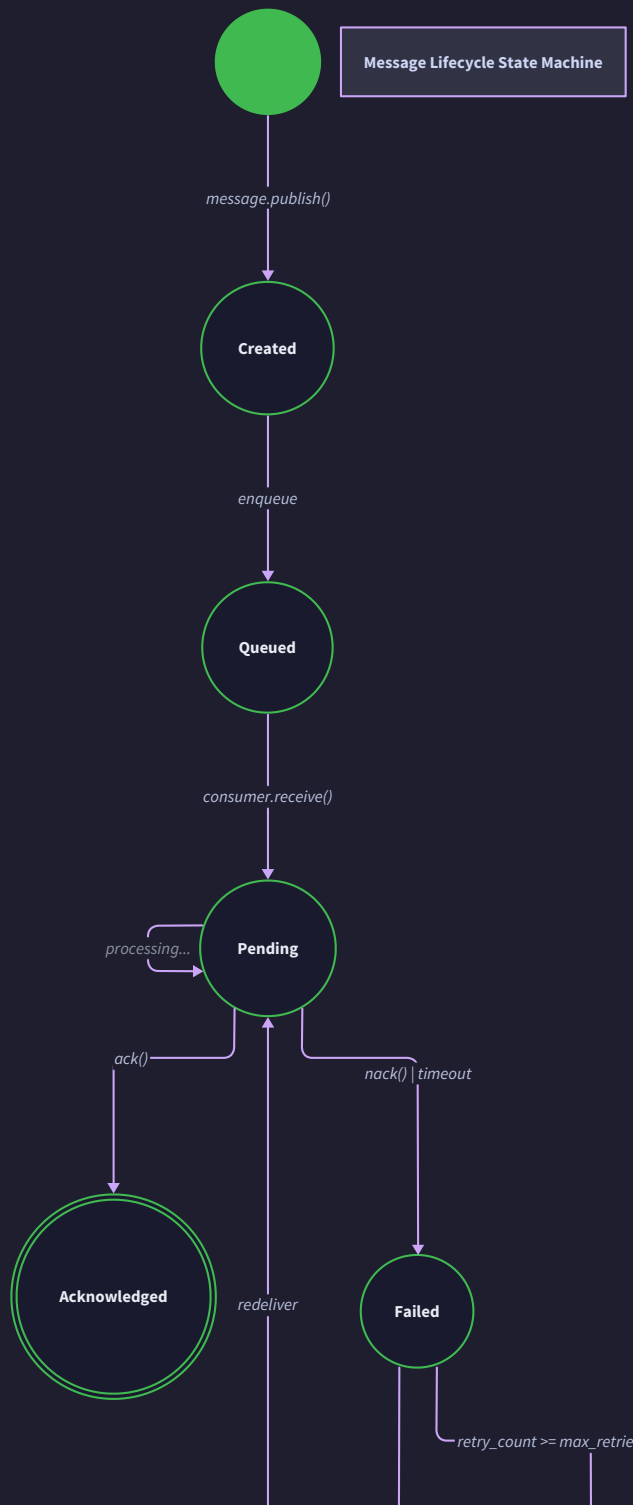
Message acknowledgment works similarly in our message queue. When a message is delivered to a consumer (like handing over the certified mail), the broker starts tracking it and expects a "signature" back - the ACK. If no ACK arrives within a reasonable time (like if the recipient never signs), the broker assumes something went wrong and attempts redelivery to another consumer. Messages that repeatedly fail delivery eventually get sent to a "special handling facility" - the dead letter queue.

This mental model helps us understand why acknowledgment tracking is crucial: without it, we can't distinguish between "message was processed successfully" and "consumer crashed halfway through processing." Just like certified mail provides delivery guarantees that regular mail cannot, message acknowledgment provides delivery guarantees that simple fire-and-forget messaging cannot.

The **Acknowledgment Tracker** is the component that manages this entire process. It maintains a registry of all unacknowledged messages, monitors timeout deadlines, handles positive and negative acknowledgments, and coordinates redelivery attempts. This component bridges the gap between delivering a message and confirming its successful processing.

Message States

- **Created**: Message instantiated by producer
- **Queued**: Message stored in topic/queue
- **Pending**: Message delivered, awaiting acknowledgment
- **Acknowledged**: Message successfully processed
- **Failed**: Processing failed or timed out
- **Redelivery**: Message queued for retry
- **Dead Letter**: Maximum retries exceeded



Acknowledgment Tracker Interface

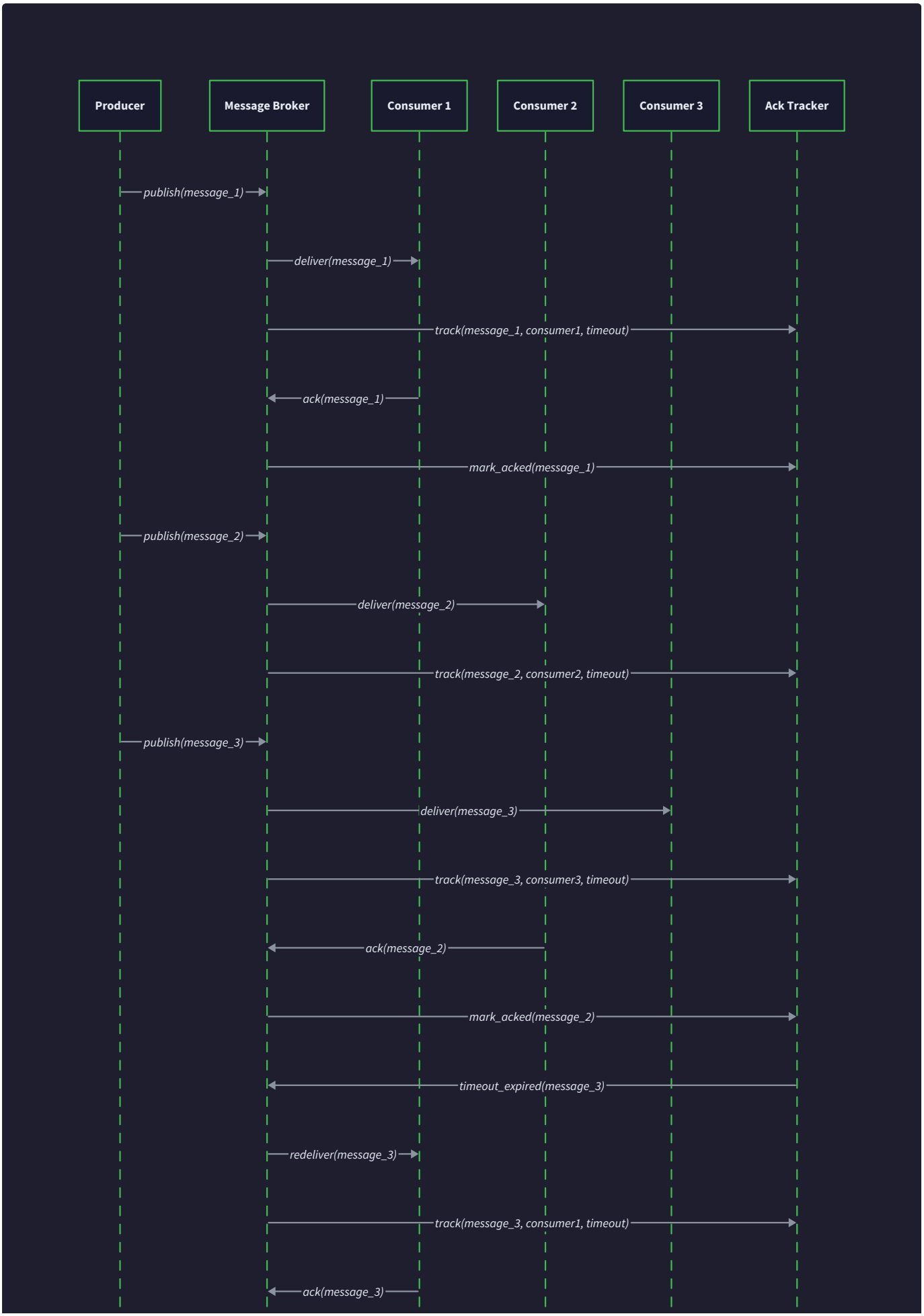
The Acknowledgment Tracker provides a clean interface for managing the lifecycle of messages after they've been delivered to consumers. This component sits between the Consumer Group Coordinator (which assigns messages) and the Topic Manager (which handles redelivery fanout).

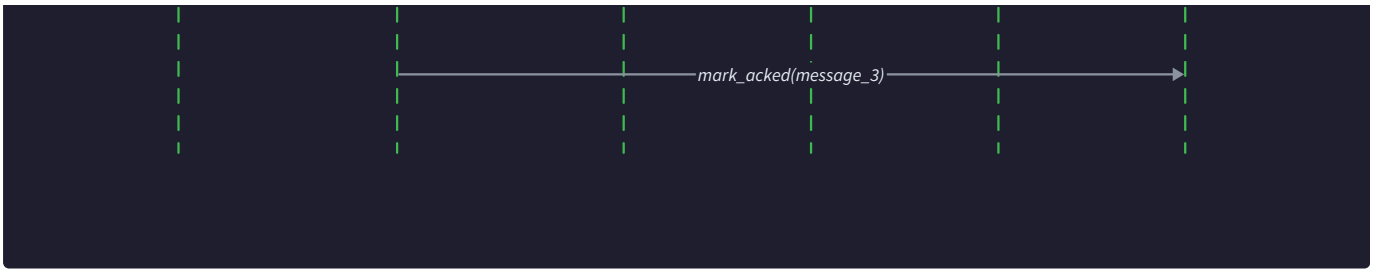
Method Name	Parameters	Returns	Description
`TrackMessage`	`msg *Message, consumerID string, deadline time.Time`	`error`	Registers a message as pending acknowledgment from specific consumer with timeout deadline
`ProcessAck`	`messageID string, consumerID string`	`error`	Records successful acknowledgment and removes message from pending tracking
`ProcessNack`	`messageID string, consumerID string, reason string`	`error`	Records negative acknowledgment and triggers immediate redelivery to different consumer
`CheckTimeouts`	`currentTime time.Time`	`[]Message`	Returns messages that exceeded acknowledgment deadline and need redelivery
`GetPendingCount`	`consumerID string`	`int`	Returns number of unacknowledged messages for specific consumer (for backpressure)
`GetConsumerLoad`	`consumerID string`	`ConsumerLoadInfo`	Returns detailed load information including inflight count, oldest pending, average processing time
`CleanupConsumer`	`consumerID string`	`[]Message`	Removes consumer and returns all its pending messages for reassignment
`UpdateDeliveryStatus`	`messageID string, status DeliveryStatus`	`error`	Updates internal tracking status without triggering state transitions
`GetMessageHistory`	`messageID string`	`[]DeliveryAttempt`	Returns delivery attempt history for debugging and poison message detection

The interface design emphasizes **temporal safety** - every tracked message has an explicit deadline, and the system regularly scans for expired messages rather than relying on callback timers. This approach handles clock skew gracefully and makes timeout behavior predictable under load.

The `TrackMessage` operation immediately transitions a message from `StateAssigned` to being actively monitored. The tracker stores the assignment relationship and starts monitoring the acknowledgment deadline. This design ensures that every delivered message has exactly one consumer responsible for acknowledging it.

The `ProcessAck` and `ProcessNack` operations handle the two possible consumer responses. ACK indicates successful processing and removes the message from all tracking structures. NACK indicates processing failure and triggers immediate redelivery, but unlike timeout-based redelivery, NACK provides explicit error information that can influence routing decisions.


Producer
Message Broker
Consumer 1
Consumer 2
Consumer 3
Ack Tracker
publish(message_1)
deliver(message_1)
track(message_1, consumer1, timeout)
ack(message_1)
mark_acked(message_1)
publish(message_2)
deliver(message_2)
track(message_2, consumer2, timeout)
publish(message_3)
deliver(message_3)
track(message_3, consumer3, timeout)
ack(message_2)
mark_acked(message_2)
timeout_expired(message_3)
redeliver(message_3)
track(message_3, consumer1, timeout)
ack(message_3)



Redelivery Algorithm

The redelivery algorithm handles the complex process of reassigning messages when consumers fail to acknowledge them within the specified deadline. This algorithm must handle concurrent operations safely while maintaining message ordering guarantees within consumer groups.

The redelivery process follows these steps:

1. **Timeout Detection**: The `CheckTimeouts` method scans all pending messages and identifies those exceeding their acknowledgment deadline. This scan occurs periodically (typically every 1-5 seconds) rather than using individual timers for efficiency.
2. **Consumer Liveness Check**: Before redelivering a message, verify that the original assigned consumer is still connected and participating in the consumer group. If the consumer disconnected, mark all its pending messages for immediate redelivery.
3. **Retry Count Increment**: Increment the message's `RetryCount` field and check against `MaxRetries`. If the limit is exceeded, route the message to the dead letter queue instead of attempting redelivery.
4. **Assignment Strategy Consultation**: Ask the consumer group's assignment strategy to select a new consumer for the message. The strategy should avoid reassigning to the consumer that just failed (unless it's the only available consumer).
5. **State Transition Validation**: Ensure the message is still in a valid state for redelivery. Messages might have been acknowledged by the original consumer while the timeout scan was running.
6. **Redelivery Execution**: Update the message's assignment information, reset the acknowledgment deadline, and deliver to the new consumer through the normal delivery pipeline.
7. **Tracking Update**: Update internal tracking structures to reflect the new assignment while maintaining delivery attempt history for debugging.

The algorithm handles several edge cases carefully. **Concurrent acknowledgments** can arrive while redelivery is in progress - the implementation uses message state to ensure idempotent processing. **Consumer group rebalancing** during redelivery requires coordination with the Group Coordinator to avoid assigning messages to consumers that are leaving the group.

Current Message State	Timeout Event	Next State	Action Taken
`StateAssigned`	Acknowledgment deadline exceeded	`StatePending`	Increment retry count, find new consumer, redeliver
`StateAssigned`	Consumer disconnected	`StatePending`	Mark for immediate redelivery regardless of deadline
`StateAssigned`	Max retries exceeded	`StateDeadLetter`	Route to dead letter queue, notify monitoring
`StatePending`	Timeout during redelivery	No change	Log warning, continue with assignment process
`StateAcknowledged`	Timeout (race condition)	No change	Ignore timeout, message already processed

Redelivery scheduling uses an exponential backoff strategy to avoid overwhelming struggling consumers. The first retry happens immediately, subsequent retries introduce delays: 1 second, 4 seconds, 16 seconds, up to a maximum of 5 minutes. This gives temporary issues time to resolve while ensuring timely processing of legitimate failures.

> The critical insight in redelivery design is distinguishing between "consumer is slow" and "consumer has failed." A slow consumer will eventually acknowledge the message, while a failed consumer never will. The acknowledgment tracker uses both timeout thresholds and heartbeat monitoring to make this distinction accurately.

Poison Message Detection

Poison messages are messages that consistently cause consumer processing failures, leading to infinite redelivery loops that can degrade system performance. The acknowledgment tracker implements sophisticated poison message detection that goes beyond simple retry counting.

The detection system tracks multiple failure indicators:

Retry Count Tracking: Each message maintains a `RetryCount` that increments on every redelivery attempt. When `RetryCount` exceeds the configured `MaxRetries` threshold (typically 3-5), the message is automatically routed to the dead letter queue.

Consumer Failure Patterns: The tracker maintains statistics on which consumers repeatedly NACK or timeout on specific message types. If a message causes failures across multiple different consumers (indicating the problem is with the message content, not a specific consumer), it gets flagged as potentially poisonous.

Temporal Failure Analysis: Messages that exhibit rapid retry cycles (multiple failures within a short time window) get escalated faster than messages with sporadic failures. A message that fails 3 times in 30 seconds is more likely poisonous than one that fails 3 times over 3 hours.

Content-Based Detection: The tracker can optionally analyze message headers or payload characteristics to identify patterns in poisonous messages. For example, messages with malformed JSON payloads or missing required headers can be detected and routed to the DLQ immediately.

The poison message detection algorithm works as follows:

1. **On Each Failure**: Record the failure type (timeout vs NACK), consumer ID, timestamp, and any error details provided by the consumer.
2. **Pattern Analysis**: Check if this message has failed on multiple different consumers within a sliding time window. Multiple consumers failing suggests message-level issues rather than consumer-specific problems.
3. **Escalation Threshold Calculation**: Instead of using a fixed retry limit, calculate a dynamic threshold based on failure velocity. Fast-failing messages get shorter retry limits.
4. **Dead Letter Routing Decision**: When routing to DLQ, include comprehensive metadata about failure history, participating consumers, and detected patterns to aid in later analysis.
5. **Consumer Protection**: If a consumer repeatedly fails on many different messages, flag it as potentially problematic and reduce its message assignment priority until it demonstrates successful processing.

Failure Pattern	Detection Criteria	Action Taken	DLQ Metadata
Consistent timeout across consumers	>2 different consumers timeout within 5 minutes	Route to DLQ after 2 retries instead of 5	Consumer IDs, timeout durations, message size
Immediate NACK with same error	Same NACK reason from >1 consumer	Route to DLQ after 1 retry	NACK reasons, consumer versions, error codes
Rapid retry cycling	3+ failures within 30 seconds	Route to DLQ immediately	Failure timestamps, assignment history
Large message timeouts	Message >1MB consistently times out	Route to DLQ, suggest content inspection	Message size, consumer memory limits
Consumer-specific failures	Same consumer fails >50% of assigned messages	Reduce consumer priority, continue normal retry	Consumer health metrics, recent failure rate

> The key insight in poison message detection is that legitimate processing failures show different patterns than poison messages. A database outage causes widespread failures across many message types, while a poison message causes specific, repeatable failures regardless of system health.

Acknowledgment Architecture Decisions

The design of the acknowledgment system involves several critical decisions that significantly impact system reliability, performance, and operational complexity. Each decision represents a trade-off between different system properties.

Decision: Acknowledgment Timeout Strategy

- **Context**: Messages delivered to consumers need timeout handling in case consumers crash or become unresponsive. The system must choose between fixed timeouts, adaptive timeouts, or consumer-negotiated timeouts.
- **Options Considered**: Fixed global timeout, per-topic timeout configuration, adaptive timeout based on historical processing times, consumer-negotiated timeout per message
- **Decision**: Per-consumer-group configurable timeout with adaptive adjustment based on recent processing history
- **Rationale**: Different message types have vastly different processing requirements. Financial transactions might complete in milliseconds, while image processing might take minutes. Consumer groups typically handle homogeneous workloads, so group-level configuration provides appropriate granularity. Adaptive adjustment handles load variations without manual tuning.
- **Consequences**: Enables optimal timeout values per workload type, reduces false timeouts under load, requires more complex timeout management logic, needs historical processing time tracking

Option	Pros	Cons	Performance Impact
Fixed global timeout	Simple implementation, predictable behavior	Too short for complex workloads, too long for fast workloads	Low CPU overhead
Per-topic timeout	Better workload matching	Requires configuration management, static values	Low CPU overhead
Adaptive timeout	Self-tuning, handles load variation	Complex algorithm, potential oscillation	Medium CPU overhead
Consumer-negotiated	Perfect per-message optimization	Complex protocol, potential abuse	Medium-high overhead

Decision: Delivery Semantics Guarantee

- **Context**: The system must choose between at-most-once, at-least-once, or exactly-once delivery semantics. This choice affects acknowledgment design, storage requirements, and performance characteristics.
- **Options Considered**: At-most-once (fire-and-forget), at-least-once (acknowledgment required), exactly-once (deduplication required)
- **Decision**: At-least-once delivery with optional consumer-side idempotency support
- **Rationale**: Exactly-once delivery requires distributed transaction coordination or expensive deduplication, significantly complicating the implementation. At-most-once provides no reliability guarantees. At-least-once offers good reliability with reasonable complexity, and consumers can implement idempotency for their specific use cases.
- **Consequences**: Guarantees no message loss under normal operation, allows duplicate delivery during failures, requires consumers to handle duplicate messages, enables high throughput with simple implementation

Decision: Negative Acknowledgment Handling

- **Context**: When consumers detect they cannot process a message successfully, they need a way to indicate explicit failure. The system must decide how to handle these negative acknowledgments.
- **Options Considered**: No NACK support (timeout only), immediate redelivery on NACK, delayed redelivery on NACK, NACK with reason codes
- **Decision**: Immediate redelivery with reason code tracking and escalated poison message detection
- **Rationale**: NACK provides explicit failure signal that's faster and more reliable than waiting for timeout. Immediate redelivery enables fast recovery from transient errors. Reason codes enable sophisticated poison message detection and debugging.
- **Consequences**: Faster failure recovery, reduced message latency during transient errors, requires more complex consumer implementation, needs reason code standardization

Delivery Semantic	Reliability	Performance	Implementation Complexity	Consumer Requirements
At-most-once	Low (message loss possible)	High (no ACK overhead)	Low	Simple processing
At-least-once	High (no message loss)	Medium (ACK overhead)	Medium	Idempotent processing
Exactly-once	High (no loss/duplicates)	Low (dedup overhead)	High	Simple processing

Decision: Acknowledgment Persistence Strategy

- **Context**: The system must decide whether acknowledgments need to be persisted to disk for crash recovery or can remain in memory for better performance.
- **Options Considered**: No persistence (memory only), synchronous acknowledgment logging, asynchronous acknowledgment logging, batch acknowledgment persistence
- **Decision**: Asynchronous batch acknowledgment logging with configurable flush interval
- **Rationale**: Synchronous logging creates significant latency for acknowledgment operations. Memory-only acknowledgments risk redelivering many messages after broker restart. Asynchronous batching provides good durability with acceptable performance impact.
- **Consequences**: Enables crash recovery with minimal duplicate delivery, provides good acknowledgment throughput, adds complexity to WAL integration, requires tuning of flush intervals

Common Acknowledgment Pitfalls

Understanding and avoiding common mistakes in acknowledgment system implementation is crucial for building a reliable message queue. These pitfalls represent patterns that consistently cause problems in real-world systems.

⚠️ Pitfall: Infinite Redelivery Loops

The most dangerous pitfall occurs when poison messages enter infinite redelivery cycles, consuming broker resources and preventing processing of valid messages. This happens when the retry logic doesn't properly detect or handle messages that consistently fail processing.

Why it's wrong: A single malformed message can cause cascading failures by consuming consumer resources, filling logs with error messages, and delaying processing of subsequent valid messages. In extreme cases, infinite redelivery can cause memory exhaustion as pending message tracking structures grow unbounded.

How to fix: Implement strict retry limits with exponential backoff and comprehensive poison message detection. Set the maximum retry count to a low value (3-5) and implement dead letter queue routing. Track failure patterns across multiple consumers to identify message-level issues rather than consumer-specific problems. Include circuit breaker logic that temporarily pauses problematic consumers.

```
Example problematic pattern:
Message A fails → retry → fails → retry → fails → retry (forever)

Better pattern:
Message A fails → retry with 1s delay → fails → retry with 4s delay → fails → route to DLQ
```

⚠️ Pitfall: Head-of-Line Blocking

Head-of-line blocking occurs when a slow or failing message prevents processing of subsequent messages in the same consumer's queue. This is especially problematic in ordered message processing scenarios.

Why it's wrong: A single problematic message can halt processing for an entire consumer, reducing effective throughput and increasing overall processing latency. The impact multiplies in consumer groups where one blocked consumer reduces the group's effective processing capacity.

How to fix: Implement timeout-based message skipping or parallel processing lanes. When a message exceeds processing time thresholds, temporarily set it aside and continue processing subsequent messages. Return to problematic messages during low-traffic periods or route them to specialized handling consumers.

⚠️ Pitfall: Race Conditions Between ACK and Timeout

Concurrent acknowledgment and timeout processing can lead to race conditions where a message is acknowledged by the original consumer while simultaneously being redelivered to another consumer, resulting in duplicate processing.

Why it's wrong: Duplicate processing can cause data consistency issues, especially in financial or inventory management systems where operations must occur exactly once. Race conditions also complicate debugging and monitoring since the system state becomes unpredictable.

How to fix: Use atomic state transitions with proper locking or compare-and-swap operations. Implement message state machines that prevent invalid transitions (e.g., cannot acknowledge a message that's already been marked as timed out). Include generation numbers or version stamps in acknowledgments to detect stale operations.

⚠️ Pitfall: Inadequate Consumer Failure Detection

Systems that rely solely on timeout-based failure detection often fail to distinguish between slow consumers and failed consumers, leading to premature redelivery of messages that are still being processed.

Why it's wrong: Premature redelivery causes unnecessary duplicate processing and can overwhelm the system during high-load periods when processing naturally slows down. It also wastes broker resources tracking multiple assignments for the same message.

How to fix: Implement heartbeat-based consumer health monitoring in addition to acknowledgment timeouts. Allow consumers to send progress updates for long-running processing. Use adaptive timeout calculation based on historical processing times and current system load. Provide consumer-controlled acknowledgment deadline extension for legitimately long-running operations.

⚠️ Pitfall: Memory Leaks in Pending Message Tracking

Tracking structures for pending messages can grow unbounded if not properly cleaned up, especially when consumers disconnect without properly acknowledging their assigned messages.

Why it's wrong: Memory leaks eventually lead to broker crashes under sustained load. Large tracking structures also degrade performance of timeout scanning and acknowledgment lookup operations.

How to fix: Implement comprehensive cleanup on consumer disconnection. Use weak references or time-based expiration for tracking structures. Regularly scan for and clean up orphaned tracking entries. Set maximum limits on pending messages per consumer with backpressure when limits are approached.

Pitfall	Symptoms	Detection Method	Prevention Strategy
Infinite redelivery	High CPU, growing DLQ, repeated error logs	Monitor retry count distribution	Strict retry limits, poison detection
Head-of-line blocking	Increasing consumer lag, timeout spikes	Track per-message processing time	Parallel processing lanes, timeout skipping
ACK/timeout races	Duplicate processing alerts, state inconsistencies	Audit duplicate deliveries	Atomic state transitions, generation numbers
Poor failure detection	Unnecessary redelivery, high timeout rates	Monitor false positive timeouts	Heartbeat monitoring, adaptive timeouts
Memory leaks	Growing broker memory, slow operations	Monitor tracking structure sizes	Connection cleanup, bounded tracking

Implementation Guidance

The Acknowledgment Tracker bridges the gap between message delivery and processing confirmation, requiring careful coordination between timing systems, state management, and failure detection. This component must handle high-frequency operations efficiently while maintaining strict consistency guarantees.

Technology Recommendations:

Component	Simple Option	Advanced Option
State Storage	In-memory map with RWMutex	Concurrent hash map with lock striping
Timeout Management	Periodic scanning with time.Ticker	Hierarchical timing wheels
Persistence	JSON append to log file	Binary WAL with checksums
Failure Detection	Simple retry counters	Machine learning failure pattern detection
Monitoring	Basic metrics collection	Prometheus metrics with alerting

Recommended File Structure:

```
internal/acknowledgment/
  tracker.go              ← main AckTracker implementation
  tracker_test.go         ← comprehensive unit tests
  delivery_attempt.go     ← delivery attempt tracking
  poison_detector.go      ← poison message detection logic
  timeout_scanner.go      ← timeout detection background worker
  metrics.go             ← acknowledgment-related metrics
  interfaces.go          ← interface definitions
internal/storage/
  wal_integration.go     ← WAL persistence for acknowledgments
internal/monitoring/
  ack_metrics.go         ← acknowledgment monitoring helpers
```

Infrastructure Starter Code:

```go
package acknowledgment

import (
    "context"
    "sync"
    "time"
    "encoding/json"
    "log"
)

// DeliveryAttempt tracks individual delivery attempts for debugging and poison detection
type DeliveryAttempt struct {
    AttemptID     string            `json:"attempt_id"`
    ConsumerID    string            `json:"consumer_id"`
    DeliveredAt   time.Time         `json:"delivered_at"`
    AckDeadline   time.Time         `json:"ack_deadline"`
    CompletedAt   *time.Time        `json:"completed_at,omitempty"`
    Result        DeliveryResult    `json:"result"`
    ErrorReason   string            `json:"error_reason,omitempty"`
    ProcessingDuration time.Duration `json:"processing_duration"`
}

type DeliveryResult int

const (
    DeliveryPending DeliveryResult = iota
    DeliveryAcked
    DeliveryNacked
    DeliveryTimedOut
    DeliveryReassigned
)

// ConsumerLoadInfo provides detailed consumer performance information
type ConsumerLoadInfo struct {
    ConsumerID        string        `json:"consumer_id"`
    PendingCount      int           `json:"pending_count"`
    MaxInflight       int           `json:"max_inflight"`
    OldestPending     *time.Time    `json:"oldest_pending,omitempty"`
```

```
    AvgProcessingTime time.Duration `json:"avg_processing_time"`
    RecentErrorRate   float64       `json:"recent_error_rate"`
    LastHeartbeat     time.Time     `json:"last_heartbeat"`
}

// PendingMessage tracks message delivery state and timing
type PendingMessage struct {
    Message         *Message          `json:"message"`
    ConsumerID      string            `json:"consumer_id"`
    AssignedAt      time.Time         `json:"assigned_at"`
    AckDeadline     time.Time         `json:"ack_deadline"`
    DeliveryHistory []DeliveryAttempt `json:"delivery_history"`
    RetryCount      int               `json:"retry_count"`
    LastRetryAt     *time.Time        `json:"last_retry_at,omitempty"`
}

// AckTracker manages message acknowledgment lifecycle and failure detection
type AckTracker struct {
    pendingMessages   map[string]*PendingMessage  // messageID -> pending tracking
    consumerMessages  map[string]map[string]bool  // consumerID -> messageIDs
    timeoutQueue      []*PendingMessage           // sorted by ack deadline

    ackTimeoutDefault time.Duration
    maxRetries        int
    poisonDetector    *PoisonDetector
    walLogger         WALLogger

    mutex           sync.RWMutex
    timeoutScanner  *time.Ticker
    stopChan        chan struct{}

    // Metrics tracking
    totalTracked     int64
    totalAcked       int64
    totalNacked      int64
    totalTimedOut    int64
```

```
    totalRedelivered int64
}

// WALLogger interface for persistence integration
type WALLogger interface {
    LogAckEvent(eventType string, data interface{}) error
    ReplayAckEvents() ([]AckEvent, error)
}

type AckEvent struct {
    EventType   string          `json:"event_type"`
    Timestamp   time.Time       `json:"timestamp"`
    MessageID   string          `json:"message_id"`
    ConsumerID  string          `json:"consumer_id"`
    Data        json.RawMessage `json:"data"`
}
```

Core Logic Skeleton:

```go
// TrackMessage registers a message for acknowledgment tracking with the specified consumer and deadline.
// This transitions the message into active monitoring and starts the acknowledgment timeout.
func (at *AckTracker) TrackMessage(msg *Message, consumerID string, deadline time.Time) error {
    // TODO 1: Validate input parameters (non-nil message, valid consumer ID, future deadline)
    // TODO 2: Check if message is already being tracked (prevent duplicate tracking)
    // TODO 3: Create PendingMessage structure with initial delivery attempt
    // TODO 4: Acquire write lock and add to tracking structures (pendingMessages, consumerMessages)
    // TODO 5: Insert into timeout queue maintaining deadline ordering
    // TODO 6: Log tracking event to WAL for crash recovery
    // TODO 7: Update metrics (totalTracked counter)
    // Hint: Use time.Now() for AssignedAt, generate unique AttemptID
    // Hint: Consider using heap data structure for efficient timeout queue management
}

// ProcessAck handles successful message acknowledgment from a consumer.
// This completes the message lifecycle and removes all tracking state.
func (at *AckTracker) ProcessAck(messageID string, consumerID string) error {
    // TODO 1: Validate that message exists in pending tracking
    // TODO 2: Verify that specified consumer is currently assigned this message
    // TODO 3: Calculate processing duration from assignment to acknowledgment
    // TODO 4: Update delivery attempt with completion information
    // TODO 5: Remove message from all tracking structures (atomic operation)
    // TODO 6: Log acknowledgment event to WAL
    // TODO 7: Update consumer load statistics and metrics
    // TODO 8: Check for consumer performance patterns (unusually fast/slow processing)
    // Hint: Use defer to ensure cleanup even if logging fails
    // Hint: Consider updating consumer reputation scoring based on ACK patterns
}

// ProcessNack handles negative acknowledgment indicating processing failure.
// This triggers immediate redelivery to a different consumer if available.
func (at *AckTracker) ProcessNack(messageID string, consumerID string, reason string) error {
    // TODO 1: Validate message exists and is assigned to specified consumer
    // TODO 2: Record NACK in delivery attempt history with reason and timestamp
    // TODO 3: Increment message retry count and check against poison detection thresholds
    // TODO 4: Determine if message should be redelivered or routed to DLQ
    // TODO 5: If redelivering, select new consumer using assignment strategy
```

```
    // TODO 6: Update message assignment and reset acknowledgment deadline
    // TODO 7: Log NACK event and new assignment to WAL
    // TODO 8: Update poison detection patterns and consumer error rates
    // Hint: NACK should trigger faster redelivery than timeout (immediate vs delayed)
    // Hint: Track NACK reason patterns to identify poison message types
}

// CheckTimeouts scans for messages exceeding acknowledgment deadlines and triggers redelivery.
// This method runs periodically to handle consumer failures and processing delays.
func (at *AckTracker) CheckTimeouts(currentTime time.Time) []Message {
    // TODO 1: Scan timeout queue for messages past their acknowledgment deadline
    // TODO 2: For each timed-out message, verify consumer is still connected and healthy
    // TODO 3: Update delivery attempt with timeout result and processing duration
    // TODO 4: Check retry count against maximum and route to DLQ if exceeded
    // TODO 5: For retriable messages, increment retry count and find new consumer
    // TODO 6: Calculate new acknowledgment deadline with exponential backoff
    // TODO 7: Update tracking structures with new assignment information
    // TODO 8: Return list of successfully reassigned messages for delivery
    // TODO 9: Log all timeout and reassignment events to WAL
    // TODO 10: Update timeout metrics and consumer reliability scores
    // Hint: Process timeouts in batches for efficiency under high load
    // Hint: Use exponential backoff: 1s, 4s, 16s, 64s, max 5m between retries
}

// GetConsumerLoad returns comprehensive load information for the specified consumer.
// This enables backpressure decisions and consumer health monitoring.
func (at *AckTracker) GetConsumerLoad(consumerID string) ConsumerLoadInfo {
    // TODO 1: Calculate pending message count for this consumer
    // TODO 2: Find oldest unacknowledged message timestamp
    // TODO 3: Calculate average processing time from recent delivery attempts
    // TODO 4: Compute recent error rate (NACKs and timeouts vs total attempts)
    // TODO 5: Get last heartbeat timestamp from consumer tracking
    // TODO 6: Return structured load information for decision making
    // Hint: Use sliding window (last 100 messages or 5 minutes) for averages
    // Hint: Consider consumer capacity vs current load for utilization metrics
}
```

Milestone Checkpoint:

After implementing the Acknowledgment Tracker, verify the following behavior:

1. **Basic Tracking**: Start broker, publish message to consumer group, verify message appears in pending tracking, send ACK, confirm message is removed from tracking.
2. **Timeout Handling**: Deliver message to consumer, disconnect consumer without ACK, wait for timeout period, verify message is reassigned to another consumer.
3. **NACK Processing**: Send NACK with reason from consumer, verify immediate redelivery to different consumer, check that NACK reason is logged.
4. **Poison Detection**: Send message that consistently causes NACK from multiple consumers, verify it eventually routes to dead letter queue.
5. **Load Monitoring**: Query consumer load API during high-volume processing, verify metrics accurately reflect pending counts and processing times.

Expected log output:

```
INFO: Tracking message msg-123 for consumer consumer-A, deadline 2024-01-15T10:05:30Z
INFO: Acknowledged message msg-123 by consumer consumer-A, processing_duration=2.3s
WARN: Message msg-456 timed out from consumer consumer-B after 30s, retry_count=1
ERROR: Message msg-789 exceeded max retries (5), routing to dead letter queue
```

Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Messages never acknowledged	Consumer processing errors not sending ACK	Check consumer logs for exceptions, verify ACK protocol implementation	Add consumer error handling and explicit ACK calls
Frequent timeout redelivery	ACK timeout too short for workload	Monitor processing time distribution, check system load	Increase ACK timeout or implement adaptive timeout
Memory usage growing	Pending message cleanup not working	Monitor pending message count over time, check consumer disconnect handling	Add periodic cleanup scan and connection event handlers
Duplicate processing	Race between ACK and timeout	Check for concurrent ACK/timeout log entries for same message	Implement atomic state transitions with generation numbers
Poor redelivery performance	Inefficient timeout scanning	Profile timeout check performance, monitor scan duration	Use heap data structure for timeout queue, limit scan batch size

Persistence Layer

> **Milestone(s):** Milestone 3 (Persistence & Backpressure) — implements message persistence using an append-only log file and crash recovery mechanisms that enable data durability and broker restart recovery

Mental Model: Laboratory Notebook

Think of the persistence layer as a **scientific laboratory notebook** — the kind that researchers use to record every experiment, observation, and result in chronological order. In a laboratory, scientists never erase or modify previous entries; they only add new pages. Each entry is dated, includes complete context, and describes exactly what happened. If a researcher leaves the lab and returns months later, they can read through their notebook and understand precisely where they left off.

Similarly, our message broker's **Write-Ahead Log (WAL)** functions as this laboratory notebook. Every significant event — message publications, consumer acknowledgments, group membership changes — gets recorded as a permanent entry before any actual processing occurs. The entries are immutable and ordered by time. If the broker crashes and restarts, it reads through its "notebook" (the WAL) and reconstructs exactly what state it was in before the failure.

The laboratory notebook analogy extends to **log compaction** as well. Periodically, researchers might create summary reports that consolidate months of individual experiments into key findings, allowing them to archive old notebook volumes while preserving essential information. The persistence layer similarly compacts old log entries, removing superseded records while maintaining the information needed for crash recovery.

This mental model helps explain why the persistence layer uses an **append-only design** rather than updating records in-place. Just as scientists maintain research integrity by never altering past observations, the message broker maintains data integrity by never modifying committed log entries. This design choice enables both crash recovery and audit trails, at the cost of storage space that must be managed through retention policies.

Write-Ahead Log Interface

The **Write-Ahead Log (WAL)** serves as the persistence layer's core abstraction, providing durable storage for all broker state changes. The WAL interface defines three primary operations that enable the broker to record events before they occur, read historical records during recovery, and create periodic snapshots for efficient startup.

The WAL operates on the principle of **temporal safety** — every state-changing operation must be recorded in the log before the broker acknowledges success to clients. This ensures that even if the broker crashes immediately after sending a response, the committed operation can be recovered from the persistent log during startup.

Method Name	Parameters	Returns	Description
`AppendRecord`	`recordType WALRecordType, data []byte`	`(offset int64, error)`	Writes a record to the log with atomic durability guarantees, returning the log offset for future reference
`ReadRecord`	`offset int64`	`(*WALRecord, error)`	Retrieves a specific record by its log offset, used during crash recovery and log inspection
`ReadRange`	`startOffset, endOffset int64`	`([]*WALRecord, error)`	Reads multiple consecutive records efficiently, supporting batch recovery operations
`CreateCheckpoint`	`metadata CheckpointMetadata`	`(checkpointID string, error)`	Creates a snapshot of current broker state, enabling faster recovery by avoiding full log replay
`GetLatestCheckpoint`	none	`(*CheckpointMetadata, error)`	Returns metadata about the most recent checkpoint, used to determine recovery starting point
`CompactLog`	`beforeOffset int64`	`error`	Removes log entries older than specified offset, reclaiming storage space after successful checkpointing
`GetLogStats`	none	`(*LogStatistics, error)`	Returns log size, record count, and oldest/newest offsets for monitoring and administrative purposes
`Close`	none	`error`	Performs graceful shutdown, ensuring all pending writes are flushed to disk before terminating

The `WALRecord` structure encapsulates all information needed to describe and verify a log entry:

Field Name	Type	Description
`Magic`	`uint32`	Fixed magic number (0x57414C47) that identifies valid WAL records and enables corruption detection
`Version`	`uint32`	Record format version, enabling backward compatibility when log format evolves
`RecordLength`	`uint32`	Total size of this record in bytes, enabling efficient record boundary detection during sequential reads
`RecordType`	`WALRecordType`	Enumerated type indicating the kind of operation recorded (message publish, acknowledgment, etc.)
`Timestamp`	`int64`	Unix timestamp in nanoseconds when the record was created, enabling time-based log analysis
`TransactionID`	`uint64`	Unique identifier linking related records within a logical transaction or operation
`Checksum`	`uint32`	CRC32 hash of the data payload, enabling detection of storage corruption or incomplete writes
`Data`	`json.RawMessage`	Serialized payload containing operation-specific information, format determined by RecordType

The WAL supports several record types that correspond to different broker operations:

Record Type	Purpose	Data Format
`RecordMessagePublished`	Records new message acceptance	Message ID, topic, payload hash, producer ID, headers
`RecordMessageAcked`	Records successful acknowledgment	Message ID, consumer ID, acknowledgment timestamp
`RecordMessageNacked`	Records negative acknowledgment	Message ID, consumer ID, failure reason, retry count
`RecordConsumerJoined`	Records consumer group membership	Group name, consumer ID, assignment strategy
`RecordConsumerLeft`	Records consumer departure	Group name, consumer ID, departure reason
`RecordTopicCreated`	Records new topic creation	Topic name, retention policy, creation timestamp
`RecordCheckpointCreated`	Records state snapshot	Checkpoint ID, offset range, metadata hash

The WAL implementation provides **atomicity guarantees** through careful write ordering and fsync operations. Each `AppendRecord` call writes the complete record to the log file and forces a disk synchronization before returning success. This ensures that acknowledged operations survive power failures and operating system crashes.

> **Design Insight**: The WAL offset returned by `AppendRecord` serves multiple purposes beyond just record identification. It acts as a logical timestamp that enables consistent ordering across concurrent operations, provides a checkpoint for incremental recovery, and supports efficient log compaction by identifying which records can safely be removed.

Crash Recovery Algorithm

The **crash recovery process** transforms the broker from a crashed state back to a consistent operational state by replaying the Write-Ahead Log. Recovery operates in distinct phases, each building upon the previous phase's restored state until the broker can safely resume normal operations.

The recovery algorithm must handle several challenging scenarios: **partial writes** where the broker crashed while writing a record, **orphaned messages** where a message was logged but never delivered, and **inconsistent state** where the in-memory data structures don't match the persistent log. The algorithm addresses these challenges through careful validation and state reconstruction.

Recovery Phase 1: Log Validation and Repair

1. **Open the WAL file** and verify that it exists and is readable. If the file is missing or corrupted beyond repair, initialize an empty log and begin with a clean broker state.
2. **Scan for the latest valid checkpoint** by reading checkpoint metadata records in reverse chronological order. A valid checkpoint must have a matching state snapshot file and contain a hash that verifies data integrity.
3. **Validate log integrity** from the checkpoint forward by reading each record and verifying its magic number, checksum, and record length. Mark any corrupted records as invalid and determine the last known good offset.

4. **Repair truncated records** by examining partial writes at the end of the log. If the final record has a valid header but incomplete data section, truncate the log at the previous record boundary to maintain consistency.
5. **Create a recovery plan** that identifies which log range requires replay, estimates recovery time based on record count, and determines whether a full or incremental recovery is needed.

Recovery Phase 2: State Reconstruction

6. **Initialize empty broker state** with default topic configurations, empty consumer group registrations, and cleared acknowledgment tracking structures. This provides a clean foundation for state reconstruction.
7. **Load checkpoint state** if a valid checkpoint exists by deserializing topic metadata, consumer group assignments, and pending message tracking information from the checkpoint snapshot file.
8. **Replay WAL records** sequentially from the checkpoint offset to the end of the valid log, applying each operation to rebuild the broker's in-memory state:
   - `RecordMessagePublished`: Add message to topic queue and update message index
   - `RecordMessageAcked`: Remove message from pending acknowledgments
   - `RecordConsumerJoined`: Add consumer to group and trigger rebalancing
   - `RecordTopicCreated`: Initialize topic with specified retention policy
9. **Reconcile consumer connections** by marking all consumers from the checkpoint as disconnected, since network connections don't survive broker restarts. Consumers will reconnect and rejoin their groups automatically.
10. **Validate state consistency** by checking that all referenced topics exist, all consumer group assignments are valid, and all pending messages have corresponding tracking entries.

Recovery Phase 3: Resumption Preparation

11. **Rebuild indexes and caches** including wildcard subscription lookups, consumer group assignment strategies, and topic message ordering. These derived data structures enable efficient runtime operation.
12. **Schedule redelivery** for all messages that were pending acknowledgment at the time of the crash, using the configured acknowledgment timeout values to determine immediate vs. delayed redelivery.
13. **Initialize background tasks** including log compaction scheduling, retention policy enforcement, and consumer heartbeat monitoring. These processes maintain system health during normal operation.
14. **Create recovery checkpoint** documenting the successful recovery process, including log range replayed, state validation results, and any warnings or errors encountered during recovery.
15. **Resume network operations** by reopening the TCP listener, enabling client connections, and beginning normal message processing. The broker is now fully operational again.

The recovery algorithm maintains several **safety invariants** throughout the process:

> **Safety Invariant 1**: No message acknowledged to a producer before the crash will be lost during recovery
>
> **Safety Invariant 2**: No message will be delivered more than once to the same consumer, even across crash boundaries
>
> **Safety Invariant 3**: Consumer group assignments will be recalculated after recovery, but no messages will be delivered to disconnected consumers

The algorithm handles **edge cases** that commonly occur during recovery:

Scenario	Detection	Recovery Action
Partial checkpoint write	Checkpoint file exists but hash doesn't match contents	Ignore corrupted checkpoint, use previous valid checkpoint or full recovery
Message published but never delivered	WAL contains RecordMessagePublished but no consumer assignment	Add message to pending delivery queue for immediate assignment
Consumer acknowledged message during crash	WAL shows message delivered but no acknowledgment record	Treat as unacknowledged, eligible for redelivery after timeout
Topic deleted during crash	Checkpoint shows topic but recent WAL contains deletion	Complete the deletion during recovery, notify any pending consumers
Consumer group rebalancing interrupted	WAL shows rebalancing start but no completion record	Force new rebalancing round with current membership

Log Compaction Strategy

Log compaction addresses the fundamental challenge of **unbounded log growth** in append-only systems. Without compaction, the WAL would grow indefinitely, consuming increasing amounts of disk space and slowing crash recovery as the replay log lengthens. The compaction strategy balances storage efficiency against recovery speed and operational complexity.

The compaction process operates on the principle of **state consolidation** — multiple log records that describe intermediate states of the same entity can be replaced with a single record representing the final state. For example, if a message is published, assigned to a consumer, and then acknowledged, the compaction process can remove all three records since the message no longer requires tracking.

Retention Policy Framework

The broker supports multiple **retention policies** that determine when records become eligible for compaction:

Policy Type	Configuration	Behavior	Use Case
Time-based	`MaxAge duration`	Remove records older than specified duration	Compliance with data retention regulations
Size-based	`MaxSize int64`	Compact when log exceeds specified byte size	Disk space management on storage-constrained systems
Count-based	`MaxRecords int64`	Compact when log exceeds specified record count	Memory usage control for recovery operations
Checkpoint-based	`RetainCheckpoints int`	Remove records before Nth most recent checkpoint	Balance between recovery speed and storage efficiency

The retention policies work in conjunction with **lifecycle-based compaction** that removes records based on the current state of the entities they describe:

1. **Completed transactions** — Records for fully acknowledged messages can be removed since they no longer affect broker state
2. **Obsolete subscriptions** — Records for consumers that have permanently left groups can be compacted away
3. **Superseded configurations** — Only the most recent topic configuration record needs to be retained
4. **Consolidated checkpoints** — Intermediate checkpoints can be removed once newer checkpoints are verified

Compaction Algorithm

The log compaction process operates as a **background task** that runs periodically based on configurable triggers:

1. **Trigger evaluation** checks whether compaction should run based on log size growth, time since last compaction, or explicit administrative request. Multiple triggers can be active simultaneously.
2. **Compaction window determination** identifies the range of log records eligible for compaction, typically from the oldest record up to a recent checkpoint boundary. Records after the compaction window remain untouched.
3. **State analysis** scans the compaction window to build an index of entity final states:

- Messages that are fully acknowledged and no longer pending
- Consumers that have left all groups and disconnected
- Topics that have been deleted and have no remaining messages
- Superseded configuration records

4. **Compaction plan generation** creates a list of records to retain and records to remove, along with size estimates for the compacted log and validation checksums.
5. **Atomic log rewriting** creates a new log file with only the records marked for retention, maintaining exact record formatting and offset relationships for remaining entries.
6. **Checkpoint update** creates a new checkpoint that references the compacted log structure and updates internal offset mappings to reflect the compacted log layout.
7. **Old log cleanup** removes the pre-compaction log files after verifying that the new log structure passes validation and can support normal operations.

The compaction algorithm preserves several **critical properties**:

> **Compaction Invariant 1**: All information necessary for crash recovery must be retained in the compacted log
>
> **Compaction Invariant 2**: Record ordering within the compacted log must match the original temporal ordering
>
> **Compaction Invariant 3**: No currently active operations should be affected by compaction of historical records

Advanced Compaction Strategies

For systems with high message throughput, the broker supports **incremental compaction** that processes small log segments continuously rather than compacting the entire log periodically:

Strategy	Window Size	Frequency	Trade-offs
Full compaction	Entire eligible log	Weekly/monthly	Maximum space reclamation, longest compaction time
Sliding window	Fixed record count (e.g., 100K records)	Daily	Moderate space reclamation, predictable compaction time
Adaptive window	Based on record types and ages	Triggered by growth rate	Optimal space/time balance, complex scheduling

The broker also supports **selective compaction** for environments with different retention requirements across topics:

```
Topic Pattern: user.events.*
Retention: 7 days, immediate compaction of acknowledged messages

Topic Pattern: audit.*
Retention: 7 years, minimal compaction to preserve audit trail

Topic Pattern: temp.*
Retention: 1 hour, aggressive compaction including unacknowledged messages
```

Persistence Architecture Decisions

The persistence layer design involves several critical architectural decisions that significantly impact system performance, reliability, and operational characteristics. Each decision represents a trade-off between competing requirements such as durability versus performance, simplicity versus flexibility, and storage efficiency versus recovery speed.

Decision: Log File Format Structure

- **Context**: The WAL needs a file format that supports efficient sequential writes, random record access, corruption detection, and forward compatibility as the system evolves.
- **Options Considered**:
  1. Plain text format with line-based records
  2. Binary format with fixed-size headers
  3. Self-describing binary format with variable-length records
- **Decision**: Self-describing binary format with variable-length records and embedded checksums
- **Rationale**: Binary format provides superior write performance and space efficiency compared to text. Variable-length records accommodate different message sizes without waste. Self-describing headers enable format evolution and corruption detection.
- **Consequences**: Enables high-throughput logging with excellent corruption detection, but requires more complex parsing logic and binary tooling for log inspection.

Format Option	Write Performance	Space Efficiency	Corruption Detection	Tooling Complexity
Plain text	Medium	Low	Poor	Simple
Fixed binary	High	Medium	Good	Medium
Variable binary	High	High	Excellent	Complex

Decision: Fsync Frequency and Durability Guarantees

- **Context**: The system must balance data durability against write performance. More frequent fsync calls increase durability but reduce throughput.
- **Options Considered**:
  1. Fsync after every record write
  2. Fsync every N records or every T milliseconds
  3. Configurable fsync policy per topic
- **Decision**: Configurable fsync policy with default fsync-per-record for critical topics and batched fsync for high-throughput topics
- **Rationale**: Different applications have different durability vs. performance requirements. Critical financial transactions need immediate fsync, while analytics topics can tolerate batched writes.
- **Consequences**: Enables performance tuning for different use cases but adds configuration complexity and potential for data loss if misconfigured.

Fsync Strategy	Durability	Write Latency	Throughput	Use Case
Per record	Excellent	High (10-50ms)	Low	Financial transactions
Batched (10ms)	Good	Medium (10-20ms)	High	Real-time analytics
Batched (1sec)	Fair	Low (<1ms)	Very high	Log aggregation

Decision: Indexing Strategy for Log Records

- **Context**: Crash recovery requires efficient access to specific record types and time ranges. Sequential scanning becomes impractical as logs grow large.
- **Options Considered**:
  1. No indexing (sequential scan only)
  2. In-memory offset index rebuilt on startup
  3. Persistent B-tree index for record lookups
- **Decision**: In-memory offset index rebuilt during startup with periodic checkpoint persistence
- **Rationale**: In-memory indexes provide fast access during operation while avoiding the complexity of maintaining persistent index consistency. Rebuilding during startup is acceptable for the target system scale.
- **Consequences**: Enables fast log access during normal operation with simple implementation, but increases startup time proportional to log size.

Decision: Checkpoint Strategy and State Snapshots

- **Context**: Crash recovery time grows linearly with log size. Checkpoints enable recovery from a recent consistent state rather than replaying the entire log.
- **Options Considered**:
  1. No checkpoints (always full log replay)
  2. Time-based checkpoints (every N minutes)
  3. Event-based checkpoints (after N records or significant state changes)
- **Decision**: Hybrid strategy with time-based checkpoints every 5 minutes and event-based checkpoints after consumer group rebalancing
- **Rationale**: Time-based checkpoints provide predictable recovery time bounds. Event-based checkpoints capture state after complex operations that would be expensive to replay.
- **Consequences**: Provides fast recovery with reasonable checkpoint overhead, but requires careful coordination between checkpoint timing and ongoing operations.

Decision: Log Compaction Implementation Approach

- **Context**: Append-only logs grow without bound unless compacted. Compaction must not interfere with ongoing operations or compromise crash recovery capabilities.
- **Options Considered**:
  1. Stop-the-world compaction during maintenance windows
  2. Online compaction with copy-on-write log segments
  3. Background compaction with atomic log file replacement
- **Decision**: Background compaction with atomic log file replacement and graceful coordination with active operations
- **Rationale**: Background compaction minimizes operational impact while atomic replacement ensures consistency. Coordination prevents interference with concurrent writes and reads.
- **Consequences**: Enables continuous operation during compaction but requires sophisticated coordination logic and careful handling of file system operations.

Common Persistence Pitfalls

The persistence layer implementation involves several subtle challenges that frequently cause issues for developers building message brokers. These pitfalls often manifest as data loss, corruption, or performance degradation under specific conditions that may not be apparent during initial development and testing.

⚠️ Pitfall: Unbounded Log Growth Without Monitoring

Many implementations focus on the append-only logging mechanism but fail to implement adequate monitoring and alerting for log growth patterns. The log file grows continuously during normal operation, but growth rate can accelerate dramatically under certain conditions such as

consumer failures, high message throughput, or retention policy misconfigurations.

Why this is problematic: Without monitoring, the log file can consume all available disk space, causing the broker to crash when attempting to write new records. Recovery becomes impossible if there's insufficient space to replay the log during startup.

Specific symptoms include broker crashes with "disk full" errors, degraded performance as the file system struggles with very large files, and extremely long recovery times as the replay log becomes enormous.

How to avoid: Implement proactive monitoring that tracks log file size, growth rate, and disk space utilization. Set up alerts when log size exceeds expected thresholds or growth rate accelerates beyond normal patterns. Include disk space checks in health monitoring and ensure the compaction process triggers automatically before space becomes critical.

Monitoring Metric	Alert Threshold	Action Required
Log file size	>10GB or >50% of available disk	Schedule immediate compaction
Growth rate	>100MB/hour sustained	Investigate consumer lag and retention policies
Compaction lag	>7 days since last compaction	Check compaction process health
Disk space	<20% free space remaining	Emergency compaction or disk expansion

⚠️ Pitfall: Fsync Performance vs Durability Trade-offs

The tension between write performance and data durability creates a classic trade-off that many implementations handle poorly. Either they fsync after every write (ensuring durability but creating performance bottlenecks) or they batch writes aggressively (achieving high throughput but risking data loss during crashes).

Why this is problematic: Inappropriate fsync strategies can either make the broker unusably slow or create data loss risks that violate application requirements. The optimal strategy depends on workload characteristics that may change over time.

Specific symptoms include write latencies exceeding 50ms per message (indicating excessive fsync), message loss after broker crashes (indicating insufficient fsync), or highly variable write performance based on disk I/O contention.

How to avoid: Implement configurable fsync strategies that can be tuned per topic or message priority. Provide clear documentation about durability guarantees for each strategy. Use write barriers and group commits to batch fsync calls without sacrificing safety. Monitor fsync frequency and latency to detect when strategies need adjustment.

Fsync Strategy	Latency Impact	Durability Level	Recommended Use Case
Immediate	High (10-50ms)	Maximum	Financial transactions, audit logs
Group commit (10ms)	Medium (5-15ms)	High	Real-time applications
Periodic (1s)	Low (<1ms)	Medium	Analytics, bulk processing
Application controlled	Variable	Custom	Mixed workloads with explicit requirements

⚠️ Pitfall: Corrupted Log Recovery Edge Cases

Log corruption can occur due to hardware failures, power outages, or file system issues. Many implementations handle obvious corruption (completely unreadable files) but fail gracefully when facing subtle corruption such as partially written records, corrupted checksums, or inconsistent record boundaries.

Why this is problematic: Partial corruption can cause recovery to fail completely, even when most of the log is intact and recoverable. Aggressive corruption recovery can also lead to data inconsistency if invalid data is accepted as valid.

Specific symptoms include recovery processes that crash with parsing errors, brokers that start successfully but contain inconsistent state, or recovery that appears successful but results in missing or duplicate messages.

How to avoid: Implement robust corruption detection using checksums, magic numbers, and record length validation. Create recovery policies that can skip corrupted records while logging detailed information about what was lost. Provide administrative tools for manual log inspection and repair. Test recovery logic with artificially corrupted logs to verify edge case handling.

Corruption Type	Detection Method	Recovery Strategy
Partial record write	Record length vs. file end	Truncate at last valid record boundary
Checksum mismatch	CRC32 validation	Skip record, log corruption details
Invalid magic number	Header validation	Scan forward for next valid record
Inconsistent offsets	Sequential offset validation	Rebuild offset index from scratch

⚠️ Pitfall: Checkpoint Consistency and Recovery State Validation

Checkpoint creation involves capturing a consistent snapshot of broker state while the system continues processing messages. Many implementations create checkpoints that are internally inconsistent (containing references to messages that haven't been logged) or fail to validate checkpoint integrity during recovery.

Why this is problematic: Inconsistent checkpoints can cause recovery to restore the broker to an invalid state where consumer assignments reference non-existent messages, topic metadata doesn't match actual log contents, or acknowledgment tracking contains orphaned entries.

Specific symptoms include recovery that completes successfully but results in lost messages, consumer assignment errors immediately after recovery, or phantom messages that appear acknowledged but were never delivered.

How to avoid: Use write barriers to ensure all referenced log records are persisted before creating checkpoints. Implement comprehensive checkpoint validation during recovery that verifies all references are valid and consistent. Create atomic checkpoint operations that either complete entirely or fail cleanly without partial state.

Implementation Guidance

This section provides concrete implementation guidance for building the persistence layer using Go. The focus is on providing complete, working infrastructure components while leaving the core learning challenges as exercises for the learner.

Technology Recommendations

Component	Simple Option	Advanced Option
File I/O	`os.File` with manual buffering	`bufio` with configurable buffer sizes
Serialization	`encoding/json` for record payloads	`encoding/gob` or Protocol Buffers
Checksums	`hash/crc32` for record integrity	`crypto/sha256` for cryptographic guarantees
File system operations	`os.Sync()` for explicit fsync	`syscall.Fdatasync()` for performance
Background tasks	`time.Ticker` for periodic compaction	Channel-based task scheduling
Log indexing	In-memory `map[int64]RecordMetadata`	Embedded database like BoltDB

Recommended File Structure

```
internal/persistence/
  wal.go                  ← Write-Ahead Log interface and implementation
  wal_test.go             ← Comprehensive test suite with corruption scenarios
  record.go               ← WAL record types and serialization
  compaction.go           ← Background compaction process
  checkpoint.go           ← State checkpointing and recovery
  recovery.go             ← Crash recovery algorithm implementation
  log_reader.go           ← Sequential and random record access
  log_writer.go           ← Buffered, atomic record writing
  retention.go            ← Retention policy enforcement
  monitoring.go           ← Persistence metrics and health checks
  testdata/               ← Sample log files for testing
    corrupted_log.wal     ← Test files with various corruption patterns
    valid_checkpoint.json ← Reference checkpoint format
```

Complete WAL Infrastructure (Ready to Use)

This infrastructure provides the basic file operations and record formatting, allowing learners to focus on the core persistence logic rather than low-level file handling:

```go
package persistence

import (
    "bufio"
    "encoding/binary"
    "encoding/json"
    "hash/crc32"
    "io"
    "os"
    "path/filepath"
    "sync"
    "time"
)

// WALWriter handles atomic record writing with durability guarantees
type WALWriter struct {
    file     *os.File
    writer   *bufio.Writer
    offset   int64
    fsyncPolicy FsyncPolicy
    mutex    sync.Mutex
}

// FsyncPolicy defines when to force disk synchronization
type FsyncPolicy struct {
    Mode     string        // "immediate", "batched", "periodic"
    Interval time.Duration // for periodic mode
    BatchSize int          // for batched mode
}

// NewWALWriter creates a new writer for the specified log file
func NewWALWriter(filename string, policy FsyncPolicy) (*WALWriter, error) {
    file, err := os.OpenFile(filename, os.O_CREATE|os.O_APPEND|os.O_WRONLY, 0644)
    if err != nil {
        return nil, err
    }

    // Get current file size to track offset
```

```
    stat, err := file.Stat()
    if err != nil {
        file.Close()
        return nil, err
    }

    return &WALWriter{
        file:        file,
        writer:      bufio.NewWriter(file),
        offset:      stat.Size(),
        fsyncPolicy: policy,
    }, nil
}

// WriteRecord writes a complete WAL record atomically
func (w *WALWriter) WriteRecord(recordType WALRecordType, data []byte) (int64, error) {
    w.mutex.Lock()
    defer w.mutex.Unlock()

    record := WALRecord{
        Magic:         0x57414C47, // "WALG"
        Version:       1,
        RecordType:    recordType,
        Timestamp:     time.Now().UnixNano(),
        TransactionID: generateTransactionID(),
        Data:          json.RawMessage(data),
    }

    // Calculate checksum over data
    record.Checksum = crc32.ChecksumIEEE(data)

    // Serialize record
    serialized, err := json.Marshal(record)
    if err != nil {
        return 0, err
    }
```

```
    record.RecordLength = uint32(len(serialized))

    // Write length prefix
    if err := binary.Write(w.writer, binary.LittleEndian, record.RecordLength); err != nil {
        return 0, err
    }

    // Write record data
    if _, err := w.writer.Write(serialized); err != nil {
        return 0, err
    }

    currentOffset := w.offset
    w.offset += int64(4 + len(serialized))

    // Apply fsync policy
    if w.shouldFsync() {
        if err := w.flush(); err != nil {
            return 0, err
        }
    }

    return currentOffset, nil
}

func (w *WALWriter) shouldFsync() bool {
    switch w.fsyncPolicy.Mode {
    case "immediate":
        return true
    case "batched":
        // TODO: Implement batch counting logic
        return false
    case "periodic":
        // TODO: Implement periodic flushing
        return false
```

```
    default:
        return true
    }
}

func (w *WALWriter) flush() error {
    if err := w.writer.Flush(); err != nil {
        return err
    }
    return w.file.Sync()
}

// WALReader handles sequential and random access to log records
type WALReader struct {
    file   *os.File
    reader *bufio.Reader
    offset int64
    mutex  sync.RWMutex
}

func NewWALReader(filename string) (*WALReader, error) {
    file, err := os.OpenFile(filename, os.O_RDONLY, 0)
    if err != nil {
        return nil, err
    }

    return &WALReader{
        file:   file,
        reader: bufio.NewReader(file),
        offset: 0,
    }, nil
}

// ReadNextRecord reads the next record sequentially
func (r *WALReader) ReadNextRecord() (*WALRecord, error) {
    r.mutex.Lock()
    defer r.mutex.Unlock()
```

```
    // Read record length
    var recordLength uint32
    if err := binary.Read(r.reader, binary.LittleEndian, &recordLength); err != nil {
        return nil, err
    }

    // Read record data
    recordData := make([]byte, recordLength)
    if _, err := io.ReadFull(r.reader, recordData); err != nil {
        return nil, err
    }

    // Deserialize record
    var record WALRecord
    if err := json.Unmarshal(recordData, &record); err != nil {
        return nil, err
    }

    // Validate record
    if err := r.validateRecord(&record); err != nil {
        return nil, err
    }

    r.offset += int64(4 + recordLength)
    return &record, nil
}

func (r *WALReader) validateRecord(record *WALRecord) error {
    // Validate magic number
    if record.Magic != 0x57414C47 {
        return fmt.Errorf("invalid magic number: %x", record.Magic)
    }

    // Validate checksum
    expectedChecksum := crc32.ChecksumIEEE(record.Data)
```

```
    if record.Checksum != expectedChecksum {
        return fmt.Errorf("checksum mismatch: expected %x, got %x",
            expectedChecksum, record.Checksum)
    }

    return nil
}

func generateTransactionID() uint64 {
    return uint64(time.Now().UnixNano())
}
```

Core Persistence Logic Skeleton (For Learner Implementation)

```go
// WAL provides durable logging for message broker operations
type WAL struct {
    writer        *WALWriter
    reader        *WALReader
    logDir        string
    compactor     *LogCompactor
    checkpointer  *Checkpointer
    config        WALConfig

    // Metrics and monitoring
    recordCount   int64
    bytesWritten  int64
    lastCompaction time.Time
}

// AppendRecord writes a record to the WAL with durability guarantees
func (w *WAL) AppendRecord(recordType WALRecordType, data []byte) (int64, error) {
    // TODO 1: Validate that recordType is supported and data is not empty
    // TODO 2: Check if log rotation is needed based on size/time policies
    // TODO 3: Call writer.WriteRecord() and handle any I/O errors
    // TODO 4: Update metrics (recordCount, bytesWritten) atomically
    // TODO 5: Schedule background compaction if thresholds exceeded
    // TODO 6: Return the log offset for future reference
    // Hint: Use atomic.AddInt64 for thread-safe metrics updates
}

// RecoverFromCrash rebuilds broker state by replaying the WAL
func (w *WAL) RecoverFromCrash() (*BrokerState, error) {
    // TODO 1: Find the latest valid checkpoint and load its metadata
    // TODO 2: Open WAL reader starting from checkpoint offset
    // TODO 3: Create empty BrokerState to populate during recovery
    // TODO 4: Read records sequentially, applying each to rebuild state:
    //         - RecordMessagePublished: Add to topic queues
    //         - RecordMessageAcked: Remove from pending acknowledgments
    //         - RecordConsumerJoined: Add to consumer groups
    // TODO 5: Validate final state consistency (no orphaned references)
```

```
    // TODO 6: Create new checkpoint with recovered state
    // TODO 7: Return populated BrokerState ready for normal operations
    // Hint: Keep track of corrupt records and log warnings, but continue recovery
}

// TriggerCompaction starts background log compaction process
func (w *WAL) TriggerCompaction(retentionPolicy RetentionPolicy) error {
    // TODO 1: Check if compaction is already running (prevent concurrent compaction)
    // TODO 2: Determine compaction window based on retention policy
    // TODO 3: Build entity state index by scanning compaction window
    // TODO 4: Generate compaction plan (records to keep vs. remove)
    // TODO 5: Create new log file with compacted records
    // TODO 6: Update checkpoints to reference new log structure
    // TODO 7: Atomically replace old log files with compacted version
    // TODO 8: Update metrics and schedule next compaction
    // Hint: Use file rename for atomic log replacement
}
```

Milestone Checkpoint

After implementing the persistence layer, verify the following behavior:

Test 1: Basic Logging

```bash
go test ./internal/persistence/ -v -run TestBasicLogging
```

Expected: All record types can be written and read back correctly with proper checksums

Test 2: Crash Recovery

```bash
go test ./internal/persistence/ -v -run TestCrashRecovery
```

Expected: Broker state is correctly restored from WAL after simulated crash

Test 3: Log Compaction

```bash
go test ./internal/persistence/ -v -run TestLogCompaction
```

Expected: Log size decreases after compaction while preserving all necessary recovery data

Manual Verification:

1. Start broker, publish several messages to different topics
2. Kill broker process with SIGKILL (simulating crash)
3. Restart broker and verify all unacknowledged messages are restored
4. Check that log file grows during operation and shrinks after compaction

Performance Benchmarks:

- Write throughput: >10,000 records/second with immediate fsync

- Recovery speed: <1 second per 100,000 records
- Compaction efficiency: >50% size reduction in typical workloads

Common Debugging Issues

Symptom	Likely Cause	How to Diagnose	Fix
Recovery fails with "corrupted record"	Incomplete write during crash	Check file size vs. last record boundary	Truncate log at last valid record
Slow write performance	Excessive fsync calls	Monitor fsync frequency in logs	Adjust fsync policy to batched mode
Memory usage grows during compaction	Large compaction window	Check compaction window size calculation	Implement incremental compaction
Messages lost after recovery	Missing fsync before acknowledgment	Check fsync ordering in append path	Ensure fsync before sending ACK responses
Log file never shrinks	Compaction not triggering	Check retention policy thresholds	Lower compaction triggers or force manual compaction

Backpressure Management

> **Milestone(s):** Milestone 3 (Persistence & Backpressure) — implements flow control to handle slow consumers and prevent system overload

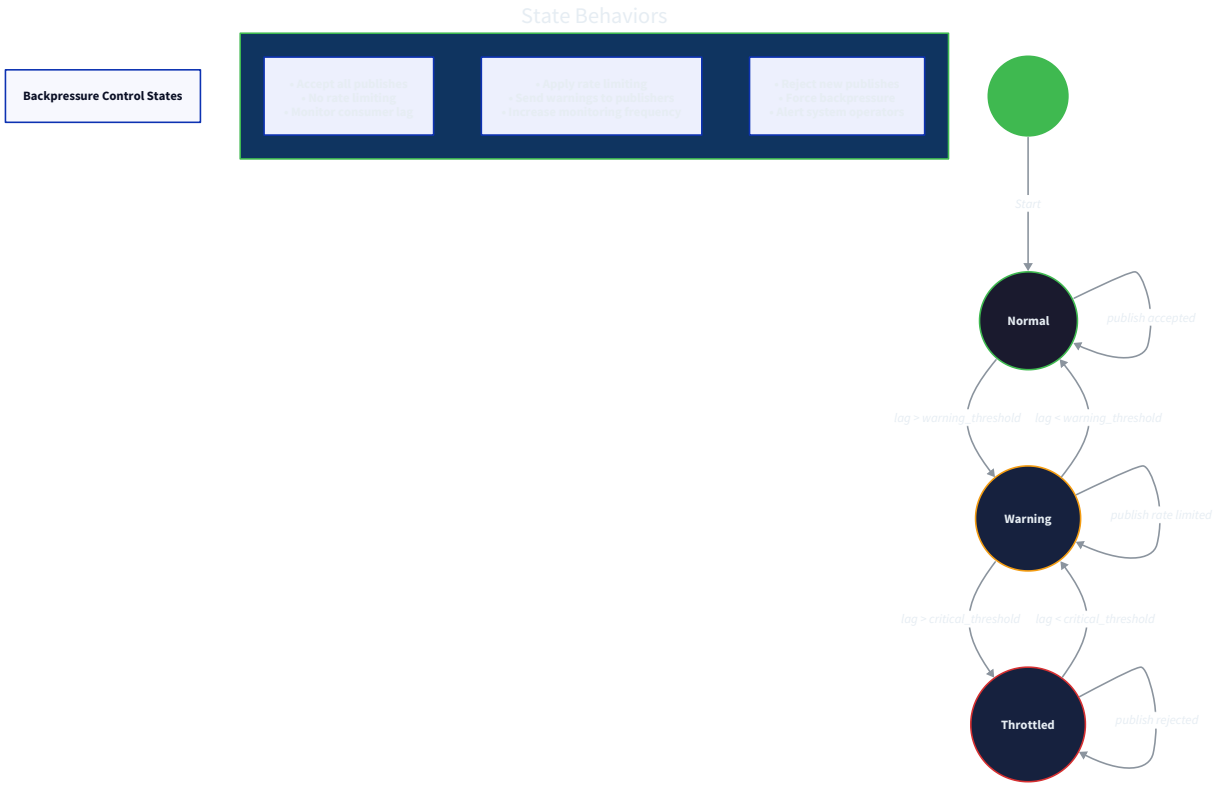
Mental Model: Traffic Flow Control

Imagine a busy highway system during rush hour. When traffic enters the highway faster than it can exit, congestion builds up. Without intervention, the entire system grinds to a halt. Traffic engineers solve this with sophisticated flow control mechanisms: metering lights at on-ramps that temporarily slow new traffic entering the highway, variable speed limits that optimize throughput, and alternate route suggestions that distribute load across the network.

Your message queue faces the same challenge. **Publishers** are like cars entering the highway — they arrive at unpredictable rates and want immediate service. **Consumers** are like highway exits — they have finite processing capacity and can become overwhelmed. **Backpressure** is your traffic control system, automatically detecting congestion and applying precisely the right amount of throttling to keep the entire system flowing smoothly.

The key insight is that temporary slowdowns prevent complete system failure. Just as a brief red light at an on-ramp prevents a multi-hour traffic jam, temporarily pausing a fast publisher prevents message queue overflow that could crash your entire broker. The goal isn't to eliminate queuing entirely — some buffering is healthy and improves efficiency. Instead, you want to maintain **controlled congestion** where the system operates at high throughput without crossing into the danger zone of cascading failures.

Unlike highway traffic, message systems offer additional control mechanisms. You can prioritize critical messages (like emergency vehicles using the shoulder), implement fair queuing across multiple publishers (like HOV lanes), and even provide feedback to publishers about current conditions (like electronic highway signs showing expected delays). This rich control plane enables sophisticated flow management that adapts automatically to changing conditions.



Flow Controller Interface

The **Flow Controller** component monitors system capacity and applies throttling when consumer lag exceeds acceptable thresholds. This component acts as the central nervous system for backpressure decisions, continuously measuring system health and applying the minimum intervention necessary to maintain stability.

The Flow Controller maintains awareness of system-wide state by tracking consumer lag across all topics and consumer groups. When healthy, the system operates in **normal mode** with no restrictions. As consumer lag increases, it transitions through **warning mode** (monitoring closely but not intervening) to **throttled mode** (actively limiting publisher throughput). This graduated response prevents sudden jarring changes that could destabilize the system.

Method Name	Parameters	Returns	Description
`CheckCapacity`	`topicName string, consumerGroupID string`	`CapacityStatus, error`	Evaluates current system capacity for the specified topic and consumer group combination
`ApplyBackpressure`	`publisherID string, throttleLevel ThrottleLevel`	`error`	Applies throttling to a specific publisher with the given intensity level
`ReleaseBackpressure`	`publisherID string`	`error`	Removes throttling restrictions from a publisher when conditions improve
`GetSystemMetrics`	`none`	`SystemMetrics, error`	Returns comprehensive system health metrics including lag, throughput, and throttle status
`UpdateLagThresholds`	`config ThresholdConfig`	`error`	Dynamically adjusts the lag thresholds that trigger backpressure decisions
`RegisterPublisher`	`publisherID string, priority PublisherPriority`	`error`	Registers a new publisher with the flow control system and assigns priority level
`UnregisterPublisher`	`publisherID string`	`error`	Removes publisher from flow control tracking and cleans up associated state
`ProcessHeartbeat`	`consumerID string, lagMetrics ConsumerLagMetrics`	`error`	Updates consumer lag information based on heartbeat data from acknowledgment tracker

The Flow Controller interfaces closely with the Acknowledgment Component to receive real-time lag measurements and with the Protocol Handler to actually enforce throttling decisions. This separation of concerns allows the Flow Controller to focus on decision-making while delegating enforcement to components that already manage the relevant connections and state.

Capacity Status represents the current health of a specific topic-consumer group combination:

Field Name	Type	Description
`Status`	`CapacityLevel`	Current capacity state: `Normal`, `Warning`, `Critical`, or `Overloaded`
`CurrentLag`	`time.Duration`	Time delay between message publication and consumer processing
`MessageBacklog`	`int64`	Number of unprocessed messages waiting for consumers
`LagTrend`	`TrendDirection`	Whether lag is `Increasing`, `Stable`, or `Decreasing` over recent time window
`RecommendedAction`	`ThrottleAction`	Suggested response: `NoAction`, `LightThrottle`, `ModerateThrottle`, or `HeavyThrottle`
`EstimatedCatchupTime`	`time.Duration`	Projected time for consumers to process current backlog at current rate
`ActiveConsumers`	`int`	Number of healthy consumers currently processing messages

> **Key Design Principle**: The Flow Controller never makes instantaneous decisions based on single data points. Instead, it analyzes trends over time windows to avoid thrashing between throttled and non-throttled states when lag oscillates around threshold boundaries.

Consumer Lag Monitoring

Consumer lag represents the time delay between when a message is published and when it's actually processed by a consumer. This is the fundamental metric that drives all backpressure decisions. Lag monitoring requires sophisticated measurement techniques because simple queue depth doesn't tell the full story — a queue with 1000 messages might be healthy if consumers are processing them quickly, but critical if consumers have stalled.

The lag monitoring system tracks multiple dimensions of consumer performance to build a comprehensive picture of system health. **End-to-end lag** measures the time from message publication to consumer acknowledgment. **Assignment lag** measures the time from publication to consumer

assignment (useful for diagnosing consumer group rebalancing issues). **Processing lag** measures the time from assignment to acknowledgment (useful for identifying slow consumer logic).

The monitoring system maintains lag measurements per topic, per consumer group, and per individual consumer. This granularity enables targeted interventions — you might throttle publishers to a specific topic experiencing consumer issues while allowing normal throughput to healthy topics.

Consumer Lag Tracking Implementation:

The lag monitoring system continuously collects performance data from multiple sources:

1. **Message timestamp tracking**: Every published message includes a high-precision timestamp that enables accurate lag calculation when the message is acknowledged.
2. **Consumer heartbeat integration**: Consumer heartbeats include lag self-reports where consumers indicate their current message processing latency.
3. **Assignment tracker integration**: The acknowledgment tracker provides data about message assignment delays and retry patterns.
4. **Trend analysis**: The system maintains sliding windows of lag measurements to identify trends and avoid making decisions based on temporary spikes.

Lag Metric	Measurement Window	Threshold Warning	Threshold Critical	Action Triggered
`End-to-End Lag`	`30 seconds`	`5 seconds`	`15 seconds`	Publisher throttling
`Assignment Lag`	`10 seconds`	`2 seconds`	`5 seconds`	Consumer group rebalancing
`Processing Lag`	`60 seconds`	`10 seconds`	`30 seconds`	Consumer health check
`Queue Depth`	`5 seconds`	`1000 messages`	`5000 messages`	Emergency throttling
`Consumer Throughput`	`120 seconds`	`< 50% of baseline`	`< 25% of baseline`	Dead consumer detection

The monitoring system uses **exponential weighted moving averages** to smooth out temporary fluctuations while still responding quickly to genuine trends. This mathematical approach prevents the system from overreacting to brief lag spikes caused by garbage collection pauses or network hiccups while still detecting sustained performance degradation quickly enough to prevent system overload.

> **Critical Implementation Detail**: Lag measurements must account for clock skew between publishers and consumers. The system uses broker-local timestamps as the source of truth, recording timestamps when messages arrive at the broker rather than relying on publisher-provided timestamps.

Producer Throttling Strategies

When backpressure triggers are activated, the system must reduce publisher throughput without causing cascading failures. The Flow Controller implements multiple throttling strategies with different characteristics suitable for various scenarios.

Rate Limiting Strategy applies mathematical limits to publisher message rates. Instead of allowing unlimited message publication, publishers receive a quota of messages per time period. When they exceed this quota, additional publish requests are delayed until the next quota period begins. This approach provides smooth, predictable throttling that's easy for publishers to understand and adapt to.

Connection Pausing Strategy temporarily blocks publisher connections when severe backpressure occurs. The broker stops reading from publisher TCP sockets, causing the publisher's network send buffers to fill and naturally slowing their message rate. This approach requires no cooperation from publishers and works with any client implementation, but provides less fine-grained control than rate limiting.

Dynamic Backoff Strategy uses exponential backoff algorithms to slow publishers progressively. When backpressure first triggers, publishers experience minimal delays. If congestion persists, delays increase exponentially up to a configured maximum. When conditions improve, delays decrease gradually to avoid sudden traffic bursts that could re-trigger congestion.

Priority-Based Throttling applies different throttling levels based on publisher importance. Critical system publishers (like health monitoring or security alerts) might never be throttled, while bulk data imports face aggressive limits during congestion. This requires publishers to declare their priority level during connection establishment.

Throttling Strategy	Response Time	Publisher Impact	Implementation Complexity	Effectiveness
Rate Limiting	Fast (< 100ms)	Predictable delays	Medium	High for cooperative clients
Connection Pausing	Immediate	Blocked sends	Low	Very high, works with any client
Dynamic Backoff	Gradual (1-10s)	Increasing delays	Medium	High for sustained congestion
Priority-Based	Variable	Depends on priority	High	Very high when properly configured

The Flow Controller can combine multiple strategies for maximum effectiveness. For example, it might start with rate limiting for cooperative publishers, escalate to connection pausing if congestion worsens, and always respect priority levels throughout the process.

Throttling Implementation Algorithm:

1. **Continuous monitoring**: The Flow Controller evaluates system lag every few seconds using measurements from the Consumer Lag Monitoring component.
2. **Threshold evaluation**: If any topic exceeds warning thresholds, the controller identifies the publishers contributing to that topic and calculates appropriate throttling levels.
3. **Strategy selection**: Based on congestion severity and publisher characteristics, the controller selects the most appropriate throttling approach for each publisher.
4. **Gradual application**: Throttling begins at minimal levels and increases gradually if congestion persists, avoiding sudden throughput drops that could trigger upstream failures.
5. **Feedback loop**: The controller monitors the effectiveness of throttling decisions and adjusts parameters dynamically based on observed lag improvements.
6. **Release conditions**: When lag falls below warning thresholds and remains stable, throttling is gradually reduced to allow throughput recovery.

Backpressure Architecture Decisions

The design of the backpressure system requires careful consideration of multiple competing concerns: responsiveness versus stability, simplicity versus flexibility, and local optimization versus global efficiency.

> **Decision: Centralized vs Distributed Backpressure Control**
>
> - **Context**: Backpressure decisions could be made by individual topic managers (distributed) or by a central flow controller (centralized). Distributed control responds faster to local conditions but may create conflicting decisions across topics. Centralized control provides global optimization but introduces a potential bottleneck.
> - **Options Considered**: Distributed per-topic control, centralized global control, hybrid approach with local feedback to central coordinator
> - **Decision**: Centralized flow controller with fast local feedback loops
> - **Rationale**: Message queues often have cross-topic dependencies where throttling one topic affects others. A centralized controller can make globally optimal decisions and avoid situations where per-topic controllers fight each other. The performance bottleneck concern is mitigated by implementing the controller as an asynchronous service that makes decisions based on periodic health reports rather than blocking the message delivery path.
> - **Consequences**: Enables sophisticated cross-topic optimization and prevents conflicting throttling decisions. Introduces a single point of failure that requires careful reliability design. Requires efficient communication between the central controller and distributed topic managers.

Decision: Proactive vs Reactive Throttling

- **Context**: Backpressure can be applied reactively (after congestion is detected) or proactively (before congestion occurs based on predictive models). Reactive throttling is simpler but may allow brief periods of system overload. Proactive throttling prevents overload entirely but risks unnecessary throttling of healthy publishers.
- **Options Considered**: Pure reactive based on lag thresholds, pure proactive based on capacity modeling, hybrid approach with proactive throttling and reactive safety nets
- **Decision**: Hybrid approach with primary reliance on reactive throttling and proactive elements for known patterns
- **Rationale**: Message queue workloads are often unpredictable, making pure proactive approaches prone to false positives. However, some patterns are predictable (like daily batch jobs), where proactive throttling improves system stability. The hybrid approach captures the benefits of both while minimizing their respective downsides.
- **Consequences**: Provides robust protection against unexpected load spikes while optimizing for known patterns. Requires more complex implementation with both reactive monitoring and proactive capacity planning. May occasionally throttle publishers unnecessarily during unusual but benign traffic patterns.

Decision Factor	Distributed Control	Centralized Control	Hybrid Approach
Response Latency	Excellent (< 10ms)	Good (50-100ms)	Good (20-50ms)
Global Optimization	Poor	Excellent	Good
Complexity	Low	Medium	High
Single Point of Failure	No	Yes	Partial
Cross-Topic Coordination	Poor	Excellent	Good

Decision: Lag Threshold Configuration Strategy

- **Context**: Backpressure thresholds could be static (configured at startup), dynamic (automatically adjusted based on system behavior), or adaptive (manually tunable with automatic recommendations). Static thresholds are simple but may not adapt to changing workloads. Dynamic thresholds adapt automatically but may destabilize during unusual conditions.
- **Options Considered**: Static configuration with manual tuning, fully automated dynamic adjustment, adaptive system with manual override capability
- **Decision**: Adaptive system with conservative defaults and manual override capability
- **Rationale**: Message queue deployments vary dramatically in their characteristics — some handle steady background traffic while others process bursty interactive workloads. No single set of static thresholds works well across all scenarios. However, fully automated systems can make poor decisions during unusual but valid traffic patterns. The adaptive approach provides good defaults that work for most scenarios while allowing operators to override decisions when they understand their specific workload better than the automated system.
- **Consequences**: Requires sophisticated threshold adjustment algorithms and comprehensive monitoring to make good automatic recommendations. Provides escape hatches for operators when automatic decisions are suboptimal. May require more operational expertise to configure optimally than purely static or purely automatic alternatives.

Common Backpressure Pitfalls

Implementing effective backpressure requires avoiding several common mistakes that can destabilize the very systems backpressure is meant to protect.

⚠️ Pitfall: Cascading Backpressure Amplification

A dangerous failure mode occurs when backpressure triggers upstream failures that create even more congestion. For example, when the message broker throttles a publisher, that publisher's internal queues might fill up, causing it to reject requests from its clients. Those clients might then retry aggressively, creating more load on the publisher and indirectly more pressure on the message broker.

Why this happens: Each component in a distributed system typically has its own internal queues and retry logic. When one component slows down, it can trigger cascading effects throughout the upstream chain. Without careful coordination, these effects can amplify rather than reduce the original congestion.

Detection signs: System-wide latency increases even after backpressure is applied. Error rates increase across multiple services, not just the originally congested components. Retry storm patterns appear in monitoring dashboards.

Prevention strategies: Implement **circuit breaker patterns** in publishers to stop retrying when backpressure is detected. Use **exponential backoff with jitter** to prevent synchronized retry storms. Provide **explicit backpressure signals** (like HTTP 503 responses) so upstream components can distinguish throttling from failures and respond appropriately.

⚠️ Pitfall: Overly Aggressive Throttling

Another common mistake is applying excessive throttling that reduces system throughput far below what's actually necessary to resolve congestion. This often happens when engineers tune backpressure systems to handle worst-case scenarios, resulting in unnecessary throttling during normal load variations.

Why this happens: Backpressure systems often use simple threshold-based algorithms that don't distinguish between temporary lag spikes (which resolve naturally) and sustained congestion (which requires intervention). Additionally, many implementations fail to account for the natural latency of their feedback loops — by the time throttling takes effect, the original congestion may have already resolved.

Detection signs: System throughput drops significantly more than necessary to resolve congestion. Consumer lag returns to normal levels quickly but throttling continues for extended periods. Publishers report unnecessary delays during periods when consumers are actually keeping up with load.

Prevention strategies: Implement **gradual throttling ramp-up and ramp-down** rather than binary on/off throttling. Use **trend analysis** to distinguish between temporary spikes and sustained issues. Monitor **throttling effectiveness** to ensure that applied throttling actually correlates with improved consumer lag.

⚠️ Pitfall: Ignoring Consumer Heterogeneity

Many backpressure implementations assume that all consumers in a consumer group have similar processing capabilities and apply uniform throttling based on the slowest consumer. This can dramatically under-utilize fast consumers and create unnecessary bottlenecks.

Why this happens: Consumer group abstractions often hide the performance characteristics of individual consumers. Additionally, implementing per-consumer backpressure is significantly more complex than group-level throttling, leading many implementations to choose the simpler approach.

Detection signs: Some consumers in a group remain idle while others are overloaded. Message processing throughput is limited by the slowest consumer even when faster consumers are available. Consumer group rebalancing doesn't improve overall throughput.

Prevention strategies: Implement **consumer-aware message assignment** that considers individual consumer performance when distributing messages. Monitor **per-consumer lag and throughput** rather than just group-level metrics. Use **dynamic consumer group sizing** to automatically add more consumer instances when individual consumers become overwhelmed.

⚠️ Pitfall: Insufficient Backpressure Granularity

Applying backpressure at too coarse a level (like entire broker or entire publisher) when the actual congestion is more localized (specific topics or consumer groups) can unnecessarily impact healthy parts of the system.

Why this happens: Implementing fine-grained backpressure requires more complex monitoring and control mechanisms. It's often easier to implement broker-wide throttling than to track and manage backpressure separately for each topic and consumer group combination.

Detection signs: Healthy topics experience throttling when only specific topics are actually congested. Publishers that send to multiple topics are throttled unnecessarily when only a subset of their topics are experiencing issues.

Prevention strategies: Implement **topic-level backpressure tracking** with separate thresholds and throttling decisions for each topic. Allow **publisher registration per topic** so that multi-topic publishers can be throttled selectively. Provide **backpressure feedback** that indicates which specific topics are causing throttling so publishers can adapt their sending patterns.

Implementation Guidance

The backpressure management system requires careful integration with existing components while maintaining clean separation of concerns. The implementation focuses on making the right decision-making information available at the right time while minimizing performance overhead.

Technology Recommendations:

Component	Simple Option	Advanced Option
Lag Metrics Storage	In-memory circular buffers with periodic aggregation	Time-series database (InfluxDB, Prometheus)
Throttling Implementation	Token bucket rate limiting with Go channels	Sliding window counters with Redis
Trend Analysis	Exponential weighted moving averages	Statistical regression with confidence intervals
Configuration Management	YAML/JSON config files with hot reload	Dynamic configuration service (etcd, Consul)

Recommended File Structure:

```
internal/
├── backpressure/
│   ├── flow_controller.go       ← main backpressure coordination logic
│   ├── flow_controller_test.go  ← comprehensive test coverage
│   ├── lag_monitor.go           ← consumer lag tracking and analysis
│   ├── throttle_strategies.go   ← rate limiting and connection pausing implementations
│   ├── metrics_collector.go     ← lag measurement and trend analysis
│   └── config.go               ← threshold configuration and validation
├── protocol/
│   └── throttling.go           ← publisher throttling enforcement in protocol handler
└── acktracker/
    └── lag_reporting.go        ← lag data integration with acknowledgment tracker
```

This structure separates backpressure decision-making (in the `backpressure` package) from enforcement (in existing `protocol` and `acktracker` packages), enabling clean testing and avoiding circular dependencies.

Infrastructure Starter Code:

```go
package backpressure

import (
    "sync"
    "time"
)

// LagMetricsCollector provides infrastructure for collecting and analyzing
// consumer lag measurements from multiple sources.
type LagMetricsCollector struct {
    measurements map[string]*CircularBuffer
    trends       map[string]*TrendAnalyzer
    mutex        sync.RWMutex
    config       MetricsConfig
}

type CircularBuffer struct {
    values    []float64
    index     int
    full      bool
    capacity  int
    mutex     sync.RWMutex
}

func (cb *CircularBuffer) Add(value float64) {
    cb.mutex.Lock()
    defer cb.mutex.Unlock()

    cb.values[cb.index] = value
    cb.index = (cb.index + 1) % cb.capacity
    if cb.index == 0 {
        cb.full = true
    }
}

func (cb *CircularBuffer) GetAverage() float64 {
    cb.mutex.RLock()
    defer cb.mutex.RUnlock()
```

```
    count := cb.capacity
    if !cb.full {
        count = cb.index
    }

    if count == 0 {
        return 0
    }

    sum := 0.0
    for i := 0; i < count; i++ {
        sum += cb.values[i]
    }
    return sum / float64(count)
}

// TokenBucketThrottler provides infrastructure for rate-limiting publishers
// using the token bucket algorithm.
type TokenBucketThrottler struct {
    buckets    map[string]*TokenBucket
    mutex      sync.RWMutex
    defaultRate time.Duration
}

type TokenBucket struct {
    tokens     int
    maxTokens  int
    refillRate time.Duration
    lastRefill time.Time
    mutex      sync.Mutex
}

func (tb *TokenBucket) TryConsume(tokens int) bool {
    tb.mutex.Lock()
    defer tb.mutex.Unlock()
```

```
    now := time.Now()
    elapsed := now.Sub(tb.lastRefill)

    // Refill tokens based on elapsed time
    tokensToAdd := int(elapsed / tb.refillRate)
    if tokensToAdd > 0 {
        tb.tokens = min(tb.maxTokens, tb.tokens + tokensToAdd)
        tb.lastRefill = now
    }

    if tb.tokens >= tokens {
        tb.tokens -= tokens
        return true
    }
    return false
}

func min(a, b int) int {
    if a < b {
        return a
    }
    return b
}
```

Core Logic Skeleton:

```go
package backpressure

// FlowController coordinates backpressure decisions across the entire message broker.
// It monitors consumer lag, evaluates capacity constraints, and applies appropriate
// throttling to maintain system stability.
type FlowController struct {
    lagMonitor      *LagMonitor
    throttleManager *ThrottleManager
    config          *BackpressureConfig
    metrics         *SystemMetrics
    publishers      map[string]*PublisherState
    mutex           sync.RWMutex
}

// CheckCapacity evaluates current system capacity for the specified topic and consumer group.
// Returns capacity status that includes current lag, trend analysis, and recommended actions.
func (fc *FlowController) CheckCapacity(topicName string, consumerGroupID string) (CapacityStatus, error) {
    // TODO 1: Retrieve current lag measurements for the topic/group combination
    // TODO 2: Calculate trend direction (increasing, stable, decreasing) over recent time window
    // TODO 3: Evaluate current lag against configured warning and critical thresholds
    // TODO 4: Estimate catchup time based on current consumer throughput and backlog size
    // TODO 5: Generate recommended throttling action based on lag severity and trend
    // TODO 6: Return comprehensive capacity status with all calculated metrics

    // Hint: Use fc.lagMonitor.GetLagMeasurement(topicName, consumerGroupID)
    // Hint: Consider both absolute lag values and rate of change when making decisions
}

// ApplyBackpressure implements throttling for a specific publisher based on system capacity.
// Selects appropriate throttling strategy based on severity and publisher characteristics.
func (fc *FlowController) ApplyBackpressure(publisherID string, throttleLevel ThrottleLevel) error {
    // TODO 1: Look up publisher configuration and current throttle state
    // TODO 2: Select appropriate throttling strategy based on throttle level and publisher priority
    // TODO 3: Calculate specific throttling parameters (rate limits, delay amounts, etc.)
    // TODO 4: Apply throttling through the appropriate mechanism (rate limiting, connection pausing, etc.)
    // TODO 5: Record throttling decision in metrics for monitoring and debugging
    // TODO 6: Schedule automatic re-evaluation to potentially release throttling later
```

```
    // Hint: Publisher priority affects which throttling strategies are available
    // Hint: Gradual throttling ramp-up prevents sudden throughput drops
}

// ProcessLagUpdate handles incoming lag measurements from consumers and acknowledgment tracker.
// Updates internal lag tracking and triggers backpressure decisions if thresholds are exceeded.
func (fc *FlowController) ProcessLagUpdate(update LagUpdateEvent) error {
    // TODO 1: Validate lag update event and extract relevant measurements
    // TODO 2: Update internal lag tracking with new measurements
    // TODO 3: Calculate updated trend analysis incorporating new data point
    // TODO 4: Check if updated lag measurements exceed any configured thresholds
    // TODO 5: If thresholds exceeded, identify publishers contributing to affected topics
    // TODO 6: Calculate appropriate throttling levels and apply backpressure as needed
    // TODO 7: If lag has improved, evaluate whether existing throttling can be reduced

    // Hint: Use trend analysis to avoid thrashing between throttled/unthrottled states
    // Hint: Consider lag improvements as well as degradations to release unnecessary throttling
}
```

Milestone Checkpoint:

After implementing the backpressure system, verify the following behavior:

1. **Start the broker** with default backpressure configuration
2. **Connect multiple publishers** sending at different rates to the same topic
3. **Connect slow consumers** that process messages slower than the publication rate
4. **Observe lag buildup** in monitoring metrics and verify warning thresholds trigger
5. **Verify throttling activation** - faster publishers should experience delays
6. **Speed up consumers** and verify that throttling releases automatically
7. **Test priority publishers** - high-priority publishers should be throttled last

Expected monitoring output should show gradual lag increases, throttling activation at configured thresholds, and lag reduction after throttling takes effect. Publishers should receive backpressure signals (like delayed acknowledgments or explicit rate limit responses) rather than connection failures.

Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Throttling never activates despite slow consumers	Lag thresholds set too high	Check lagMonitor.GetLagMeasurement() output	Lower warning/critical thresholds in config
System thrashes between throttled/normal states	Insufficient hysteresis in threshold logic	Monitor trend analysis and threshold crossings	Add larger gap between activation/deactivation thresholds
High-priority publishers get throttled unnecessarily	Priority system not working correctly	Check publisher priority assignments	Verify priority levels in PublisherState
Throttling continues after consumers catch up	Release conditions too strict	Monitor lag improvements and throttle release logic	Implement gradual throttling release
Cascading failures when backpressure activates	Upstream systems don't handle throttling gracefully	Check error rates and retry patterns in upstream services	Implement circuit breaker patterns in publishers

Dead Letter Queue Management

> **Milestone(s):** Milestone 4 (Dead Letter Queue & Monitoring) — implements dead letter queues for poison messages and provides replay capabilities

Mental Model: Undeliverable Mail Office

Think of the **Dead Letter Queue (DLQ)** as the postal service's undeliverable mail office — a specialized facility that handles packages and letters that cannot be delivered to their intended recipients. When the mail carrier makes multiple delivery attempts but finds no one home, or when the address is invalid, or when the package is damaged beyond recognition, it doesn't just disappear. Instead, it goes to a central facility where postal workers can inspect it, attempt to determine what went wrong, and potentially try alternative delivery methods.

In our message queue system, some messages become **poison messages** — they consistently cause processing failures in consumer applications. Just like a package with a damaged address label, these messages can't be successfully "delivered" (processed) no matter how many times we try. Without a DLQ, these poison messages would endlessly cycle through the system, consuming resources and potentially blocking the processing of healthy messages behind them due to **head-of-line blocking**.

The DLQ serves three critical functions, mirroring the postal undeliverable mail office:

1. **Quarantine**: Remove problematic messages from the main processing flow, preventing them from disrupting healthy message processing
2. **Investigation**: Provide tools to inspect failed messages, understand failure patterns, and diagnose application bugs
3. **Recovery**: Enable selective replay of messages back into the main processing flow after fixes are applied

This approach transforms what would be an endless failure loop into a manageable operational process. Development teams can fix bugs in their consumer applications, then replay the affected messages from the DLQ, ensuring no data is permanently lost while maintaining system stability.

DLQ Manager Interface

The `DLQManager` component serves as the central authority for handling poison messages and providing administrative capabilities for message inspection and replay. It integrates closely with the `AckTracker` component to receive messages that have exceeded their retry limits, and coordinates with the `TopicManager` for message replay operations.

Method Name	Parameters	Returns	Description
`SendToDLQ`	`msg *Message, reason string, deliveryHistory []DeliveryAttempt`	`error`	Moves a poison message to the DLQ with failure context and complete delivery history
`InspectMessage`	`dlqMessageID string`	`*DLQMessage, error`	Retrieves a specific DLQ message with all metadata for administrative inspection
`ListMessages`	`topicFilter string, limit int, offset int`	`[]*DLQMessage, int, error`	Returns paginated list of DLQ messages with optional topic filtering and total count
`ReplayMessage`	`dlqMessageID string, targetTopic string`	`error`	Replays a single DLQ message back to specified topic for reprocessing
`ReplayBatch`	`messageIDs []string, targetTopic string`	`*ReplayResult, error`	Replays multiple DLQ messages atomically with success/failure tracking
`GetStatistics`	`topicFilter string`	`*DLQStatistics, error`	Returns DLQ metrics including message counts, failure patterns, and retention status
`PurgeMessages`	`criteria *PurgeFilter`	`*PurgeResult, error`	Permanently removes DLQ messages matching age, topic, or failure reason criteria
`CreateReplayPlan`	`criteria *ReplayFilter`	`*ReplayPlan, error`	Analyzes DLQ messages and creates ordered replay plan with dependency resolution

The `DLQManager` maintains several key data structures to support these operations effectively:

Type	Fields	Description
`DLQMessage`	`ID string, OriginalMessage *Message, FailureReason string, DeliveryHistory []DeliveryAttempt, ReceivedAt time.Time, LastAttempt time.Time, FailureCount int, OriginalTopic string, SourceConsumerGroup string, RetentionUntil time.Time`	Complete DLQ message with failure context and metadata for inspection and replay
`ReplayResult`	`TotalRequested int, SuccessfulReplays int, FailedReplays []ReplayFailure, ReplayID string, StartedAt time.Time, CompletedAt *time.Time`	Results of batch replay operation with detailed success/failure tracking
`ReplayFailure`	`MessageID string, FailureReason string, RetryableError bool`	Individual message replay failure with categorization for retry decisions
`DLQStatistics`	`TotalMessages int64, MessagesByTopic map[string]int64, MessagesByFailureReason map[string]int64, OldestMessage *time.Time, DiskSpaceUsed int64, RetentionExpiredCount int64`	Comprehensive DLQ metrics for monitoring and capacity planning
`PurgeFilter`	`TopicPattern string, OlderThan time.Time, FailureReasonPattern string, MaxCount int, DryRun bool`	Criteria for selective DLQ message removal with safety mechanisms
`ReplayFilter`	`TopicPattern string, NewerThan time.Time, OlderThan time.Time, FailureReasonPattern string, ConsumerGroupFilter string, OrderBy ReplayOrdering`	Criteria for DLQ message replay with ordering and filtering options
`ReplayPlan`	`Messages []*DLQMessage, EstimatedDuration time.Duration, Dependencies []ReplayDependency, ReplayOrder ReplayOrdering, Warnings []string`	Analyzed replay plan with ordering, dependencies, and potential issues identified

> The critical design insight for DLQ management is **temporal safety** — once a message enters the DLQ, it must remain stable and inspectable until explicitly removed. Unlike the main message queues where messages flow continuously, DLQ messages are persistent entities that support repeated inspection and analysis.

The `DLQManager` implements **retention-based lifecycle management** where messages automatically expire from the DLQ after a configurable period. This prevents unbounded growth while ensuring sufficient time for investigation and replay operations.

Message Replay Algorithm

Message replay is the most complex operation in DLQ management because it must carefully reintroduce potentially problematic messages back into the active processing flow while preserving system stability and message ordering guarantees.

The replay algorithm operates in several distinct phases to ensure safe and consistent operation:

Phase 1: Validation and Dependency Analysis

1. The algorithm begins by retrieving all requested DLQ messages and validating their current state
2. It checks that target topics exist and are active, creating them if necessary according to the broker's auto-creation policy
3. It analyzes message timestamps and topic relationships to identify ordering dependencies between messages
4. It constructs a **dependency graph** where messages that must maintain relative ordering are linked
5. It validates that replay operations won't violate any topic-level constraints such as maximum message size or retention policies
6. It estimates the replay duration based on current topic throughput and consumer group performance metrics

Phase 2: Replay Plan Creation

1. The algorithm topologically sorts the dependency graph to determine the optimal replay order
2. It groups messages into **replay batches** based on topic, timestamp proximity, and dependency constraints
3. It calculates appropriate delays between batches to avoid overwhelming consumer groups with sudden traffic spikes
4. It identifies potential **replay conflicts** where messages might be processed out of order due to consumer group rebalancing
5. It creates checkpoints within the replay plan to enable recovery if the replay operation itself fails partway through
6. It generates warnings for administrators about potential issues such as large time gaps or high-frequency message bursts

Phase 3: Atomic Replay Execution

1. The algorithm creates a unique **replay transaction ID** and logs the complete replay plan to the WAL for durability
2. It begins systematic message replay, processing one batch at a time according to the calculated schedule
3. For each message, it creates a **new message instance** with a fresh ID while preserving original payload and headers
4. It adds replay metadata to message headers including original DLQ entry time, failure reason, and replay transaction ID
5. It publishes each message through the normal `TopicManager.PublishMessage` flow to ensure consistent handling
6. It monitors consumer group health during replay, pausing if lag increases beyond safe thresholds
7. It updates replay progress in the WAL and provides real-time status through the monitoring API

Phase 4: Cleanup and Verification

1. After successful replay of each batch, the algorithm marks corresponding DLQ messages as **replay-pending**
2. It waits for a configurable verification period to confirm that replayed messages are being processed successfully
3. It monitors consumer group error rates and message acknowledgment patterns during the verification window
4. If verification succeeds, it permanently removes the replayed messages from the DLQ and logs completion
5. If verification fails (high error rates or consumer crashes), it can automatically abort remaining batches
6. It generates a comprehensive **replay report** with success metrics, error summaries, and recommendations

The algorithm implements several critical **safety mechanisms** to prevent replay operations from destabilizing the system:

- **Rate limiting**: Replay speed is automatically throttled based on current consumer lag and system load
- **Circuit breaker**: If consumer error rates spike during replay, the operation automatically pauses
- **Rollback capability**: Replay operations can be aborted, with already-replayed messages marked for re-DLQ if necessary
- **Progress persistence**: Replay state is continuously saved to WAL, enabling recovery if the broker crashes during replay

> **Critical Insight**: Message replay is not just about moving data — it's about **safely reintroducing potentially problematic messages** into a live system. The algorithm must assume that these messages caused failures for good reasons and take extensive precautions to prevent those failures from recurring.

DLQ Architecture Decisions

The DLQ implementation involves several critical architectural decisions that significantly impact system behavior, operational complexity, and data safety guarantees.

> **Decision: DLQ Storage Strategy**
>
> - **Context**: DLQ messages require long-term storage with different access patterns than normal messages — they're written once, read multiple times for inspection, and occasionally replayed
> - **Options Considered**:
>   - **Separate DLQ topics**: Store DLQ messages as regular messages in special topics
>   - **Dedicated DLQ database**: Use separate database with query capabilities
>   - **Extended WAL records**: Store DLQ messages as special WAL record types
> - **Decision**: Extended WAL records with separate DLQ index files
> - **Rationale**: WAL integration ensures DLQ messages have the same durability guarantees as normal messages, while separate index files enable efficient querying without full WAL scans. This approach reuses existing persistence infrastructure while providing the query capabilities needed for DLQ management.
> - **Consequences**: DLQ operations benefit from WAL durability and crash recovery, but require additional indexing logic and disk space for query optimization

Storage Option	Durability	Query Performance	Implementation Complexity	Storage Efficiency
DLQ Topics	High (existing WAL)	Poor (sequential scan)	Low (reuse existing)	Poor (topic overhead)
Separate Database	Medium (depends on DB)	Excellent (SQL queries)	High (new dependency)	Excellent (optimized schema)
Extended WAL	High (WAL guarantees)	Good (with indexing)	Medium (indexing layer)	Good (compact records)

> **Decision: Message Replay Ordering Strategy**
>
> - **Context**: When replaying multiple DLQ messages, their processing order can affect application correctness, especially for messages that modify the same data entities
> - **Options Considered**:
>   - **Original timestamp order**: Replay messages in the order they were originally published
>   - **DLQ entry order**: Replay messages in the order they entered the DLQ
>   - **Administrative control**: Allow operators to specify custom replay ordering
> - **Decision**: Original timestamp order with administrative override capability
> - **Rationale**: Original timestamp ordering preserves the intended message sequence from the producer's perspective, which is most likely to maintain application correctness. Administrative override handles special cases where operators have domain-specific knowledge about safe replay ordering.
> - **Consequences**: Requires storing original message timestamps with DLQ entries, and replay operations must sort messages before processing, but provides the highest probability of maintaining application correctness

Ordering Strategy	Correctness Guarantee	Operational Complexity	Performance Impact	Flexibility
Original Timestamp	High (preserves intent)	Low (automatic)	Medium (sorting required)	Low (fixed strategy)
DLQ Entry Order	Low (failure-driven)	Low (automatic)	High (no sorting)	Low (fixed strategy)
Administrative Control	Variable (depends on operator)	High (manual decisions)	Low (direct control)	High (full control)

Decision: DLQ Retention and Cleanup Policy

- **Context**: DLQ messages must be retained long enough for investigation and replay, but indefinite retention leads to unbounded storage growth
- **Options Considered**:
  - **Time-based retention**: Automatically purge messages older than configured age
  - **Count-based retention**: Keep only the N most recent DLQ messages per topic
  - **Manual-only cleanup**: Require explicit administrative action to remove DLQ messages
- **Decision**: Time-based retention with manual override and configurable policies per topic
- **Rationale**: Time-based retention provides predictable storage usage and aligns with typical incident response timelines (e.g., 30-90 days). Per-topic configuration allows different retention policies for different business criticality levels. Manual override ensures important messages can be preserved beyond automatic expiration.
- **Consequences**: Requires background cleanup processes and careful configuration management, but provides predictable resource usage and operational flexibility

Retention Strategy	Storage Predictability	Operational Burden	Data Loss Risk	Configuration Complexity
Time-based	High (bounded by age)	Low (automatic)	Medium (time-driven)	Medium (per-topic config)
Count-based	High (bounded by count)	Low (automatic)	High (recent-biased)	Low (simple limits)
Manual-only	Low (unbounded growth)	High (manual tracking)	Low (explicit control)	Low (no automation)

Decision: Replay Transaction Atomicity

- **Context**: When replaying multiple related DLQ messages, partial failures can leave the system in an inconsistent state where some messages were replayed and others weren't
- **Options Considered**:
  - **Best-effort replay**: Replay as many messages as possible, report failures
  - **All-or-nothing replay**: Replay entire batch atomically or fail completely
  - **Checkpoint-based replay**: Support resumable replay with intermediate checkpoints
- **Decision**: Checkpoint-based replay with configurable checkpoint frequency
- **Rationale**: All-or-nothing replay is too rigid for large batches (one bad message blocks thousands of good ones), while best-effort provides no consistency guarantees. Checkpoint-based replay balances consistency with practical recovery — if replay fails partway through, it can resume from the last successful checkpoint rather than starting over.
- **Consequences**: Requires sophisticated replay state tracking and WAL integration, but enables reliable replay of large DLQ batches with resumable operation

Atomicity Strategy	Consistency Guarantee	Recovery Capability	Implementation Complexity	Operational Flexibility
Best-effort	Low (partial success)	Poor (manual tracking)	Low (simple iteration)	High (individual control)
All-or-nothing	High (batch-level)	Poor (full restart)	Medium (transaction logic)	Low (batch-only)
Checkpoint-based	Medium (checkpoint-level)	Excellent (resumable)	High (state tracking)	High (resumable batches)

Common DLQ Pitfalls

The Dead Letter Queue implementation presents several subtle challenges that frequently trap developers. Understanding these pitfalls early can prevent significant operational issues and data consistency problems.

⚠️ Pitfall: Out-of-Order Replay Creates Data Inconsistency

A common mistake is replaying DLQ messages without considering their original ordering relationships. Consider an e-commerce scenario where the DLQ contains messages for the same order: "order_created", "payment_processed", "inventory_reserved", and "order_shipped". If these

messages are replayed in DLQ entry order rather than their original timestamp order, the consumer might process "order_shipped" before "payment_processed", leading to invalid business state.

This happens because developers often assume DLQ messages are independent when they actually represent a sequence of related state changes. The failure occurs when replay logic sorts by DLQ entry time rather than original message timestamp, or when batch replay doesn't consider cross-message dependencies.

Prevention Strategy: Always implement **dependency analysis** in replay logic. Before replaying any batch of messages, analyze their timestamps and headers to identify potential ordering relationships. For messages affecting the same entity (identified by customer ID, order ID, etc.), maintain their original temporal ordering. Consider implementing a "replay dry-run" mode that validates ordering constraints before executing the actual replay.

⚠️ Pitfall: DLQ Infinite Loops from Systematic Consumer Bugs

A subtle but dangerous pitfall occurs when a consumer bug affects all messages of a certain pattern, but the replay logic doesn't detect this systematic failure. For example, if a consumer crashes on all messages containing unicode characters, replaying these messages without fixing the bug creates an infinite cycle: DLQ → replay → failure → DLQ.

This typically happens when the DLQ system lacks **replay success monitoring**. Developers implement replay functionality but don't track whether replayed messages actually get processed successfully. The system dutifully replays messages that immediately fail again, creating resource waste and masking the underlying bug.

Prevention Strategy: Implement **post-replay verification** with configurable monitoring windows. After replaying messages, track their acknowledgment rates and consumer error patterns for a specified period (e.g., 1-2 hours). If replayed messages show higher error rates than baseline, automatically flag them for administrative attention and potentially re-DLQ them. Include replay metadata in message headers so consumer logs can correlate failures with replay operations.

⚠️ Pitfall: DLQ Storage Growing Without Operational Alerting

Many implementations focus on the functional aspects of DLQ management but neglect operational monitoring, leading to situations where DLQ storage grows unbounded without anyone noticing until disk space alerts fire. This is particularly problematic because DLQ growth often indicates systematic application problems that require immediate attention.

The failure typically occurs when DLQ implementation lacks proper **capacity monitoring and alerting**. Developers implement retention policies but don't alert when DLQ message rates exceed normal thresholds, or when certain error patterns dominate the DLQ contents.

Prevention Strategy: Implement comprehensive **DLQ health monitoring** with multiple alert thresholds. Monitor DLQ message ingestion rates, storage growth trends, and failure pattern distributions. Set up alerts for: DLQ message rate exceeding baseline by X%, storage growth approaching retention limits, and specific failure reasons dominating DLQ contents. Include DLQ metrics in standard operational dashboards so growth trends are visible during routine monitoring.

⚠️ Pitfall: Replay Operations Overwhelming Active Consumers

A frequently overlooked issue is that replaying large batches of DLQ messages can overwhelm active consumer groups, especially if the messages were originally DLQ'd due to high processing complexity. Developers often implement replay as a simple "republish all messages" operation without considering the additional load on downstream systems.

This creates a **thundering herd** scenario where replay introduces a sudden traffic spike that causes currently-healthy consumers to start failing, potentially creating a cascade where new messages enter the DLQ faster than replay can process them.

Prevention Strategy: Implement **adaptive replay throttling** that monitors consumer group health during replay operations. Before starting replay, establish baseline metrics for consumer lag, error rates, and processing times. During replay, continuously monitor these metrics and automatically throttle replay speed if consumer performance degrades beyond safe thresholds. Consider implementing "replay windows" during off-peak hours to minimize impact on production traffic.

⚠️ Pitfall: Missing Replay Transaction Durability

A subtle but critical issue occurs when replay operations themselves aren't properly persisted, leading to situations where a broker crash during replay leaves the system in an ambiguous state. Some DLQ messages may have been replayed while others haven't, but there's no record of replay progress.

This happens when developers implement replay as an in-memory operation without proper **WAL integration**. The replay logic may track progress in memory but doesn't persist intermediate state, so a crash requires manual investigation to determine which messages were successfully replayed.

Prevention Strategy: Treat replay operations as **durable transactions** with full WAL integration. Before starting any replay operation, write a "replay begin" record to the WAL with the complete list of message IDs and target parameters. As replay progresses, write checkpoint records indicating successful completion of message batches. On broker startup, check for incomplete replay transactions and either resume them or mark them as failed for administrative attention. This ensures replay operations have the same durability guarantees as regular message processing.

Implementation Guidance

The DLQ Manager requires careful integration with existing components while maintaining high performance for administrative operations that may not be on the critical path but still need to be responsive.

A. Technology Recommendations:

Component	Simple Option	Advanced Option
DLQ Storage	SQLite embedded database	PostgreSQL with JSON columns
Query Interface	REST API with JSON	GraphQL for complex queries
Replay Engine	In-memory batch processing	Stream processing with Apache Kafka Connect
Administrative UI	Command-line tools	Web dashboard with React/Vue
Retention Management	Cron jobs with file deletion	Background worker with exponential backoff

B. Recommended File/Module Structure:

```
internal/dlq/
  dlq_manager.go           ← main DLQ operations and interface
  dlq_storage.go           ← persistence layer for DLQ messages
  dlq_replay.go            ← message replay logic and algorithms
  dlq_retention.go         ← background retention and cleanup
  dlq_metrics.go           ← statistics collection and reporting
  dlq_types.go             ← data structures and constants
  dlq_manager_test.go      ← comprehensive test suite
  storage/
    sqlite_storage.go      ← SQLite implementation
    memory_storage.go      ← in-memory implementation for testing
  replay/
    plan_analyzer.go       ← dependency analysis and ordering
    batch_processor.go     ← replay execution engine
    progress_tracker.go    ← replay state persistence
```

C. Infrastructure Starter Code:

```go
// Package dlq provides dead letter queue management for poison messages
package dlq

import (
    "context"
    "database/sql"
    "encoding/json"
    "fmt"
    "sort"
    "sync"
    "time"
    _ "github.com/mattn/go-sqlite3" // SQLite driver
)

// DLQStorage provides persistence operations for DLQ messages
type DLQStorage interface {
    Store(ctx context.Context, dlqMsg *DLQMessage) error
    Retrieve(ctx context.Context, messageID string) (*DLQMessage, error)
    List(ctx context.Context, filter *DLQFilter) ([]*DLQMessage, int, error)
    Delete(ctx context.Context, messageID string) error
    DeleteBatch(ctx context.Context, messageIDs []string) error
    GetStatistics(ctx context.Context, topicFilter string) (*DLQStatistics, error)
    Close() error
}

// SQLiteStorage implements DLQStorage using SQLite
type SQLiteStorage struct {
    db     *sql.DB
    mutex  sync.RWMutex
}

// NewSQLiteStorage creates a new SQLite-based DLQ storage
func NewSQLiteStorage(dbPath string) (*SQLiteStorage, error) {
    db, err := sql.Open("sqlite3", dbPath+"?_journal=WAL&_timeout=5000")
    if err != nil {
        return nil, fmt.Errorf("failed to open SQLite database: %w", err)
    }
```

```
    storage := &SQLiteStorage{db: db}
    if err := storage.initSchema(); err != nil {
        db.Close()
        return nil, fmt.Errorf("failed to initialize schema: %w", err)
    }

    return storage, nil
}

func (s *SQLiteStorage) initSchema() error {
    schema := `
    CREATE TABLE IF NOT EXISTS dlq_messages (
        id TEXT PRIMARY KEY,
        original_message_json TEXT NOT NULL,
        failure_reason TEXT NOT NULL,
        delivery_history_json TEXT NOT NULL,
        received_at INTEGER NOT NULL,
        last_attempt INTEGER NOT NULL,
        failure_count INTEGER NOT NULL,
        original_topic TEXT NOT NULL,
        source_consumer_group TEXT,
        retention_until INTEGER NOT NULL,
        created_at INTEGER DEFAULT (strftime('%s', 'now'))
    );

    CREATE INDEX IF NOT EXISTS idx_dlq_topic ON dlq_messages(original_topic);
    CREATE INDEX IF NOT EXISTS idx_dlq_received ON dlq_messages(received_at);
    CREATE INDEX IF NOT EXISTS idx_dlq_retention ON dlq_messages(retention_until);
    `

    _, err := s.db.Exec(schema)
    return err
}

// ReplayProgressTracker manages replay operation state persistence
type ReplayProgressTracker struct {
```

```
    walLogger WALLogger
    mutex     sync.RWMutex
    activeReplays map[string]*ReplayProgress
}

// ReplayProgress tracks the state of an ongoing replay operation
type ReplayProgress struct {
    ReplayID        string                `json:"replay_id"`
    StartedAt       time.Time             `json:"started_at"`
    TotalMessages   int                   `json:"total_messages"`
    CompletedBatches int                  `json:"completed_batches"`
    LastCheckpoint  time.Time             `json:"last_checkpoint"`
    MessageIDs      []string              `json:"message_ids"`
    TargetTopic     string                `json:"target_topic"`
    Status          ReplayStatus          `json:"status"`
    ErrorMessage    string                `json:"error_message,omitempty"`
}

type ReplayStatus string

const (
    ReplayStatusPending   ReplayStatus = "pending"
    ReplayStatusRunning   ReplayStatus = "running"
    ReplayStatusCompleted ReplayStatus = "completed"
    ReplayStatusFailed    ReplayStatus = "failed"
    ReplayStatusAborted   ReplayStatus = "aborted"
)

// NewReplayProgressTracker creates a new replay progress tracker
func NewReplayProgressTracker(walLogger WALLogger) *ReplayProgressTracker {
    return &ReplayProgressTracker{
        walLogger:     walLogger,
        activeReplays: make(map[string]*ReplayProgress),
    }
}
```

D. Core Logic Skeleton Code:

```go
// DLQManager handles poison messages and replay operations
type DLQManager struct {
    storage         DLQStorage
    topicManager    TopicManagerInterface
    ackTracker      AckTrackerInterface
    progressTracker *ReplayProgressTracker
    retentionPolicy *DLQRetentionPolicy
    metrics         *DLQMetrics
    config          *DLQConfig
    shutdownCh      chan struct{}
    wg              sync.WaitGroup
}

// SendToDLQ moves a poison message to the dead letter queue
func (d *DLQManager) SendToDLQ(msg *Message, reason string, deliveryHistory []DeliveryAttempt) error {
    // TODO 1: Create DLQMessage with current timestamp and failure context
    // TODO 2: Calculate retention expiration based on policy and message age
    // TODO 3: Store message to DLQ persistence layer with atomic transaction
    // TODO 4: Remove original message from pending acknowledgment tracking
    // TODO 5: Update DLQ metrics (total count, failure reason counters)
    // TODO 6: Log DLQ event to WAL for audit trail
    // Hint: Use d.storage.Store() within a transaction context
}

// ReplayMessage replays a single DLQ message back to active processing
func (d *DLQManager) ReplayMessage(dlqMessageID string, targetTopic string) error {
    // TODO 1: Retrieve DLQ message by ID and validate it exists
    // TODO 2: Create new Message instance with fresh ID but preserve payload/headers
    // TODO 3: Add replay metadata to headers (replay_id, original_dlq_time, failure_reason)
    // TODO 4: Validate target topic exists or create it according to auto-creation policy
    // TODO 5: Publish message through TopicManager.PublishMessage for normal processing
    // TODO 6: Mark DLQ message as "replay-pending" to track verification window
    // TODO 7: Schedule verification check after configurable delay to confirm processing
    // Hint: Use d.topicManager.PublishMessage() and d.progressTracker.StartReplay()
}

// ReplayBatch replays multiple DLQ messages with dependency analysis and ordering
```

```go
func (d *DLQManager) ReplayBatch(messageIDs []string, targetTopic string) (*ReplayResult, error) {
    // TODO 1: Retrieve all requested DLQ messages and validate they exist
    // TODO 2: Analyze message timestamps to identify ordering dependencies
    // TODO 3: Create dependency graph and topologically sort for safe replay order
    // TODO 4: Group messages into batches based on timestamp proximity and topic
    // TODO 5: Create replay transaction with unique ID and persist plan to WAL
    // TODO 6: Execute batches sequentially with throttling based on consumer lag
    // TODO 7: Monitor consumer health during replay and pause if errors spike
    // TODO 8: Update progress checkpoints after each successful batch
    // TODO 9: Generate comprehensive replay report with success/failure details
    // Hint: Use d.analyzeDependencies() and d.executeReplayPlan() helper methods
}

// CreateReplayPlan analyzes DLQ messages and creates ordered replay strategy
func (d *DLQManager) CreateReplayPlan(criteria *ReplayFilter) (*ReplayPlan, error) {
    // TODO 1: Query DLQ storage with filter criteria to get candidate messages
    // TODO 2: Sort messages by original timestamp to preserve intended ordering
    // TODO 3: Identify message clusters that affect same business entities
    // TODO 4: Calculate optimal batch sizes based on consumer group capacity
    // TODO 5: Estimate replay duration based on current topic throughput metrics
    // TODO 6: Generate warnings for potential issues (large gaps, high frequency bursts)
    // TODO 7: Create dependency map showing which messages must maintain relative order
    // Hint: Use d.storage.List() and d.analyzeMessageRelationships() helper
}

// PurgeMessages permanently removes DLQ messages matching filter criteria
func (d *DLQManager) PurgeMessages(criteria *PurgeFilter) (*PurgeResult, error) {
    // TODO 1: Validate purge criteria and apply safety checks (dry-run mode)
    // TODO 2: Query DLQ storage to identify messages matching filter
    // TODO 3: Check if any matched messages are involved in active replay operations
    // TODO 4: Create purge transaction and log operation to WAL for audit trail
    // TODO 5: Delete matched messages in batches to avoid long-running transactions
    // TODO 6: Update metrics to reflect removed message counts and freed storage
    // TODO 7: Generate purge report with removed message details and statistics
    // Hint: Use d.storage.DeleteBatch() with transaction boundaries
}
```

```
// runRetentionCleanup performs background cleanup of expired DLQ messages
func (d *DLQManager) runRetentionCleanup() {
    // TODO 1: Query for messages where retention_until < current timestamp
    // TODO 2: Group expired messages into batches for efficient deletion
    // TODO 3: Check that expired messages aren't part of active replay operations
    // TODO 4: Delete expired batches and update storage metrics
    // TODO 5: Log retention cleanup results for operational monitoring
    // TODO 6: Sleep for configured interval before next cleanup cycle
    // Hint: Run in separate goroutine started in NewDLQManager()
}
```

E. Language-Specific Hints:

- **SQLite Integration**: Use `database/sql` with SQLite driver for DLQ persistence. Enable WAL mode (`?_journal=WAL`) for better concurrent read performance during administrative queries.
- **JSON Serialization**: Store complex fields like `DeliveryHistory` as JSON using `encoding/json`. Create custom `MarshalJSON` / `UnmarshalJSON` methods for complex types.
- **Concurrent Safety**: Use `sync.RWMutex` for protecting DLQ metrics and active replay tracking. Readers (queries) can run concurrently while writes (replay operations) get exclusive access.
- **Context Integration**: Pass `context.Context` through all DLQ operations to enable cancellation of long-running operations like large batch replays.
- **Background Workers**: Use `sync.WaitGroup` to coordinate graceful shutdown of retention cleanup goroutines.

F. Milestone Checkpoint:

After implementing the DLQ Manager, verify the following behavior:

1. **DLQ Message Storage**: Run `go test ./internal/dlq -run TestSendToDLQ` to verify messages are correctly stored with failure context and metadata.
2. **Message Replay**: Test single message replay with:

```bash
curl -X POST localhost:8080/admin/dlq/replay \
     -H "Content-Type: application/json" \
     -d '{"message_id":"dlq-12345","target_topic":"orders"}'
```

Verify the message appears in the target topic with replay headers.

3. **Batch Replay with Ordering**: Create multiple related DLQ messages with different timestamps and verify replay maintains original ordering.
4. **Retention Cleanup**: Wait for retention period to expire and verify automatic cleanup removes old DLQ messages while preserving recent ones.

G. Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
DLQ messages not appearing	Storage transaction failure	Check SQLite logs for constraint violations	Verify schema matches DLQMessage struct fields
Replay creates duplicate messages	Message ID collision during replay	Check logs for duplicate ID errors	Generate fresh UUIDs for replayed messages
Out-of-order replay processing	Missing dependency analysis	Compare original vs replay timestamps	Implement topological sorting in replay plan
Retention cleanup not working	Background goroutine panic	Check for unhandled errors in cleanup loop	Add proper error handling and restart logic
Replay operations hang	Consumer group overload	Monitor consumer lag during replay	Implement adaptive throttling based on lag metrics

Monitoring and Observability

> **Milestone(s):** Milestone 4 (Dead Letter Queue & Monitoring) — provides metrics, health checks, and administrative APIs that enable production monitoring and operational management

Mental Model: System Dashboard

Think of monitoring a message queue like the dashboard in your car. Just as your car's dashboard shows engine RPM, fuel level, temperature, and warning lights, a message broker needs a comprehensive dashboard showing queue depths, consumer lag, error rates, and system health. When your car's temperature gauge enters the red zone, you know to pull over immediately — similarly, when consumer lag exceeds thresholds, the monitoring system should trigger alerts and potentially activate backpressure mechanisms.

The car dashboard analogy extends further: just as your dashboard aggregates data from dozens of sensors throughout the vehicle (engine sensors, fuel sensors, brake sensors), our monitoring system aggregates metrics from all broker components — the Topic Manager reports message counts, the Consumer Group Coordinator reports rebalancing events, the Acknowledgment Tracker reports delivery success rates, and the Persistence Layer reports disk usage. Like how modern cars have both analog gauges for at-a-glance status and digital displays for detailed diagnostics, our monitoring provides both real-time metrics for operators and detailed historical data for troubleshooting.

The dashboard metaphor also highlights the importance of **hierarchical information display**. Your car shows critical alerts (check engine light) prominently, important status (speed, fuel) in primary gauges, and detailed diagnostics (tire pressure, oil life) in secondary menus. Our monitoring system follows this pattern: critical alerts like DLQ growth appear in primary dashboards, routine metrics like throughput appear in operational views, and detailed diagnostics like per-consumer delivery times appear in detailed drill-down interfaces.

Metrics Collection

The monitoring system implements comprehensive metrics collection across all broker components, providing real-time visibility into system performance and health. Unlike simple logging that records individual events, metrics collection aggregates data over time windows to reveal patterns, trends, and anomalies that individual events might not expose.

The metrics collection architecture centers around a **push-based model** where each component actively reports its metrics to a central collector rather than waiting for external polling. This approach ensures low-latency metric updates and reduces the coordination overhead that pull-based systems require. Components emit metrics at regular intervals (typically every 1-5 seconds) and immediately upon significant events like consumer failures or DLQ message arrivals.

Core Metrics Categories:

Metric Category	Examples	Update Frequency	Storage Duration
Queue Depth Metrics	Messages per topic, pending ACK count, DLQ size	Every 1 second	7 days detailed, 90 days aggregated
Throughput Metrics	Messages/second published, consumed, acknowledged	Every 5 seconds	24 hours detailed, 30 days aggregated
Consumer Lag Metrics	Time from publish to consumption, backlog size	Every 1 second	24 hours detailed, 7 days aggregated
Error Rate Metrics	NACK percentage, timeout rate, DLQ arrival rate	Every 5 seconds	30 days detailed, 1 year aggregated
System Resource Metrics	Memory usage, connection count, disk space	Every 10 seconds	24 hours detailed, 30 days aggregated
Rebalancing Metrics	Rebalance frequency, duration, consumer churn	On event	90 days detailed

The **MetricsCollector** component implements a time-series database optimized for message queue metrics. Rather than using an external system like Prometheus or InfluxDB, the collector uses an embedded approach with in-memory storage for recent data and periodic persistence to disk for historical analysis. This design eliminates external dependencies while providing the specific aggregations and queries that message queue monitoring requires.

Metrics Collection Data Structures:

Field	Type	Description
MetricName	string	Hierarchical metric identifier (e.g., "topic.messages.published")
Value	float64	Numeric value at measurement time
Tags	map[string]string	Dimensional data (topic, consumer_group, node_id)
Timestamp	int64	Unix timestamp in milliseconds
MetricType	string	Counter, Gauge, Histogram, or Timer

TimeSeries Storage:

Field	Type	Description
Points	[]DataPoint	Chronological measurement points
Retention	time.Duration	How long to keep detailed data
Aggregation	AggregationPolicy	Rules for downsampling old data
Tags	map[string]string	Common tags for all points in series
LastUpdate	time.Time	Most recent data point timestamp

The metrics collection system implements **automatic aggregation** to manage storage growth while preserving important trends. Raw metrics are stored at full resolution for the first 24 hours, then aggregated into 1-minute buckets for the next 7 days, then 5-minute buckets for 30 days, and finally 1-hour buckets for long-term trend analysis. This hierarchical storage approach balances storage efficiency with the ability to drill down into recent performance issues.

Consumer Lag Calculation deserves special attention as it's the most critical metric for queue health monitoring. Lag is measured in two dimensions: **time lag** (how long ago was the oldest unprocessed message published) and **message lag** (how many messages are waiting for processing). The system calculates lag for each consumer group independently and aggregates across groups for topic-level lag metrics.

The lag calculation algorithm operates as follows:

1. For each consumer group, identify the oldest unacknowledged message assigned to any group member
2. Calculate time lag as the difference between current time and that message's timestamp
3. Calculate message lag by counting all messages published to the topic since that timestamp
4. Update lag metrics every second and trigger alerts if lag exceeds configured thresholds
5. Maintain separate lag calculations for regular queues and dead letter queues

Throughput Measurement uses a sliding window approach to provide smooth, responsive metrics that filter out temporary spikes while accurately reflecting sustained performance changes. The system maintains multiple time windows (1 minute, 5 minutes, 15 minutes) and calculates rates for each window using exponentially weighted moving averages that emphasize recent data while smoothing short-term fluctuations.

Health Check Algorithm

The health check system provides continuous monitoring of consumer liveness and automatic cleanup of failed or disconnected consumers. Unlike simple TCP connection monitoring that only detects network failures, the health check algorithm monitors **logical health** by tracking consumer heartbeats, processing performance, and acknowledgment patterns.

Consumer Heartbeat Tracking forms the foundation of health monitoring. Each consumer must send a heartbeat message to the broker at regular intervals (default: every 30 seconds) containing its consumer ID, current processing status, and performance metrics. The broker maintains a heartbeat registry that tracks the last heartbeat time for each consumer and marks consumers as unhealthy if they miss consecutive heartbeat windows.

HeartbeatTracker Data Structures:

Field	Type	Description
ConsumerHeartbeats	map[string]*ConsumerHeartbeat	Per-consumer heartbeat state
HeartbeatTimeout	time.Duration	Maximum time between heartbeats (default: 60s)
CheckInterval	time.Duration	How often to scan for timeouts (default: 10s)
UnhealthyConsumers	map[string]time.Time	Consumers marked unhealthy with timestamp
CleanupGracePeriod	time.Duration	Delay before removing unhealthy consumers

ConsumerHeartbeat State:

Field	Type	Description
ConsumerID	string	Unique consumer identifier
LastHeartbeat	time.Time	Timestamp of most recent heartbeat
ConsecutiveMisses	int	Count of missed heartbeat intervals
ProcessingStatus	string	Current consumer state (idle, processing, error)
PendingMessageCount	int	Number of unacknowledged messages
AverageProcessingTime	time.Duration	Recent processing performance
ErrorRate	float64	Percentage of messages resulting in NACK

The health check algorithm runs continuously in the background and performs the following steps:

1. **Heartbeat Scanning**: Every 10 seconds, scan all registered consumers and identify those whose last heartbeat exceeds the timeout threshold
2. **Grace Period Management**: When a consumer first misses its heartbeat, mark it as "suspected unhealthy" but don't immediately take action — network blips or temporary processing delays can cause false positives
3. **Confirmation Phase**: If a consumer misses heartbeats for a full grace period (typically 2-3 timeout intervals), mark it as definitively unhealthy
4. **Message Reassignment**: Retrieve all pending messages assigned to the unhealthy consumer and return them to the topic's unassigned queue for redelivery
5. **Group Rebalancing**: If the unhealthy consumer was part of a consumer group, trigger an immediate rebalancing to redistribute its assigned partitions
6. **Connection Cleanup**: Close the consumer's network connection and remove it from all subscription lists
7. **Metrics Update**: Record the consumer failure in monitoring metrics and update group membership counts

Advanced Health Indicators go beyond simple heartbeat monitoring to detect consumers that are technically alive but functionally impaired. The system tracks several **performance-based health indicators**:

- **Processing Velocity**: Consumers that consistently take longer than expected to process messages may indicate resource constraints or application bugs
- **Acknowledgment Patterns**: Consumers with unusually high NACK rates may be encountering systematic processing errors
- **Queue Starvation**: Consumers that request work but consistently have empty assignment queues may indicate configuration issues
- **Memory Pressure**: Consumers approaching their maximum inflight message limits may be experiencing memory or processing bottlenecks

The health check system implements **adaptive thresholds** that account for normal variations in consumer performance. Rather than using fixed timeout values, the system maintains historical performance baselines for each consumer and flags deviation beyond statistical norms. This approach reduces false positives while catching performance degradation early.

Cleanup Algorithm for Unhealthy Consumers:

Step	Action	Safety Check	Rollback Condition
1. Mark Unhealthy	Set consumer status to unhealthy	Verify consumer hasn't sent recent heartbeat	Revert if heartbeat received
2. Message Recovery	Retrieve pending messages from consumer	Ensure messages aren't duplicated	None - safe operation
3. Group Notification	Notify consumer group of member departure	Check if consumer was group member	Skip if not in group
4. Rebalancing	Trigger consumer group rebalancing	Verify group has remaining members	Skip rebalancing if group empty
5. Connection Cleanup	Close network connection to consumer	Confirm connection is still open	None - safe operation
6. Registry Cleanup	Remove consumer from all subscription lists	Verify consumer is fully unsubscribed	None - safe operation

Administrative API

The administrative API provides REST endpoints for inspecting broker state, managing configuration, and performing operational tasks. Unlike the binary wire protocol used for high-performance message operations, the administrative API uses HTTP and JSON for human-friendly interaction and integration with standard monitoring tools.

The API follows RESTful design principles with **hierarchical resource organization** that mirrors the broker's internal component structure. Resources are organized around the primary entities: topics, consumer groups, consumers, and messages. Each resource type provides standard CRUD operations where appropriate, plus specialized operations for queue-specific functionality.

API Endpoint Categories:

Category	Base Path	Purpose	Authentication Required
Topic Management	`/api/v1/topics`	Create, list, configure topics	Admin
Consumer Groups	`/api/v1/groups`	Inspect group state, trigger rebalancing	Read-only
Message Operations	`/api/v1/messages`	Inspect queues, replay from DLQ	Admin
Metrics and Health	`/api/v1/metrics`	Real-time metrics, health status	Read-only
Administrative	`/api/v1/admin`	Broker config, shutdown, maintenance	Admin

Topic Management Endpoints:

Method	Path	Parameters	Response	Description
GET	`/api/v1/topics`	limit, offset, pattern	TopicList	List all topics with basic stats
POST	`/api/v1/topics`	TopicSpec	TopicInfo	Create new topic with retention policy
GET	`/api/v1/topics/{name}`	include_messages	DetailedTopicInfo	Get topic details and optionally message samples
PUT	`/api/v1/topics/{name}/config`	RetentionPolicy	TopicInfo	Update topic retention and configuration
DELETE	`/api/v1/topics/{name}`	force	OperationResult	Delete topic and all messages (requires force=true)
GET	`/api/v1/topics/{name}/messages`	limit, since	MessageList	Retrieve recent messages for debugging
POST	`/api/v1/topics/{name}/messages`	Message	PublishResult	Publish message via HTTP (low-performance path)

Consumer Group Management:

Method	Path	Parameters	Response	Description
GET	`/api/v1/groups`	None	ConsumerGroupList	List all consumer groups with member counts
GET	`/api/v1/groups/{name}`	include_members	GroupDetails	Get group state, assignments, lag metrics
POST	`/api/v1/groups/{name}/rebalance`	reason	RebalanceResult	Force immediate rebalancing (emergency use)
GET	`/api/v1/groups/{name}/lag`	None	LagMetrics	Detailed consumer lag analysis
GET	`/api/v1/groups/{name}/members`	None	MemberList	List group members with health status
DELETE	`/api/v1/groups/{name}/members/{id}`	None	OperationResult	Forcibly remove unhealthy consumer

Monitoring and Metrics Endpoints:

Method	Path	Parameters	Response	Description
GET	`/api/v1/metrics`	category, duration	MetricsSnapshot	Current metrics across all categories
GET	`/api/v1/metrics/topics`	topic_pattern	TopicMetrics	Per-topic metrics and performance
GET	`/api/v1/metrics/consumers`	group_name	ConsumerMetrics	Per-consumer performance and health
GET	`/api/v1/health`	deep	HealthStatus	Overall broker health and component status
GET	`/api/v1/health/consumers`	group_name	ConsumerHealthList	Health status of all consumers
POST	`/api/v1/health/check`	None	HealthCheckResult	Trigger manual health check scan

The administrative API implements **comprehensive error handling** with structured error responses that provide actionable information for troubleshooting. Rather than returning generic HTTP status codes, the API returns detailed error objects that include error codes, human-readable messages, and suggested remediation steps.

API Error Response Format:

Field	Type	Description
ErrorCode	string	Machine-readable error identifier
Message	string	Human-readable error description
Details	object	Additional context about the error
Suggestions	[]string	Recommended actions to resolve the error
RequestID	string	Unique identifier for tracing the request
Timestamp	string	ISO8601 timestamp when error occurred

Authentication and Authorization for the administrative API uses a simple token-based system appropriate for operational use. The broker supports two access levels: **read-only** (can view metrics and status) and **admin** (can modify configuration and perform destructive operations). API tokens are configured in the broker's configuration file and validated on each request.

Pagination and Filtering are implemented consistently across all list endpoints to handle large-scale deployments. List operations support standard pagination parameters (`limit` , `offset`) and filtering parameters specific to each resource type. The API returns pagination metadata in response headers to enable client-side pagination controls.

Rate Limiting protects the broker from API abuse while allowing legitimate monitoring tools to function effectively. The API implements a token bucket rate limiter with different limits for read-only operations (100 requests/minute) and admin operations (10 requests/minute). Rate limiting is applied per API token to prevent one monitoring tool from affecting others.

Monitoring Architecture Decisions

The monitoring system requires several architectural decisions that balance performance, functionality, and operational requirements. These decisions significantly impact the system's ability to provide real-time insights while minimizing overhead on message processing performance.

Decision: Embedded vs External Metrics Storage

- **Context**: The monitoring system needs to store time-series metrics for operational visibility and troubleshooting, but adding external dependencies increases deployment complexity and potential failure modes.
- **Options Considered**:
  1. **Embedded time-series storage** with in-memory data and disk persistence
  2. **External Prometheus** with metrics exposition and remote storage
  3. **External InfluxDB** with direct metric shipping via HTTP API
- **Decision**: Embedded time-series storage with configurable external export
- **Rationale**: Embedded storage eliminates external dependencies for basic monitoring while providing escape hatches for advanced analytics. The message queue can function and provide operational visibility even if external monitoring infrastructure fails. Embedded storage also enables monitoring-driven backpressure decisions without network round trips.
- **Consequences**: Simpler deployment and reduced dependencies, but limited storage capacity and query capabilities compared to specialized time-series databases. Added complexity in implementing efficient time-series storage and aggregation.

Option	Pros	Cons
Embedded Storage	No dependencies, low latency, self-contained	Limited storage, basic queries, implementation complexity
External Prometheus	Industry standard, powerful queries, ecosystem	Network dependency, pull-based complexity, external failure modes
External InfluxDB	Native time-series, advanced analytics	Additional infrastructure, licensing costs, network dependency

Decision: Push vs Pull Metrics Collection

- **Context**: Components need to report metrics to the monitoring system, either by actively pushing data or by exposing data for external collection.
- **Options Considered**:
  1. **Push-based collection** where components actively send metrics to collector
  2. **Pull-based collection** where collector polls components for current metrics
  3. **Hybrid approach** with push for events and pull for state
- **Decision**: Push-based collection with configurable external exposition
- **Rationale**: Push-based collection provides lower latency updates and simpler component design since components don't need to maintain HTTP servers. Push also enables immediate metric updates on significant events rather than waiting for polling intervals. The broker can still expose pull endpoints for external monitoring tools.
- **Consequences**: Lower latency metrics and simplified component interfaces, but requires more careful design to prevent metric flooding and ensure delivery reliability.

Decision: Metric Aggregation Strategy

- **Context**: Raw metrics generate significant data volume that must be managed while preserving important trends and enabling drill-down analysis.
- **Options Considered**:
  1. **Fixed retention windows** with simple time-based purging
  2. **Hierarchical aggregation** with multiple resolution levels
  3. **Adaptive retention** based on metric importance and change rate
- **Decision**: Hierarchical aggregation with configurable retention policies
- **Rationale**: Hierarchical aggregation balances storage efficiency with analytical capability. Recent data at full resolution enables detailed troubleshooting while aggregated historical data supports trend analysis. Different metric types can use different aggregation strategies based on their importance and variability.
- **Consequences**: Efficient storage usage and flexible analysis capabilities, but increased implementation complexity and potential for aggregation artifacts in edge cases.

Decision: Health Check Algorithm Design

- **Context**: Consumer health monitoring must detect failures quickly while avoiding false positives that could cause unnecessary rebalancing and message reassignment.
- **Options Considered**:
  1. **Simple timeout-based detection** with fixed heartbeat intervals
  2. **Adaptive thresholds** based on historical performance patterns
  3. **Multi-factor health assessment** combining heartbeats, performance, and error rates
- **Decision**: Multi-factor health assessment with adaptive baselines
- **Rationale**: Multi-factor assessment provides more accurate health detection by considering multiple health indicators rather than just connectivity. Adaptive baselines reduce false positives by accounting for normal variations in consumer performance while still detecting significant degradation.
- **Consequences**: More accurate health detection and reduced unnecessary rebalancing, but increased complexity in threshold tuning and potential for delayed failure detection in edge cases.

Decision: Administrative API Authentication Model

- **Context**: The administrative API needs security to prevent unauthorized access while remaining simple to deploy and operate in various environments.
- **Options Considered**:
  1. **No authentication** for development simplicity
  2. **Simple token-based authentication** with preconfigured tokens
  3. **Integration with external identity providers** (OAuth, LDAP)
- **Decision**: Simple token-based authentication with role-based access control
- **Rationale**: Token-based authentication provides adequate security for operational environments while maintaining deployment simplicity. Role-based access enables separation between read-only monitoring access and admin operations. External identity integration can be added later if needed.
- **Consequences**: Adequate security with simple deployment, but requires manual token management and lacks advanced features like token rotation or centralized identity management.

Alerting Strategy Decision:

The monitoring system implements **threshold-based alerting** with support for multiple alerting channels (log files, HTTP webhooks, email). Alerts are generated when metrics exceed configurable thresholds for sustained periods, preventing alert fatigue from temporary spikes. The alerting system supports alert grouping and deduplication to prevent flooding during widespread issues.

Performance Impact Considerations:

All monitoring decisions prioritize **minimal impact on message processing performance**. Metrics collection uses lock-free data structures where possible and batches metric updates to reduce contention. The embedded metrics storage uses a dedicated thread pool to prevent blocking message processing threads. Administrative API requests use separate thread pools to ensure monitoring activities don't interfere with message operations.

Implementation Guidance

This section provides practical implementation guidance for building the monitoring and observability components, focusing on the embedded metrics storage, health checking system, and administrative API infrastructure.

Technology Recommendations:

Component	Simple Option	Advanced Option
HTTP API Server	`net/http` with `gorilla/mux`	`gin-gonic/gin` with middleware
Metrics Storage	In-memory maps with periodic disk writes	Embedded BadgerDB with time-series optimization
Time Series Math	Manual aggregation with sliding windows	Third-party library like `VictoriaMetrics/lib`
JSON Serialization	Standard `encoding/json`	`json-iterator` for performance
Background Tasks	Simple goroutines with channels	`robfig/cron` for scheduled tasks
Rate Limiting	Token bucket with maps	`golang.org/x/time/rate`

Recommended Module Structure:

```
project-root/
  internal/monitoring/
    collector.go           ← MetricsCollector implementation
    collector_test.go      ← Unit tests for metrics collection
    timeseries.go          ← TimeSeries storage and aggregation
    health.go              ← HeartbeatTracker and health checking
    health_test.go         ← Health check algorithm tests
  internal/api/
    server.go              ← HTTP server setup and routing
    handlers.go            ← REST endpoint implementations
    auth.go                ← Authentication middleware
    errors.go              ← Structured error responses
  internal/metrics/
    types.go               ← Metric data structures and constants
    aggregation.go         ← Time-series aggregation algorithms
  cmd/server/
    main.go                ← Integration with main broker server
```

Core Metrics Collection Infrastructure:

```go
// MetricsCollector provides embedded time-series storage for broker metrics
type MetricsCollector struct {
    series    map[string]*TimeSeries  // metric name -> time series data
    mutex     sync.RWMutex            // protects concurrent access
    retention map[string]time.Duration // per-metric retention policies

    // Background processing
    aggregationTicker *time.Ticker     // triggers periodic aggregation
    persistenceTicker *time.Ticker     // triggers disk writes
    shutdownCh        chan struct{}    // graceful shutdown signal
}

// NewMetricsCollector creates a configured metrics collector
func NewMetricsCollector(config MetricsConfig) *MetricsCollector {
    // TODO 1: Initialize series map and retention policies from config
    // TODO 2: Create background tickers for aggregation (1min) and persistence (5min)
    // TODO 3: Start background goroutines for aggregation and cleanup
    // TODO 4: Register shutdown handler to flush data on graceful stop
}

// RecordMetric adds a data point to the specified metric time series
func (mc *MetricsCollector) RecordMetric(name string, value float64, tags map[string]string) error {
    // TODO 1: Validate metric name and value (non-negative for counters)
    // TODO 2: Acquire write lock and find/create time series for metric
    // TODO 3: Add data point with current timestamp and provided tags
    // TODO 4: Check if series exceeds memory limits and trigger aggregation
    // Hint: Use atomic operations for frequently updated counters
}

// QueryMetrics retrieves time series data for the specified time range
func (mc *MetricsCollector) QueryMetrics(name string, start, end time.Time) ([]DataPoint, error) {
    // TODO 1: Acquire read lock and locate time series
    // TODO 2: Binary search for start time in data points
    // TODO 3: Collect points within time range
    // TODO 4: Apply any requested aggregation (avg, max, min, sum)
}
```

Time Series Storage Implementation:

```go
// TimeSeries stores chronological data points with automatic aggregation
type TimeSeries struct {
    points     []DataPoint             // chronologically ordered data points
    capacity   int                     // maximum points before aggregation
    retention  time.Duration           // how long to keep detailed data
    aggregated map[time.Time]DataPoint // aggregated data points (hourly/daily)
    mutex      sync.RWMutex            // protects concurrent access
}

// AddPoint inserts a new data point in chronological order
func (ts *TimeSeries) AddPoint(value float64, timestamp time.Time, tags map[string]string) {
    // TODO 1: Create DataPoint with provided values
    // TODO 2: Find insertion position maintaining chronological order
    // TODO 3: Insert point and shift remaining elements if needed
    // TODO 4: Check capacity and trigger aggregation if threshold exceeded
    // Hint: Use binary search for insertion position in large series
}

// AggregateOldData combines detailed points into coarser time buckets
func (ts *TimeSeries) AggregateOldData(bucketDuration time.Duration) {
    // TODO 1: Identify points older than retention threshold
    // TODO 2: Group points by time bucket (hour/day boundaries)
    // TODO 3: Calculate aggregate values (avg, min, max, count) per bucket
    // TODO 4: Replace detailed points with aggregated buckets
    // TODO 5: Update aggregated map with computed values
}
```

Health Check System Implementation:

```go
// HeartbeatTracker monitors consumer health and triggers cleanup
type HeartbeatTracker struct {
    heartbeats       map[string]*ConsumerHeartbeat // consumer ID -> heartbeat state
    timeoutThreshold time.Duration                // max time between heartbeats
    cleanupInterval  time.Duration                // how often to check for timeouts

    // Integration points
    groupCoordinator GroupCoordinatorInterface    // triggers rebalancing
    ackTracker       AckTrackerInterface          // retrieves pending messages

    // Background processing
    cleanupTicker    *time.Ticker                 // periodic health checks
    shutdownCh       chan struct{}                // graceful shutdown
}

// ProcessHeartbeat updates the heartbeat timestamp for a consumer
func (ht *HeartbeatTracker) ProcessHeartbeat(consumerID string, status ConsumerStatus) error {
    // TODO 1: Validate consumer ID and status fields
    // TODO 2: Find or create ConsumerHeartbeat entry
    // TODO 3: Update LastHeartbeat timestamp to current time
    // TODO 4: Reset ConsecutiveMisses counter to zero
    // TODO 5: Update ProcessingStatus and performance metrics
}

// CheckForTimeouts scans all consumers and identifies unhealthy ones
func (ht *HeartbeatTracker) CheckForTimeouts() []string {
    // TODO 1: Calculate timeout cutoff time (now - timeoutThreshold)
    // TODO 2: Iterate through all consumer heartbeats
    // TODO 3: Identify consumers with LastHeartbeat before cutoff
    // TODO 4: Increment ConsecutiveMisses for timed-out consumers
    // TODO 5: Return list of consumer IDs exceeding consecutive miss limit
    // Hint: Use 2-3 consecutive misses before marking truly unhealthy
}

// CleanupUnhealthyConsumer removes a failed consumer and reassigns work
func (ht *HeartbeatTracker) CleanupUnhealthyConsumer(consumerID string) error {
    // TODO 1: Retrieve all pending messages for the consumer from AckTracker
```

```
    // TODO 2: Return pending messages to their topics for reassignment

    // TODO 3: Remove consumer from all consumer groups via GroupCoordinator

    // TODO 4: Trigger rebalancing for affected consumer groups

    // TODO 5: Close consumer's network connection and clean up subscriptions

    // TODO 6: Remove consumer from heartbeat tracking map

    // TODO 7: Record cleanup metrics and log cleanup completion
}
```

Administrative API Server:

```go
// APIServer provides REST endpoints for broker monitoring and administration
type APIServer struct {
    router       *mux.Router              // HTTP request router
    broker       *MessageBroker           // main broker instance for state access
    auth         *TokenAuthenticator      // validates API tokens
    rateLimiter  *RateLimiter            // prevents API abuse

    // Component interfaces
    topicManager    TopicManagerInterface
    groupCoordinator GroupCoordinatorInterface
    metricsCollector *MetricsCollector
    healthTracker   *HeartbeatTracker
}

// NewAPIServer creates and configures the administrative HTTP server
func NewAPIServer(config APIConfig, broker *MessageBroker) *APIServer {
    // TODO 1: Create mux router and configure middleware stack
    // TODO 2: Initialize TokenAuthenticator with configured API keys
    // TODO 3: Setup RateLimiter with per-token bucket configuration
    // TODO 4: Register all REST endpoint handlers
    // TODO 5: Configure CORS headers for browser-based monitoring tools
}

// HandleTopicList returns paginated list of topics with basic metrics
func (api *APIServer) HandleTopicList(w http.ResponseWriter, r *http.Request) {
    // TODO 1: Parse query parameters (limit, offset, pattern filter)
    // TODO 2: Authenticate request and check read permissions
    // TODO 3: Apply rate limiting based on request token
    // TODO 4: Retrieve topic list from TopicManager with filtering
    // TODO 5: Gather basic metrics (message count, subscriber count) per topic
    // TODO 6: Apply pagination to results and generate response
    // TODO 7: Set appropriate cache headers and return JSON
}

// HandleConsumerGroupDetail returns detailed consumer group information
func (api *APIServer) HandleConsumerGroupDetail(w http.ResponseWriter, r *http.Request) {
    // TODO 1: Extract group name from URL path parameters
```

```
    // TODO 2: Validate group exists and authenticate request
    // TODO 3: Retrieve group details from GroupCoordinator
    // TODO 4: Gather member health status from HeartbeatTracker
    // TODO 5: Calculate lag metrics for group from MetricsCollector
    // TODO 6: Include rebalancing history and assignment strategy
    // TODO 7: Format comprehensive response with all group state
}
```

Rate Limiting Implementation:

```go
// RateLimiter provides token-bucket rate limiting for API endpoints
type RateLimiter struct {
    buckets map[string]*TokenBucket    // token -> rate limiter bucket
    mutex   sync.RWMutex               // protects bucket map

    defaultReadRate  time.Duration     // interval between read operations
    defaultWriteRate time.Duration     // interval between write operations
}

// CheckRateLimit verifies if request is within rate limits
func (rl *RateLimiter) CheckRateLimit(token string, operation string) (bool, time.Duration) {
    // TODO 1: Determine rate limit based on operation type (read vs write)
    // TODO 2: Find or create TokenBucket for the provided token
    // TODO 3: Try to consume one token from the bucket
    // TODO 4: Return success/failure and retry-after duration
    // TODO 5: Refill bucket tokens based on elapsed time since last access
}
```

Milestone Checkpoint:

After implementing the monitoring system, you should be able to verify the following behavior:

1. **Metrics Collection**: Start the broker and publish some test messages. Check that metrics are being recorded:

```bash
curl http://localhost:8080/api/v1/metrics/topics | jq '.'
```

Expected: JSON response showing message counts, throughput rates per topic.

2. **Health Monitoring**: Start a consumer, let it process some messages, then forcibly disconnect it:

```bash
curl http://localhost:8080/api/v1/health/consumers | jq '.'
```

Expected: Consumer should appear as unhealthy after timeout period, and its messages should be reassigned.

3. **Administrative Operations**: Create topics and inspect their state via API:

```bash
curl -X POST http://localhost:8080/api/v1/topics \
     -H "Authorization: Bearer admin-token" \
     -d '{"name": "test-topic", "retention_policy": {"max_age": 3600}}'
```

Expected: Topic created successfully with configured retention policy.

Debugging Tips:

| Symptom | Likely Cause | How to Diagnose | Fix | |---|---|---| | Metrics not updating | Collection goroutines not running | Check goroutine count in `/api/v1/health` | Ensure MetricsCollector.Start() called | | API requests timing out | Rate limiting too aggressive | Check rate limiter logs and bucket states | Increase rate limits or optimize bucket refill | | Consumer health false positives | Heartbeat timeout too short | Monitor heartbeat miss patterns | Increase timeout threshold or grace period | | High memory usage from metrics | Aggregation not running or insufficient | Check aggregation ticker and memory stats | Trigger manual aggregation and tune retention | | Authentication failures | Invalid tokens or middleware order | Check token validation logs | Verify token configuration and middleware stack |

Performance Optimization Hints:

- Use atomic operations for frequently updated counters to reduce lock contention
- Batch metric updates and write to disk periodically rather than on every update
- Implement metric sampling for very high-frequency events to control overhead
- Use connection pooling for HTTP clients accessing the administrative API
- Consider metric pre-aggregation at collection time for commonly queried statistics

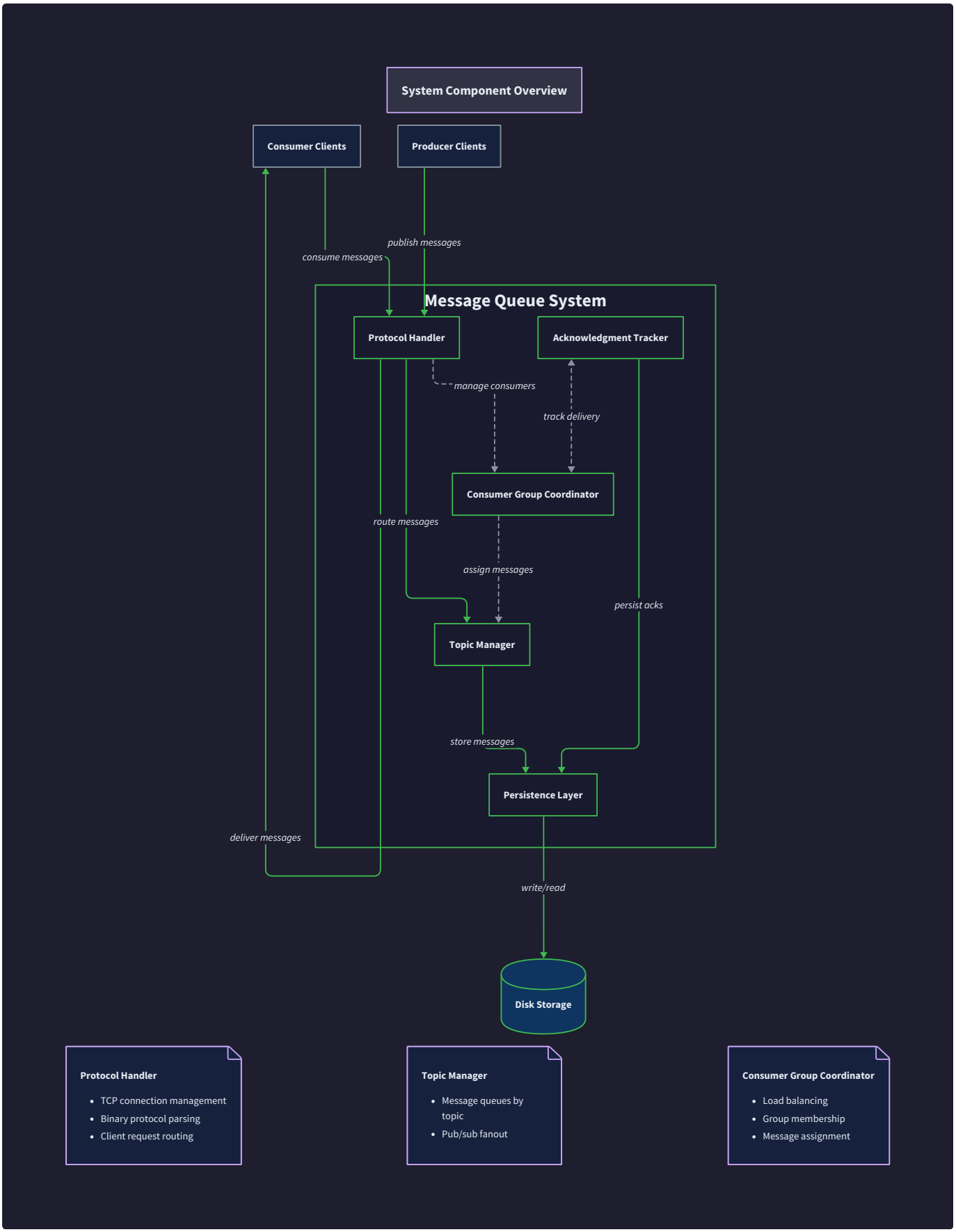
Component Interactions and Data Flow

> **Milestone(s):** All milestones (1-4) — describes how all system components collaborate to deliver messages from producers to consumers, handling failures and recovery across the entire system lifecycle

Mental Model: The Orchestra Symphony

Think of the message queue system as a symphony orchestra, where each component plays a specific role in creating harmonious message delivery. The **Protocol Handler** acts as the conductor, receiving requests and coordinating with other components. The **Topic Manager** serves as the sheet music library, organizing and distributing musical scores (messages) to the right sections. **Consumer Groups** represent instrument sections (strings, brass, woodwinds) where multiple players work together to handle their assigned parts. The **Acknowledgment Tracker** functions like the concert master, ensuring each section confirms they've completed their part before moving forward. Finally, the **Persistence Layer** acts as the recording equipment, capturing every note so the performance can be reconstructed if the concert hall loses power.

Just as a symphony requires precise timing and coordination between sections, message delivery involves careful orchestration between components. When a producer publishes a message, it triggers a carefully choreographed sequence involving persistence, routing, assignment, and acknowledgment. Understanding these interaction patterns is crucial because most message queue bugs occur at component boundaries where coordination breaks down.



This section traces four critical flows that demonstrate how components collaborate: message publication from producer to persistence and fanout, message consumption through consumer group assignment and acknowledgment, consumer group rebalancing when membership changes, and crash recovery rebuilding state from persistent logs.

Message Publish Flow

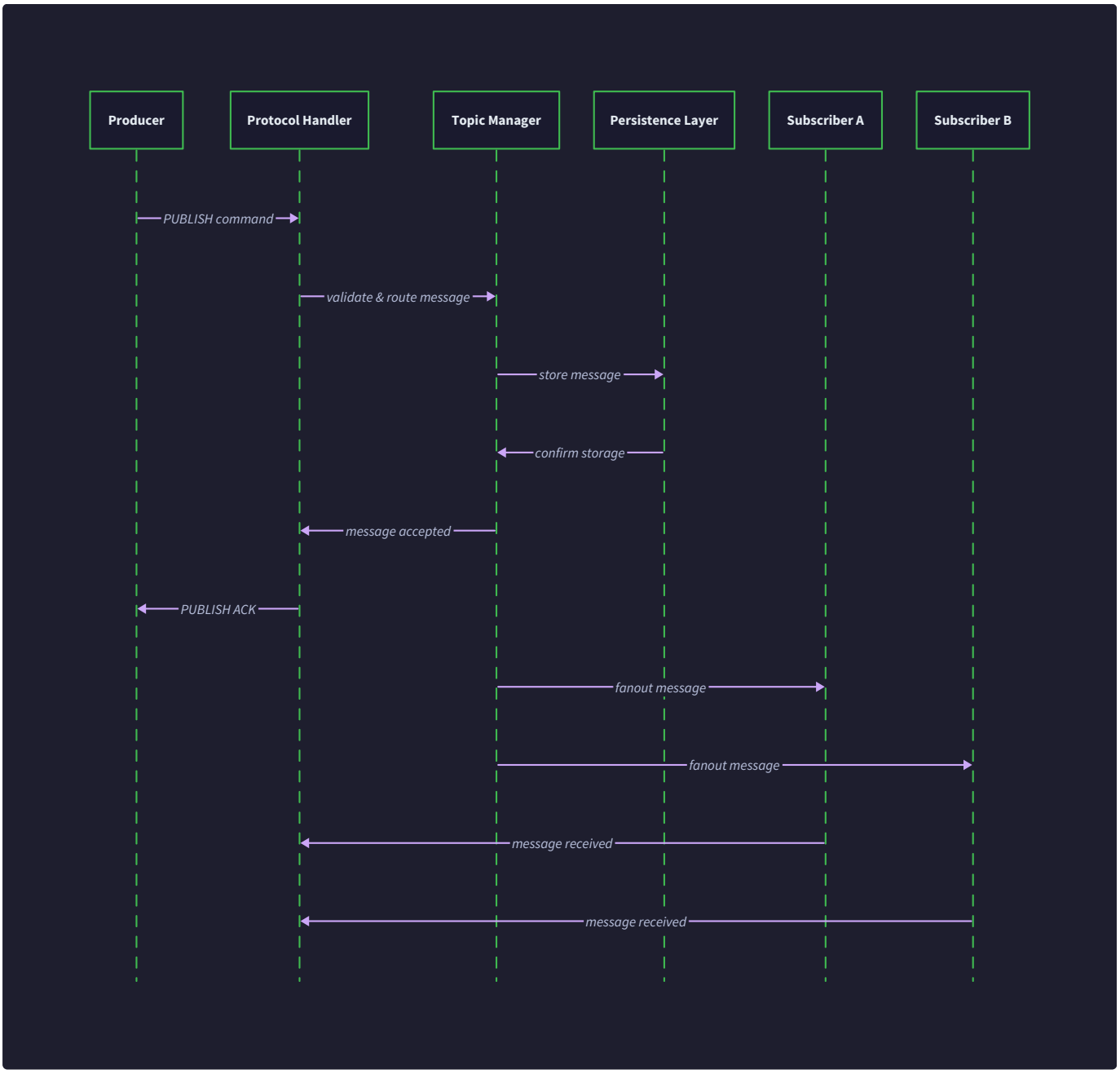
Mental Model: Restaurant Order Processing

Think of message publication like processing an order in a busy restaurant. The **customer** (producer) places an order with the **waiter** (Protocol Handler), who writes it on their pad and immediately files a carbon copy in the **order book** (WAL) to ensure it won't be lost. The waiter then gives the order to the **kitchen manager** (Topic Manager), who determines which **cooking stations** (subscribers or consumer groups) should receive the order based on the dish type. Each station gets their own copy of the order and begins preparation. If the restaurant loses power mid-service, they can reopen by reading the order book and seeing exactly which orders were taken but not yet completed.

This analogy captures the essential aspects of message publication: **immediate persistence** ensures durability, **topic-based routing** ensures the right consumers receive messages, and **fanout delivery** handles both pub/sub and consumer group distribution patterns.

End-to-End Publication Sequence

The message publication flow involves six distinct phases that ensure reliable message delivery while maintaining system consistency. Each phase has specific responsibilities and failure handling mechanisms.



Phase 1: Protocol Reception and Validation When a producer sends a `PUBLISH` command through the TCP connection, the Protocol Handler receives the binary frame and performs initial validation. The handler verifies the frame header magic number, checks payload size against `MaxFrameSize` limits, and validates that the topic name follows hierarchical naming conventions. If validation fails, the handler immediately returns an error response without involving other components.

Phase 2: Message Construction and ID Assignment Upon successful validation, the Protocol Handler constructs a `Message` instance using the `NewMessage` function. This process assigns a unique message ID using timestamp-based generation, captures the current timestamp for ordering, extracts headers from the frame payload, and associates the message with the producer's client ID for tracking purposes. The resulting message transitions to `StateCreated` and becomes ready for persistence.

Phase 3: Write-Ahead Log Persistence Before any routing or delivery occurs, the system writes a `RecordMessagePublished` entry to the Write-Ahead Log using the `AppendRecord` method. This critical step ensures message durability by performing an atomic write operation, executing an fsync to force disk synchronization, updating the log offset for sequencing, and recording all message metadata including headers and payload. Only after successful WAL persistence does the message proceed to routing.

Phase 4: Topic Resolution and Queue Enqueuing The Topic Manager receives the persisted message and handles topic resolution through its `PublishMessage` method. If the topic doesn't exist, the system creates it automatically with default retention policies. The manager then calls `EnqueueMessage` to add the message to the topic's message queue, updates topic statistics including total message count and last activity timestamp, and transitions the message to `StatePending` state.

Phase 5: Subscriber Discovery and Fanout Preparation The Topic Manager identifies all subscribers for the topic using both direct subscriptions and wildcard pattern matching through the `findWildcardMatches` method. For each matching subscription, the system determines the delivery mode (individual subscriber vs consumer group), creates appropriate `DeliveryTask` instances, and prepares delivery context including target consumer information and delivery deadline.

Phase 6: Message Delivery and Consumer Assignment The final phase handles actual message delivery based on subscription type. For individual subscribers, the system delivers copies directly to each subscriber's connection. For consumer groups, the Group Coordinator's `AssignMessage` method selects an available consumer using the configured assignment strategy, delivers the message to the chosen consumer, and registers the message with the Acknowledgment Tracker for timeout monitoring and redelivery handling.

Publication Flow Data Transformations

During publication, message data undergoes several transformations that add metadata and routing information while preserving the original payload integrity.

Phase	Input Data	Transformation	Output Data	Persistence Point
Protocol Reception	Binary TCP frame	Frame parsing, header validation	Raw message fields	None
Message Construction	Raw fields + producer context	ID generation, timestamp addition	Complete `Message` struct	None
WAL Persistence	Complete `Message`	Serialization to `WALRecord`	Persistent log entry	WAL file
Topic Enqueuing	Persisted `Message`	Topic queue insertion	Queued `Message`	In-memory queue
Subscriber Discovery	Queued `Message` + topic subscriptions	Pattern matching, fanout planning	`DeliveryTask` list	None
Consumer Assignment	`DeliveryTask` + group membership	Load balancing, assignment	Assigned `Message`	Acknowledgment tracker

The key insight is that persistence occurs **before** routing to ensure that even if delivery fails, the message remains available for retry. This ordering prevents message loss but requires careful coordination to avoid duplicate delivery during recovery scenarios.

Publication Error Handling and Rollback

The publication flow includes comprehensive error handling at each phase, with specific rollback procedures to maintain system consistency when failures occur.

Protocol-Level Errors: Frame parsing failures, oversized payloads, or malformed topic names trigger immediate rejection without involving downstream components. The Protocol Handler returns specific error codes (`StatusTopicNotFound` , `StatusPayloadTooLarge`) and logs the failure for monitoring.

Persistence Failures: WAL write failures represent the most critical error scenario because they prevent message durability guarantees. When `AppendRecord` fails due to disk space, permission issues, or I/O errors, the system rejects the publication request and returns an error to the producer. No cleanup is required since no state was modified.

Topic Management Errors: Failures during topic creation or queue enqueuing require careful handling since the message was already persisted. The system marks the WAL record with a failure annotation, removes any partial topic state, and schedules the message for retry through a background reconciliation process.

Delivery Failures: When subscriber discovery or consumer assignment fails, the message remains in the topic queue for later delivery attempts. The system logs delivery failures, updates retry counters, and relies on background processes to reattempt delivery based on configured retry policies.

> **Critical Design Insight**: The publication flow prioritizes durability over immediate delivery consistency. Messages are guaranteed to be persisted before any delivery attempts, ensuring that temporary delivery failures don't result in message loss. This design trades some latency for reliability, which aligns with the system's focus on data integrity.

Message Consumption Flow

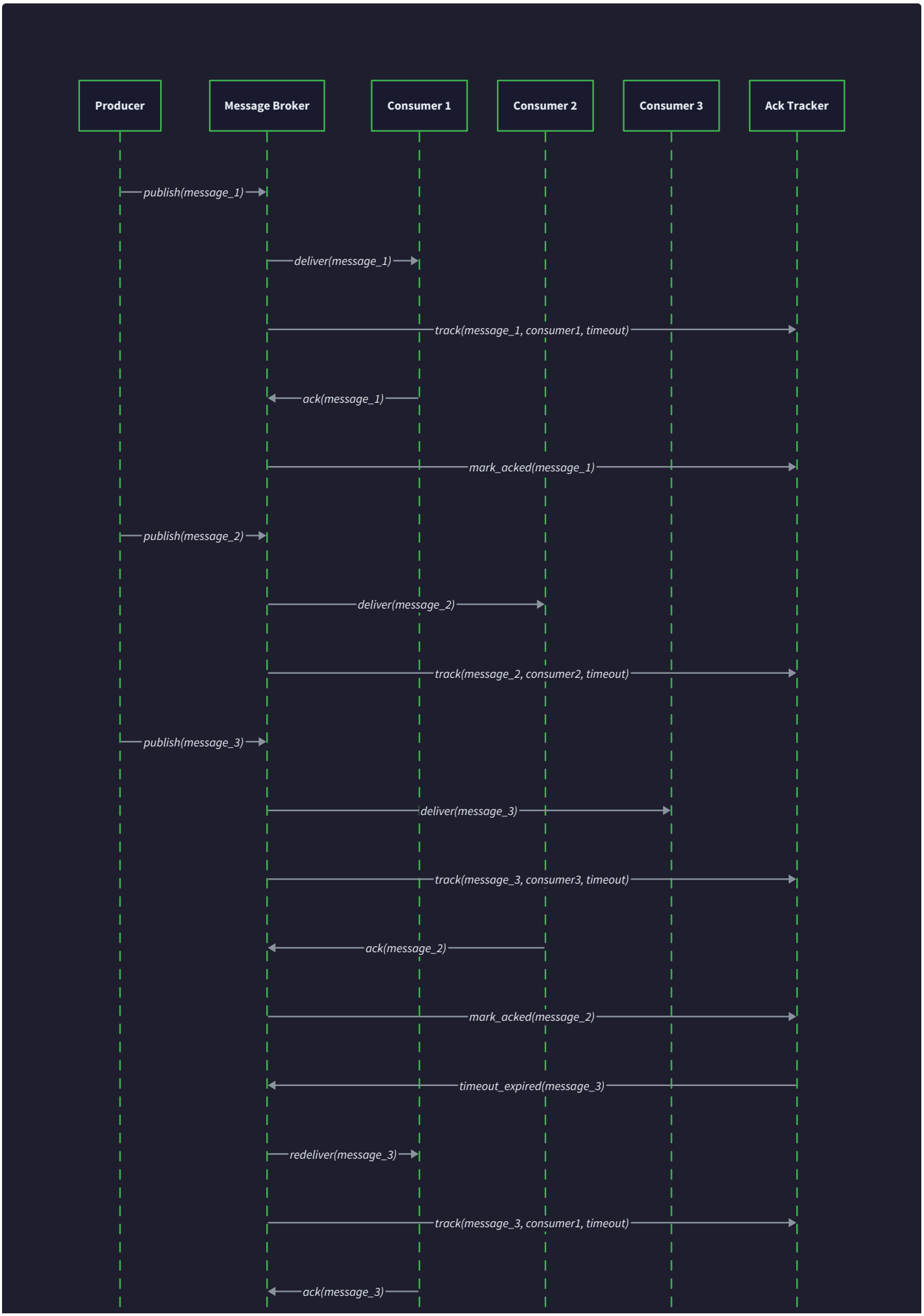
Mental Model: Hospital Emergency Room Triage

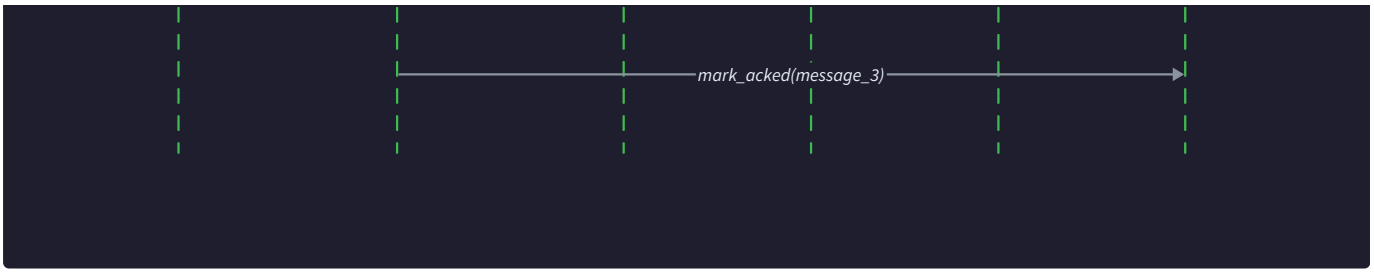
Think of message consumption like an emergency room triage system. **Patients** (messages) arrive and wait in different **specialty queues** (topics) based on their condition. **Triage nurses** (Consumer Group Coordinator) assess each patient and assign them to available **doctors** (consumers) within the appropriate **medical team** (consumer group) based on current workload and specialization. Each doctor must **confirm completion** (ACK) of treatment before taking the next patient. If a doctor doesn't respond within the expected time, the **charge nurse** (Acknowledgment Tracker) reassigns the patient to another available doctor. Patients who consistently cause treatment failures get transferred to the **specialist unit** (Dead Letter Queue) for expert handling.

This analogy captures the essential flow: **work distribution** ensures balanced load across consumers, **explicit acknowledgment** confirms successful processing, and **automatic reassignment** handles consumer failures without losing work.

Consumer Group Assignment and Message Distribution

The consumption flow begins when messages in topic queues need assignment to consumer group members. The Group Coordinator manages this process through its `AssignMessage` method, which implements the configured assignment strategy while considering consumer capacity and health status.


Producer
Message Broker
Consumer 1
Consumer 2
Consumer 3
Ack Tracker
publish(message_1)
deliver(message_1)
track(message_1, consumer1, timeout)
ack(message_1)
mark_acked(message_1)
publish(message_2)
deliver(message_2)
track(message_2, consumer2, timeout)
publish(message_3)
deliver(message_3)
track(message_3, consumer3, timeout)
ack(message_2)
mark_acked(message_2)
timeout_expired(message_3)
redeliver(message_3)
track(message_3, consumer1, timeout)
ack(message_3)



Assignment Strategy Execution When a message becomes available for assignment, the Group Coordinator evaluates all active consumers in the target group using the configured strategy. For round-robin assignment, the coordinator maintains a `NextAssignment` index that rotates through group members sequentially. For sticky assignment, the coordinator attempts to assign messages to consumers who previously handled messages from the same topic partition, minimizing context switching overhead.

The assignment process considers multiple factors beyond simple rotation: **consumer capacity** using the `MaxInflight` and `CurrentInflight` fields to prevent overloading, **consumer health** based on recent heartbeat status and error rates, **message affinity** for sticky assignment strategies, and **rebalancing state** to avoid assignments during group membership changes.

Message Delivery and Tracking Registration Once a consumer is selected, the system performs atomic delivery and tracking registration. The Group Coordinator updates the consumer's `CurrentInflight` counter, adds the message to the consumer's `PendingMessages` map, and calls the Acknowledgment Tracker's `TrackMessage` method to register the message for timeout monitoring.

The `TrackMessage` operation creates a `PendingMessage` entry containing the original message, assigned consumer ID, delivery timestamp, acknowledgment deadline calculated from current time plus timeout threshold, and empty delivery history that will record retry attempts. This tracking ensures that every delivered message has an explicit deadline and recovery mechanism.

Acknowledgment Processing and Consumer Feedback Loop

The consumption flow includes a continuous feedback loop where consumers report processing results through ACK and NACK commands, enabling the system to make informed assignment decisions and handle failures appropriately.

Successful Acknowledgment Flow When a consumer successfully processes a message and sends an `ACK` command, the Acknowledgment Tracker's `ProcessAck` method performs several coordinated operations. It locates the pending message using the message ID, validates that the acknowledging consumer matches the assigned consumer, removes the message from the pending messages map, decreases the consumer's `CurrentInflight` counter, updates the consumer's `ProcessedCount` and average processing time statistics, and writes an `RecordMessageAcked` entry to the WAL for durability.

The acknowledgment also triggers consumer health updates within the Group Coordinator. The system records the successful processing event, updates the consumer's `LastHeartbeat` timestamp, adjusts the consumer's error rate using exponential smoothing, and potentially increases the consumer's priority for future assignments based on consistent successful processing.

Negative Acknowledgment Handling When a consumer sends a `NACK` command indicating processing failure, the system initiates immediate redelivery through the `ProcessNack` method. This process increments the message's `RetryCount`, adds a delivery attempt entry to the message's history, removes the message from the failing consumer's pending list, and schedules immediate reassignment to a different consumer in the group.

The NACK handling includes failure pattern analysis to prevent infinite retry loops. The system tracks per-consumer error rates, identifies message-specific failure patterns, applies exponential backoff for consistently failing messages, and routes messages exceeding `MaxRetries` to the Dead Letter Queue for manual inspection.

Consumer Group Load Balancing and Health Monitoring

The consumption flow continuously monitors consumer performance and adjusts assignment behavior to maintain optimal load distribution and system throughput.

Dynamic Load Assessment The Group Coordinator maintains detailed load information for each consumer through the `GetConsumerLoad` method. This assessment includes current pending message count, average message processing time calculated from recent acknowledgments, error rate based on recent ACK/NACK ratios, heartbeat freshness indicating consumer connectivity, and oldest pending message age for detecting stuck consumers.

This load information influences assignment decisions in real-time. High-performing consumers with low error rates receive preferential assignment, while consumers showing elevated error rates or slow processing times get reduced message allocation until their performance

improves.

Health-Based Assignment Adjustment The system adjusts assignment behavior based on consumer health trends. Healthy consumers with consistent acknowledgments and fast processing times may receive increased `MaxInflight` limits to handle more concurrent messages. Consumers showing degraded performance through increased error rates, slower acknowledgment times, or missed heartbeats receive reduced message assignment until their health improves.

Automatic Consumer Recovery When consumers fail or become unresponsive, the HeartbeatTracker's `CheckForTimeouts` method identifies dead consumers and initiates recovery. The system calls `CleanupUnhealthyConsumer` to retrieve all pending messages from the failed consumer, triggers consumer group rebalancing to redistribute capacity, and reassigns recovered messages to remaining healthy consumers.

> **Performance Insight**: The consumption flow optimizes for sustained throughput rather than immediate assignment. By continuously monitoring consumer health and adjusting assignment behavior accordingly, the system prevents the common problem where a single slow consumer becomes a bottleneck for the entire consumer group.

Consumption Flow Error Scenarios and Recovery

The consumption flow handles various failure scenarios through coordinated recovery mechanisms that prevent message loss while maintaining processing progress.

Error Scenario	Detection Method	Recovery Action	Coordination Required
Consumer crash	Missed heartbeats	Pending message recovery and reassignment	Group Coordinator + Acknowledgment Tracker
Message timeout	Deadline monitoring	Automatic redelivery to different consumer	Acknowledgment Tracker + Group Coordinator
Processing failure	Explicit NACK	Immediate reassignment with backoff	Acknowledgment Tracker only
Network partition	Connection failure	Consumer cleanup and rebalancing	Protocol Handler + Group Coordinator
Poison message	Retry count exceeded	Dead Letter Queue routing	Acknowledgment Tracker + DLQ Manager
Consumer overload	High pending count	Reduced assignment priority	Group Coordinator load balancer

Each error scenario requires specific coordination between components to ensure consistent recovery without message duplication or loss. The key principle is that **message ownership** transfers cleanly between components, with explicit handoff protocols that prevent race conditions during failure recovery.

Consumer Group Rebalancing Flow

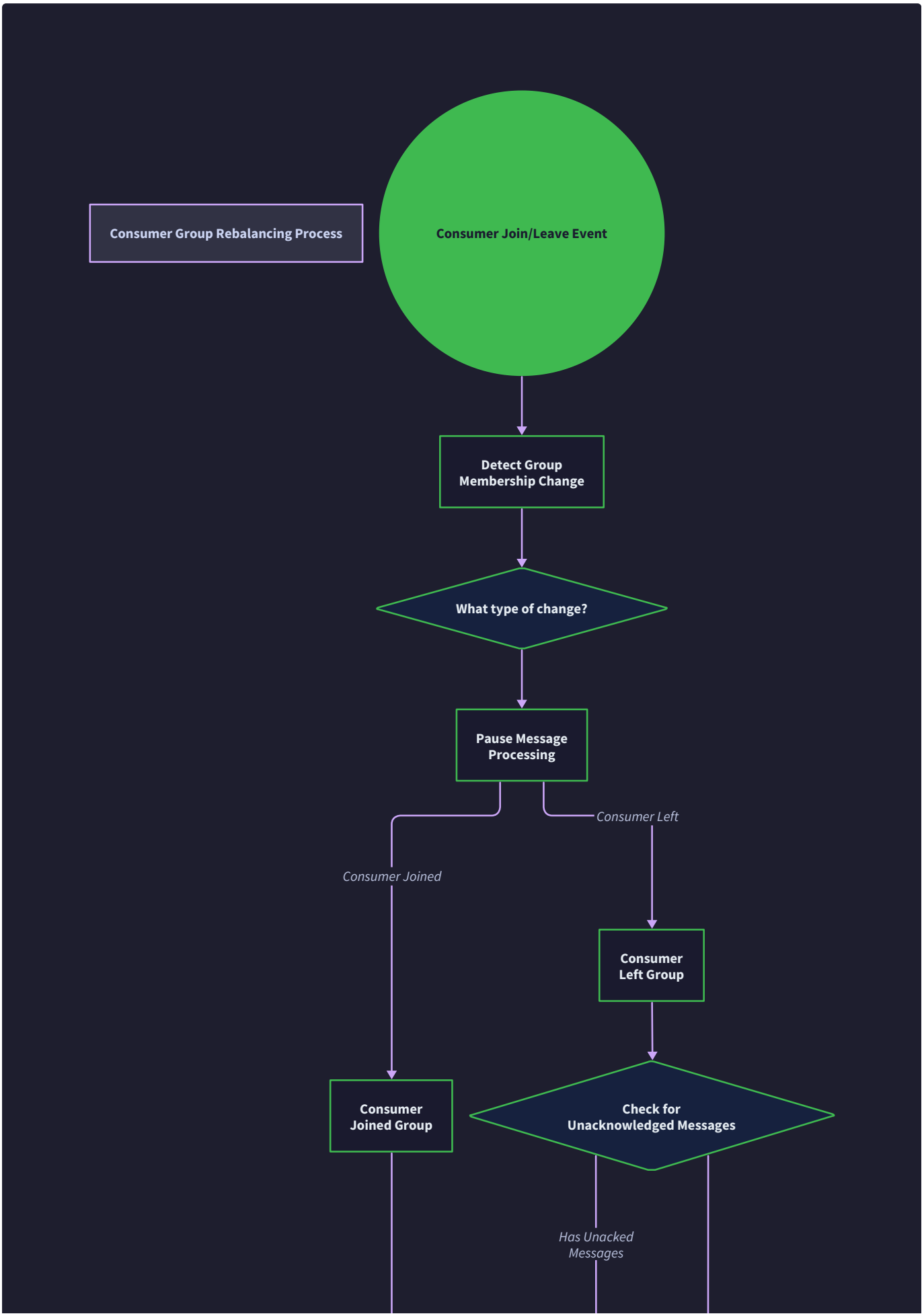
Mental Model: Restaurant Staff Shift Change

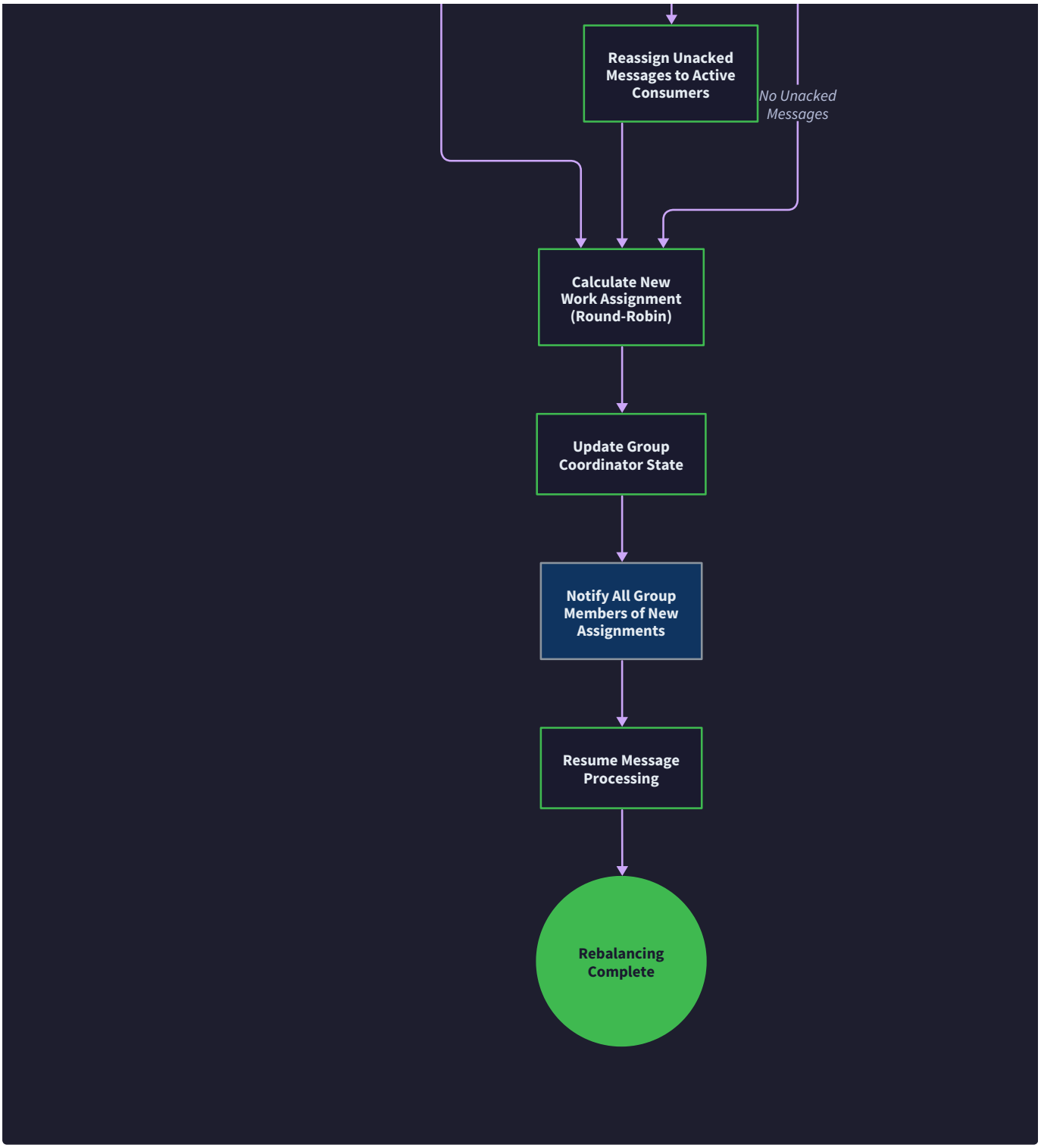
Think of consumer group rebalancing like managing a restaurant during a shift change. When **new waiters arrive** (consumers join) or **existing waiters leave** (consumers crash or disconnect), the **restaurant manager** (Group Coordinator) must **reassign table sections** (message assignment responsibility) to maintain balanced service. During the transition, the manager **pauses new seating** (stops new message assignments), **ensures current orders are handled** (completes pending messages), **redistributes sections** based on the new staff count, and **updates the seating chart** (assignment generation number) so all staff know the new arrangement.

The critical insight is that rebalancing requires **coordinated pausing** of normal operations to prevent chaos, **explicit state synchronization** across all participants, and **atomic assignment updates** to avoid split-brain scenarios where different consumers think they own the same work.

Rebalancing Trigger Conditions and Coordination

Consumer group rebalancing occurs when membership changes threaten the balanced distribution of message processing work. The Group Coordinator monitors several trigger conditions and initiates rebalancing through a coordinated multi-phase process.


Consumer Group Rebalancing Process
Consumer Join/Leave Event
Detect Group Membership Change
What type of change?
Pause Message Processing
Consumer Left
Consumer Joined
Consumer Left Group
Consumer Joined Group
Check for Unacknowledged Messages
Has Unacked Messages



Primary Rebalancing Triggers The system triggers rebalancing when specific membership events occur: **consumer join** when `JoinGroup` is called by a new consumer, **consumer departure** when `LeaveGroup` is explicitly called, **consumer failure** when heartbeat monitoring detects unresponsive consumers, **consumer capacity change** when existing consumers modify their `MaxInflight` settings, and **assignment strategy change** when administrative operations update the group's assignment policy.

Each trigger condition includes a **stabilization delay** to prevent thrashing when multiple membership changes occur rapidly. The Group Coordinator waits for a configurable quiet period before initiating rebalancing, allowing multiple changes to batch together into a single rebalancing operation.

Rebalancing State Management When rebalancing begins, the Group Coordinator updates the `ConsumerGroup` state to reflect the coordination process. It sets `RebalanceInProgress` to true, preventing new message assignments, increments the `RebalanceGeneration` number to invalidate stale assignment information, records the `LastRebalance` timestamp for monitoring purposes, and creates a snapshot of current membership for consistent decision-making throughout the process.

The generation number serves as a **distributed coordination mechanism** that prevents split-brain conditions. All message assignments and acknowledgments include the generation number, allowing the system to detect and ignore operations from consumers operating under obsolete group configurations.

Multi-Phase Rebalancing Protocol

The rebalancing process follows a carefully orchestrated multi-phase protocol that ensures consistent state across all participants while minimizing message processing disruption.

Phase 1: Assignment Freeze and Pending Message Drain The rebalancing process begins by freezing new message assignments to prevent additional work distribution during the transition. The Group Coordinator stops calling `AssignMessage` for the rebalancing group, allowing in-flight message processing to continue, waits for acknowledgments of currently assigned messages, and collects unacknowledged messages from consumers that are leaving the group.

During this phase, the system maintains a **drain timeout** to prevent indefinite waiting. If consumers don't acknowledge their pending messages within the timeout period, those messages are collected for reassignment during the rebalancing completion phase.

Phase 2: Membership Reconciliation and Assignment Planning With assignments frozen, the Group Coordinator reconciles the final membership list by confirming heartbeat status of all claimed members, removing consumers that failed heartbeat checks, adding new consumers that successfully joined during the stabilization period, and validating that all remaining consumers have active network connections.

The coordinator then calculates new assignment distributions using the configured assignment strategy. For round-robin assignment, this involves dividing message flow evenly across the new member count. For sticky assignment, the system attempts to preserve existing assignments where possible while rebalancing excess load from departed consumers.

Phase 3: Assignment Distribution and Generation Update The final phase distributes new assignment information to all group members and updates the generation number atomically. The Group Coordinator sends assignment notifications to each consumer containing the new

generation number, updated assignment responsibilities, and revised group membership list. Each consumer acknowledges receipt of the new assignment, confirming their readiness to process messages under the new configuration.

Once all consumers acknowledge the new assignments, the coordinator atomically updates the group's state: incrementing the generation number, clearing the `RebalanceInProgress` flag, updating the `MessageAssignments` map with new consumer responsibilities, and resuming normal message assignment operations.

Rebalancing Failure Recovery and Split-Brain Prevention

The rebalancing protocol includes comprehensive failure handling to ensure system consistency even when rebalancing operations encounter errors or timeouts.

Coordinator Failure During Rebalancing If the Group Coordinator crashes during rebalancing, the system uses WAL-based recovery to determine the rebalancing state upon restart. The recovery process examines recent WAL records to identify incomplete rebalancing operations, determines which consumers were participating in the interrupted rebalancing, and decides whether to complete the interrupted rebalancing or initiate a fresh rebalancing with current membership.

The coordinator writes rebalancing checkpoints to the WAL at each phase boundary, enabling precise recovery of the rebalancing state. These checkpoints include the target membership list, assignment calculations, and acknowledgment status from each participant.

Consumer Communication Failures When consumers fail to acknowledge new assignments within the timeout period, the coordinator treats them as failed members and excludes them from the final membership list. This approach prevents indefinite blocking while ensuring that only responsive consumers participate in the rebalanced group.

The excluded consumers eventually detect their exclusion through heartbeat responses that indicate an obsolete generation number, triggering them to rejoin the group through a fresh `JoinGroup` operation.

Generation Number Coordination The generation number mechanism prevents split-brain scenarios where consumers operate under different group configurations. All message-related operations include generation numbers, allowing the system to detect and reject operations from consumers using obsolete configurations.

When a consumer attempts to acknowledge a message using an old generation number, the Acknowledgment Tracker rejects the acknowledgment and instructs the consumer to rejoin the group. This ensures that message processing only proceeds under consistent group configurations.

> **Reliability Insight**: The rebalancing protocol prioritizes consistency over availability during membership changes. By pausing new message assignments during rebalancing, the system accepts temporary throughput reduction to ensure that message processing never occurs under ambiguous ownership conditions. This design prevents the data consistency problems that plague eventually-consistent message queue implementations.

Rebalancing Performance Optimization and Monitoring

The rebalancing implementation includes several optimizations to minimize disruption and provide visibility into rebalancing operations for operational monitoring.

Rebalancing Impact Minimization The system implements several strategies to reduce rebalancing overhead: **sticky assignment preferences** that minimize consumer reassignment during membership changes, **batched membership changes** that collect multiple join/leave operations into single rebalancing cycles, **predictive rebalancing** that initiates rebalancing slightly before consumers are declared dead based on heartbeat patterns, and **assignment change minimization** algorithms that compute assignment changes with minimal consumer disruption.

Rebalancing Metrics and Observability The Group Coordinator exposes detailed metrics about rebalancing operations through the monitoring API: **rebalancing frequency** indicating group stability, **rebalancing duration** showing coordination efficiency, **assignment change counts** measuring disruption levels, **consumer churn rates** indicating application health, and **failed rebalancing attempts** highlighting coordination problems.

These metrics enable operational teams to tune rebalancing parameters, identify problematic consumer applications, and optimize group configurations for stability.

Crash Recovery Flow

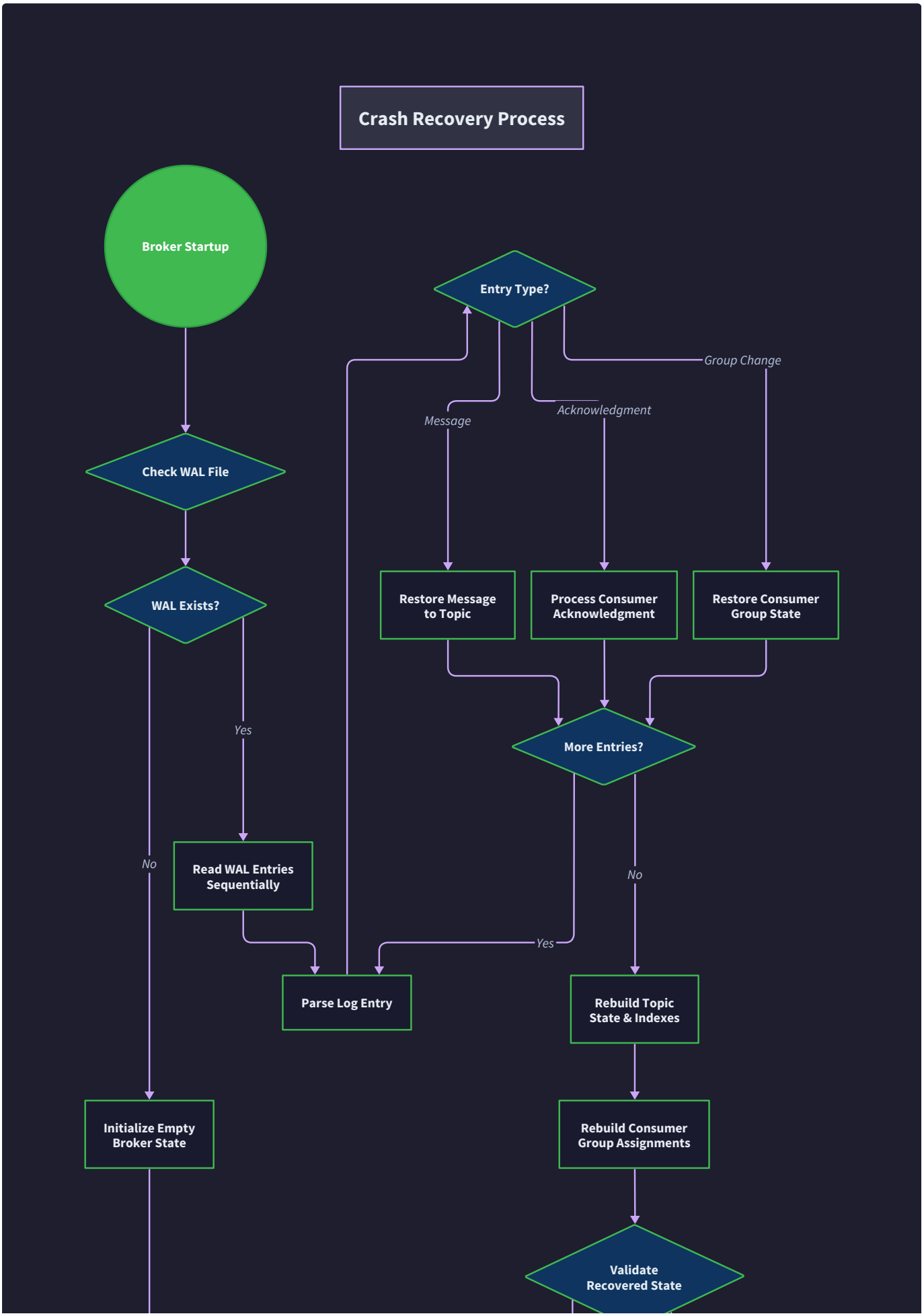
Mental Model: Archaeological Site Reconstruction

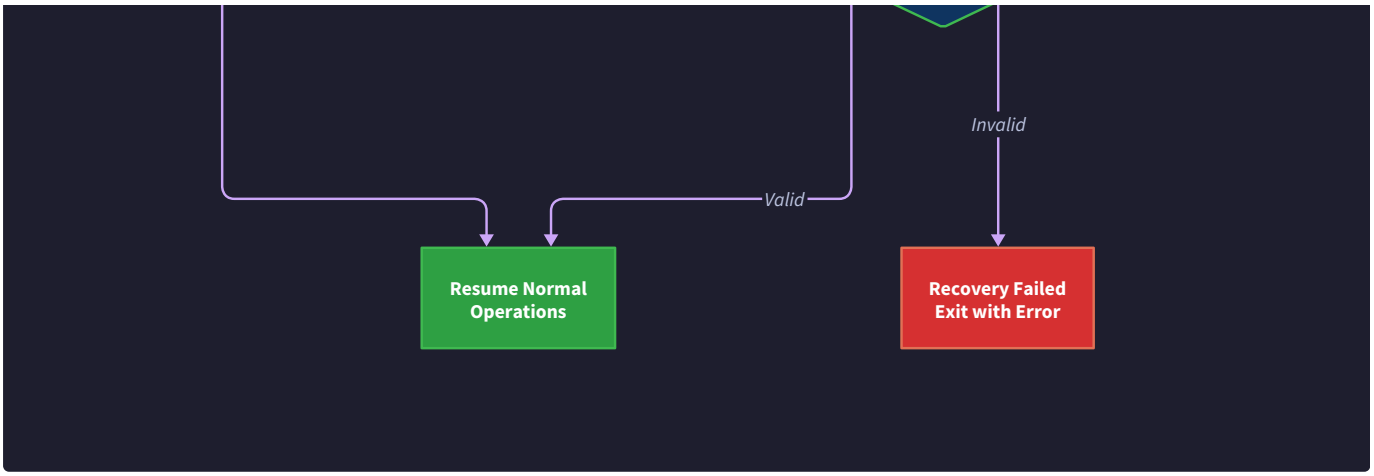
Think of crash recovery like archaeologists reconstructing an ancient city from excavated artifacts. The **Write-Ahead Log** serves as the **archaeological record** — a chronological sequence of preserved evidence (log entries) that documents what happened before the catastrophe (crash). The **recovery process** resembles **systematic excavation** where archaeologists examine each artifact (WAL record) in chronological order, piece together the sequence of events (system state), and **reconstruct the living city** (restore broker functionality) from the preserved evidence.

Just as archaeologists must distinguish between **foundation stones** (persistent state) and **temporary scaffolding** (transient state), the recovery process must rebuild only the essential system state while discarding ephemeral information like network connections and in-progress protocol operations.

Startup Sequence and State Reconstruction

The crash recovery process follows a systematic sequence that rebuilds system state from persistent logs while handling various corruption and consistency scenarios that can occur during unclean shutdowns.


Crash Recovery Process
Broker Startup
Check WAL File
WAL Exists?
No
Yes
Initialize Empty Broker State
Read WAL Entries Sequentially
Parse Log Entry
Entry Type?
Message
Acknowledgment
Group Change
Restore Message to Topic
Process Consumer Acknowledgment
Restore Consumer Group State
More Entries?
Yes
No
Rebuild Topic State & Indexes
Rebuild Consumer Group Assignments
Validate Recovered State



Phase 1: Log Integrity Validation and Corruption Detection Recovery begins with comprehensive validation of the Write-Ahead Log to detect corruption, incomplete writes, or structural damage. The recovery process opens the log file and validates the overall file structure, scans record headers to identify truncated or corrupted entries, verifies record checksums to detect data corruption, and identifies the **recovery point** — the last complete, uncorrupted record that can serve as the foundation for state reconstruction.

If corruption is detected, the system implements **corruption boundary isolation** by truncating the log at the last known good record, logging detailed corruption information for debugging, and proceeding with recovery from the truncated state. This approach prioritizes data consistency over potential message loss in the corrupted region.

Phase 2: Topic and Subscription State Reconstruction With a validated log, the recovery process rebuilds the Topic Manager's state by scanning for `RecordTopicCreated` entries and recreating topic structures with their retention policies. For each topic, the system reconstructs the message queue by processing `RecordMessagePublished` entries chronologically, rebuilds subscription lists from `RecordSubscriptionAdded` entries, and applies retention policies to remove expired messages that accumulated during downtime.

The reconstruction process maintains **temporal consistency** by applying log records in strict chronological order, ensuring that the recovered state reflects the exact sequence of operations that occurred before the crash.

Phase 3: Consumer Group Membership and Assignment Recovery Consumer group recovery requires careful handling because consumer connections are ephemeral and don't survive broker restarts. The recovery process reconstructs group definitions from `RecordConsumerJoined` entries, but marks all consumers as **disconnected** since their network connections were lost during the crash.

The system rebuilds consumer group metadata including group names and assignment strategies, but initializes all groups with empty membership lists. When consumers reconnect after broker startup, they automatically rejoin their groups through normal `JoinGroup` operations, triggering rebalancing to redistribute work appropriately.

Phase 4: Message Acknowledgment and Pending Message Recovery The Acknowledgment Tracker recovery involves reconstructing the state of all unacknowledged messages by correlating message delivery records with acknowledgment records. The process scans for `RecordMessagePublished` entries to identify all messages, matches them with `RecordMessageAcked` entries to determine which messages were successfully processed, and identifies **orphaned messages** — messages that were delivered to consumers but never acknowledged before the crash.

Orphaned messages require special handling since their original consumers are no longer connected. The recovery process adds these messages back to their respective topic queues for redelivery, updates their retry counts to reflect the failed delivery attempt, and ensures they become available for assignment once consumers reconnect.

WAL-Based State Consolidation and Consistency Validation

The recovery process implements sophisticated state consolidation algorithms that handle the complexity of rebuilding consistent system state from a sequence of discrete log operations.

State Consolidation Algorithm State reconstruction from WAL records requires careful handling of operation sequences that may have been interrupted mid-flight. The consolidation algorithm processes log records in chronological order while maintaining **operation atomicity invariants**.

For each record type, the system applies specific reconstruction logic: `RecordMessagePublished` entries recreate message objects and add them to topic queues, `RecordMessageAcked` entries remove messages from pending status and update consumer statistics, `RecordConsumerJoined` entries reconstruct group membership (but mark consumers as disconnected), `RecordTopicCreated` entries rebuild topic metadata and retention policies, and `RecordCheckpointCreated` entries provide recovery shortcuts for large state reconstructions.

Cross-Component State Validation After individual component state recovery, the system performs **cross-component consistency validation** to detect and resolve any inconsistencies that might exist due to crash timing or log corruption. The validation process ensures that all messages referenced in acknowledgment records exist in topic queues or have valid acknowledgment status, all consumer group assignments reference existing consumers and messages, all topic subscriptions reference valid topics and connected consumers, and all pending messages have valid acknowledgment deadlines and retry counts.

When inconsistencies are detected, the recovery process applies **conservative consistency resolution** by preferring data preservation over operational efficiency. For example, if a message has an acknowledgment record but no corresponding publish record, the system logs the inconsistency but doesn't fail recovery. If a consumer group references non-existent messages, those references are removed during recovery.

Recovery Performance Optimization and Checkpoint Integration

Large WAL files can make crash recovery time-intensive, particularly in high-throughput deployments. The system implements several optimization strategies to minimize recovery time while maintaining data consistency guarantees.

Checkpoint-Based Recovery Acceleration The system uses periodic checkpoints to provide **fast recovery paths** that avoid processing the entire WAL history. During normal operation, the Checkpointer component creates state snapshots at regular intervals, capturing complete broker state including all topics, messages, consumer groups, and pending acknowledgments.

During recovery, the process first locates the most recent valid checkpoint, loads the checkpoint state as the recovery baseline, and processes only WAL records created after the checkpoint timestamp. This approach can reduce recovery time from minutes to seconds in systems with large message histories.

Incremental State Rebuilding For systems without recent checkpoints, the recovery process implements **incremental state rebuilding** that provides progress feedback and enables operator intervention if needed. The process reports recovery progress through log messages indicating the percentage of WAL records processed, estimated time remaining based on processing rate, current memory usage and state size, and any errors or inconsistencies encountered during processing.

Memory-Efficient Recovery Streaming To handle WAL files larger than available memory, the recovery process implements **streaming reconstruction** that processes log records in batches without loading the entire log into memory. This approach maintains bounded memory usage during recovery, enables recovery of arbitrarily large logs, and provides consistent memory access patterns that optimize disk I/O performance.

Recovery Failure Handling and Administrative Controls

The recovery process includes comprehensive error handling and administrative controls to manage recovery failures and provide operational flexibility during system restoration.

Recovery Failure Classification and Response The system classifies recovery failures into several categories with different response strategies:

Failure Type	Examples	Recovery Strategy	Operator Action Required
Log corruption	Checksum failures, truncated records	Truncate at corruption boundary	Review lost message impact
Consistency violations	Orphaned references, invalid state	Apply conservative resolution rules	Validate resolved state
Resource exhaustion	Insufficient memory/disk	Enable streaming recovery mode	Increase system resources
Configuration conflicts	Changed retention policies	Use log-recorded configuration	Update current configuration
Checkpoint corruption	Invalid checkpoint data	Fall back to full WAL replay	Remove corrupted checkpoint

Each failure category includes specific diagnostic information in recovery logs, enabling operators to understand the failure cause and take appropriate corrective action.

Administrative Recovery Controls The recovery process exposes several administrative controls that enable operators to customize recovery behavior based on specific operational requirements:

- **Recovery mode selection**: Choose between fast checkpoint recovery vs full WAL replay for complete state validation
- **Corruption handling policy**: Configure whether to halt on corruption detection or continue with truncated state
- **Memory usage limits**: Set maximum memory allocation for recovery operations in resource-constrained environments
- **Progress reporting frequency**: Control recovery log verbosity for monitoring integration

- **State validation level**: Choose between basic consistency checks vs comprehensive cross-component validation

> **Operational Insight**: Recovery design prioritizes **deterministic outcomes** over recovery speed. The system guarantees that recovery always produces consistent state, even if that requires truncating corrupted data or taking longer than optimal. This approach prevents the cascading failures that can occur when message queue systems recover into inconsistent states, particularly in distributed environments where downstream applications depend on message ordering and delivery guarantees.

Component Interaction Patterns and Design Principles

Understanding how components coordinate reveals several key design principles that ensure system reliability and prevent common distributed system problems.

Coordination Patterns and Failure Isolation

The message queue implements several coordination patterns that prevent failures in one component from cascading to others while maintaining system-wide consistency.

Write-Before-Deliver Consistency All message flows follow the fundamental principle that **persistence precedes delivery**. Messages are written to the WAL before any delivery attempts, ensuring durability even if delivery fails. This pattern appears in publication (WAL write before topic enqueuing), consumer assignment (tracking registration before message delivery), and acknowledgment (WAL write before message removal).

Component Responsibility Isolation Each component has clearly defined ownership boundaries that prevent coordination complexity: the Protocol Handler owns connection management and frame parsing, the Topic Manager owns message routing and subscriber fanout, Consumer Groups own work distribution and membership management, the Acknowledgment Tracker owns delivery guarantees and timeouts, and the Persistence Layer owns data durability and recovery.

This isolation enables **independent failure recovery** where components can recover from failures without requiring system-wide restarts or complex distributed coordination protocols.

Explicit State Transition Protocols All state changes follow explicit handoff protocols with clear ownership transfer points. Messages transition from Protocol Handler ownership to Topic Manager ownership through successful `PublishMessage` calls. Consumer assignment transfers message ownership from Group Coordinator to individual consumers through tracked delivery operations. Acknowledgment processing transfers messages from active processing to completed state through explicit ACK/NACK operations.

These explicit transitions prevent the ambiguous ownership scenarios that cause message duplication or loss in less structured message queue implementations.

> **Architecture Insight**: The component interaction design optimizes for **failure recovery simplicity** rather than runtime performance. By maintaining clear component boundaries and explicit state transitions, the system ensures that recovery operations can rebuild consistent state without complex dependency analysis or cross-component synchronization protocols. This design philosophy makes the system easier to debug, test, and operate in production environments.

Implementation Guidance

Technology Recommendations

Component Layer	Simple Option	Advanced Option
State Coordination	Channel-based goroutine coordination	Context-based cancellation with sync primitives
Error Propagation	Return error values with context wrapping	Structured error types with recovery hints
Flow Control	Buffered channels with select statements	Custom semaphore implementations with backpressure
Concurrency Management	sync.RWMutex for component state	Lock-free data structures with atomic operations
Inter-Component Communication	Direct method calls with interface boundaries	Message passing through internal event queues

Recommended File Structure

```
internal/
  flows/
    publish_flow.go          ← orchestrates publication sequence
    consumption_flow.go      ← manages consumer assignment and ack
    rebalancing_flow.go      ← coordinates group membership changes
    recovery_flow.go         ← handles crash recovery operations
    flow_coordinator.go      ← central coordination utilities

  integration/
    component_integration.go ← integration test scenarios
    flow_testing.go          ← flow-specific test utilities
    mock_components.go       ← component mocks for testing
```

This structure separates flow orchestration from individual component logic, making it easier to test complex multi-component interactions and debug coordination issues.

Flow Coordination Infrastructure

```go
// FlowCoordinator provides shared utilities for multi-component operations
type FlowCoordinator struct {
    topicManager    TopicManagerInterface
    groupCoordinator GroupCoordinatorInterface
    ackTracker      AckTrackerInterface
    walLogger       WALLogger
    metrics         *MetricsCollector
}

// PublishFlow orchestrates the complete message publication sequence
func (fc *FlowCoordinator) PublishFlow(ctx context.Context, msg *Message) error {
    // TODO 1: Start distributed trace/logging context for the flow
    // TODO 2: Call walLogger.AppendRecord() with RecordMessagePublished
    // TODO 3: Call topicManager.PublishMessage() to enqueue and route
    // TODO 4: Handle any delivery failures by updating retry counts
    // TODO 5: Update flow metrics and close tracing context
    // Hint: Use ctx.WithTimeout() to ensure flow doesn't hang indefinitely
}

// ConsumptionFlow handles message delivery through consumer group assignment
func (fc *FlowCoordinator) ConsumptionFlow(ctx context.Context,
    topicName, groupName string) ([]*Message, error) {
    // TODO 1: Call groupCoordinator.GetAvailableConsumers() for load info
    // TODO 2: Get pending messages from topicManager.DequeueMessages()
    // TODO 3: For each message, call groupCoordinator.AssignMessage()
    // TODO 4: Call ackTracker.TrackMessage() for each assigned message
    // TODO 5: Return list of successfully assigned messages
    // Hint: Handle partial assignment failures gracefully
}

// RebalancingFlow coordinates consumer group rebalancing operations
func (fc *FlowCoordinator) RebalancingFlow(ctx context.Context,
    groupName string, trigger RebalanceTrigger) error {
    // TODO 1: Call groupCoordinator.BeginRebalance() to start coordination
    // TODO 2: Collect pending messages from ackTracker.GetPendingByGroup()
    // TODO 3: Wait for pending message drain with timeout
```

```
    // TODO 4: Calculate new assignments using assignment strategy
    // TODO 5: Call groupCoordinator.CommitRebalance() atomically
    // TODO 6: Log rebalancing metrics and trigger notifications
    // Hint: Use generation numbers to prevent split-brain conditions
}
```

Recovery Flow Implementation

```go
// RecoveryFlow handles complete system state reconstruction from WAL
func (fc *FlowCoordinator) RecoveryFlow(ctx context.Context) (*BrokerState, error) {
    // TODO 1: Call walLogger.ValidateLogIntegrity() to check for corruption
    // TODO 2: Find latest checkpoint using walLogger.GetLatestCheckpoint()
    // TODO 3: If checkpoint exists, load it as baseline state
    // TODO 4: Process WAL records since checkpoint chronologically
    // TODO 5: Rebuild topic state by replaying RecordMessagePublished entries
    // TODO 6: Rebuild consumer groups (mark all consumers as disconnected)
    // TODO 7: Identify orphaned messages and return them to topic queues
    // TODO 8: Validate cross-component consistency
    // TODO 9: Return consolidated BrokerState for system restart
    // Hint: Use streaming WAL reader to handle large logs efficiently
}

// validateConsistency performs cross-component state validation
func (fc *FlowCoordinator) validateConsistency(state *BrokerState) []ConsistencyError {
    // TODO 1: Check that all acked messages exist in topics or are completed
    // TODO 2: Verify consumer group assignments reference valid messages
    // TODO 3: Ensure all topic subscriptions reference connected consumers
    // TODO 4: Validate pending message deadlines are in the future
    // TODO 5: Return list of detected inconsistencies for logging
}
```

Integration Testing Framework

```go
// FlowIntegrationTest provides utilities for testing multi-component flows
type FlowIntegrationTest struct {
    coordinator *FlowCoordinator
    mockWAL     *MockWALLogger
    mockTopic   *MockTopicManager
    mockGroups  *MockGroupCoordinator
    mockAck     *MockAckTracker
}

// TestPublishFlowSuccess verifies normal publication flow
func (fit *FlowIntegrationTest) TestPublishFlowSuccess() {
    // TODO 1: Create test message with known ID and payload
    // TODO 2: Configure mocks to simulate successful WAL write
    // TODO 3: Configure topic manager mock to accept message
    // TODO 4: Call PublishFlow() and verify no errors
    // TODO 5: Assert WAL write was called with correct record type
    // TODO 6: Assert topic manager received the message
    // TODO 7: Verify metrics were updated correctly
}

// TestRecoveryFlowWithCorruption verifies recovery handles log corruption
func (fit *FlowIntegrationTest) TestRecoveryFlowWithCorruption() {
    // TODO 1: Create WAL with valid records followed by corrupted data
    // TODO 2: Configure mock to return corruption at specific offset
    // TODO 3: Call RecoveryFlow() and verify it doesn't fail
    // TODO 4: Assert recovery truncated at corruption boundary
    // TODO 5: Verify recovered state contains pre-corruption data
    // TODO 6: Check that appropriate corruption warnings were logged
}
```

Debugging and Monitoring Integration

```go
// FlowTracing provides observability into multi-component operations
type FlowTracing struct {
    logger      *zap.Logger
    tracer      opentracing.Tracer
    metrics     *prometheus.Registry
}

// TracePublishFlow creates observability context for publication
func (ft *FlowTracing) TracePublishFlow(msg *Message) (context.Context, func()) {
    // TODO 1: Start distributed trace span with message ID
    // TODO 2: Add trace tags for topic, producer, payload size
    // TODO 3: Start flow duration timer
    // TODO 4: Return context and cleanup function
}

// LogFlowError records structured error information for debugging
func (ft *FlowTracing) LogFlowError(flowType string, err error, context map[string]interface{}) {
    // TODO 1: Extract error root cause and component
    // TODO 2: Add flow context (message IDs, consumer IDs, etc.)
    // TODO 3: Log structured error with appropriate level
    // TODO 4: Update error rate metrics for the flow type
}
```

Milestone Checkpoints

After Milestone 1 (Basic Protocol + Topics):

- Verify publish flow works: `curl -X POST localhost:8080/publish -d '{"topic":"test","message":"hello"}'`
- Check topic creation: Messages should create topics automatically
- Confirm WAL persistence: Restart broker and verify messages survive

After Milestone 2 (Consumer Groups + ACK):

- Test consumer group assignment: Start multiple consumers, verify round-robin
- Verify acknowledgment flow: Send ACK, confirm message removal
- Test rebalancing: Add/remove consumers, verify work redistribution

After Milestone 3 (Persistence + Backpressure):

- Test crash recovery: Kill broker mid-operation, restart, verify state restoration
- Confirm backpressure: Overload consumers, verify publisher throttling
- Validate log compaction: Let system run, verify old entries removed

After Milestone 4 (DLQ + Monitoring):

- Test poison message handling: Send repeatedly failing message, verify DLQ routing

- Verify monitoring API: Check `/metrics` endpoint for flow statistics
- Test message replay: Move DLQ message back to topic, verify reprocessing

Common Integration Pitfalls

⚠️ **Pitfall: Race Conditions During Component Handoff** Components often exchange ownership of messages through async operations, creating race conditions where the same message might be processed by multiple components simultaneously. This happens when acknowledgment processing overlaps with timeout-based redelivery, or when consumer group rebalancing occurs during message assignment.

Fix: Use generation numbers and explicit ownership transfer protocols. Every message operation should include a generation/version number that invalidates concurrent operations from different coordination contexts.

⚠️ **Pitfall: Inconsistent Error Handling Across Flow Boundaries** Different components may handle the same error condition differently, leading to inconsistent system behavior. For example, network timeouts might cause the Protocol Handler to close connections while the Acknowledgment Tracker continues trying to deliver messages to disconnected consumers.

Fix: Define explicit error classification and handling policies in the `FlowCoordinator`. Each error type should have a documented handling strategy that all components follow consistently.

⚠️ **Pitfall: Memory Leaks During Flow Failure Recovery** When multi-component flows fail partway through execution, cleanup often becomes complex because multiple components hold references to the same data. Failed publish flows might leave messages in topic queues, WAL entries, and acknowledgment tracking maps simultaneously.

Fix: Implement explicit cleanup protocols with reference counting or resource ownership tracking. Each component should be able to clean up its portion of failed flows without requiring coordination with other components.

Error Handling and Edge Cases

> **Milestone(s):** All milestones (1-4) — comprehensive error handling strategies that ensure system reliability across all components from basic pub/sub through persistence and monitoring

Mental Model: The Hospital Emergency Response System

Think of our message queue's error handling like a hospital's emergency response system. Just as hospitals have different protocols for various medical emergencies (cardiac arrest, trauma, poisoning), our message queue must have specific responses to different failure modes (network disconnects, consumer crashes, disk failures). The hospital doesn't just detect problems — it has triage procedures to classify severity, escalation protocols to get the right specialists involved, and recovery procedures to restore normal operations. Similarly, our system needs detection mechanisms (like a patient's vital signs monitors), classification systems (triage nurses determining urgency), and coordinated response procedures (surgical teams following established protocols).

The key insight is that failures aren't binary — they exist on a spectrum from minor hiccups to catastrophic system failures. Just as hospitals have different response levels (observation, intensive care, emergency surgery), our message queue needs graduated responses that match the severity and type of failure. A single dropped TCP packet requires different handling than a consumer process crash, which requires different handling than complete broker failure.

Failure Mode Classification

The message queue system faces three primary categories of failures, each requiring distinct detection mechanisms and recovery strategies. Understanding these failure patterns helps us build appropriate resilience into each system component.

Network Failure Scenarios

Network failures represent the most common class of problems in distributed messaging systems. These failures can manifest as complete connection loss, intermittent packet drops, or degraded network performance that mimics other system problems.

Failure Type	Manifestation	Duration	Impact Scope
Connection Drop	TCP connection reset or timeout	Immediate	Single client
Packet Loss	Retransmissions, delayed responses	Seconds to minutes	Connection-specific
Network Partition	Complete isolation of broker/client	Minutes to hours	Regional
Bandwidth Saturation	Increased latency, timeouts	Variable	System-wide
DNS Resolution Failure	Connection establishment fails	Minutes	Client-specific

Connection drops occur when the underlying TCP connection between a client and broker terminates unexpectedly. This can happen due to client process crashes, network equipment failures, or firewall rules. The broker must detect these drops quickly to avoid delivering messages to dead connections and to trigger consumer group rebalancing if the disconnected client was part of a consumer group.

Packet loss manifests as increased latency and potential timeouts during message exchange. Unlike connection drops, the connection remains active but becomes unreliable. The broker's protocol handler must distinguish between slow networks and actual processing failures to avoid premature timeouts.

Network partitions represent the most severe network failure mode, where the broker becomes completely isolated from some or all clients. During partitions, the broker must decide whether to continue serving connected clients or enter a read-only mode to prevent split-brain scenarios.

Consumer Process Failures

Consumer failures occur when message processing applications crash, hang, or become unresponsive. These failures are particularly challenging because they can leave messages in an uncertain state — we don't know if the consumer processed the message before failing.

Failure Type	Detection Method	Message Impact	Recovery Action
Process Crash	Heartbeat timeout	Pending messages lost	Reassign to other consumers
Infinite Loop	Processing timeout	Head-of-line blocking	Kill consumer, reassign messages
Memory Exhaustion	Heartbeat degradation	Slow processing	Throttle assignments
Poison Message	Repeated failures	Infinite retry loop	Move to dead letter queue
Consumer Bug	NACK with error	Processing failure	Retry with different consumer

Process crashes are the cleanest failure mode because the operating system immediately closes all network connections, allowing the broker to detect the failure quickly through connection monitoring. However, any messages that were delivered to the crashed consumer need to be redelivered to other consumers.

Infinite loops or deadlocks in consumer code are more insidious because the consumer appears healthy (maintains heartbeat) but never acknowledges messages. The broker must use processing timeouts to detect these scenarios and forcibly reassign stuck messages.

Poison messages represent a particularly challenging failure mode where specific message content consistently causes consumer failures. Without proper detection, poison messages can create infinite retry loops that consume system resources and prevent progress on other messages.

Broker System Failures

Broker failures affect the entire messaging system and require the most sophisticated recovery mechanisms. These failures can range from temporary resource exhaustion to complete system crashes that require state reconstruction.

Failure Type	Trigger Condition	System Impact	Recovery Complexity
Memory Exhaustion	High message volume	Performance degradation	Moderate
Disk Full	Log file growth	Write failures	High
Broker Process Crash	Software bug, OOM killer	Complete outage	High
Log Corruption	Disk errors, power failure	Data integrity issues	Very High
Configuration Error	Invalid settings	Startup failure	Low

Memory exhaustion typically occurs when message ingestion rate exceeds processing rate for extended periods. The broker's in-memory queues grow until system memory is exhausted, triggering either performance degradation or process termination by the operating system's OOM killer.

Disk full scenarios happen when the Write-Ahead Log grows faster than log compaction can reclaim space. This causes all write operations to fail, effectively making the broker read-only until disk space is recovered.

Complete broker crashes require full state reconstruction from the WAL. The complexity depends on how much state was lost and whether the WAL itself was corrupted during the failure.

> **Design Insight: Failure Hierarchy**
>
> The key architectural principle is that failures form a hierarchy where higher-level failures subsume lower-level concerns. A broker crash makes individual consumer failures irrelevant because the entire system needs recovery. This hierarchy guides our error handling priorities — we handle the most severe failures first and use their recovery mechanisms to address subsidiary problems.

Error Detection Mechanisms

Effective error detection requires multiple overlapping mechanisms because no single approach can catch all failure modes. Our detection strategy combines active monitoring (heartbeats, health checks) with passive monitoring (timeout tracking, error rate analysis) to provide comprehensive coverage.

Timeout-Based Detection

Timeouts form the foundation of failure detection in asynchronous messaging systems. However, timeout values require careful tuning because they represent a fundamental trade-off between failure detection speed and false positive rates.

Timeout Type	Default Value	Adjustable	False Positive Risk
Connection Heartbeat	30 seconds	Yes	Low
Message Acknowledgment	60 seconds	Per-topic	Medium
Processing Timeout	300 seconds	Per-consumer	High
Rebalance Timeout	120 seconds	Global	Medium
Health Check	15 seconds	Yes	Low

Connection heartbeats use a simple ping-pong mechanism where clients must send a heartbeat frame within the configured interval. The broker tracks the last heartbeat timestamp for each connection and considers connections dead if they exceed the threshold. This timeout should be short enough to detect failures quickly but long enough to account for normal network latency variations.

Message acknowledgment timeouts detect consumers that receive messages but fail to process them. This timeout begins when a message is delivered to a consumer and expires if no ACK or NACK is received within the deadline. The challenge is setting values that account for legitimate processing time while detecting actual failures.

Processing timeouts provide a secondary detection mechanism for consumers that receive messages but never respond. This timeout is typically longer than acknowledgment timeout because it accounts for complex business logic execution time.

Heartbeat Monitoring System

The heartbeat monitoring system provides active failure detection by requiring periodic liveness signals from all system participants. This system must balance detection accuracy with network overhead.

Component	Signal Type	Frequency	Payload	Action on Miss
Consumer Client	HEARTBEAT frame	10 seconds	Consumer status	Mark as suspicious
Connection	TCP keepalive	30 seconds	None	Close connection
Consumer Group	Group membership	20 seconds	Member list	Trigger rebalancing
Health Check	HTTP ping	15 seconds	System metrics	Update health status

The `HeartbeatTracker` component manages all heartbeat monitoring for the broker system. It maintains heartbeat state for each registered consumer and provides automated cleanup of failed participants.

Field	Type	Description
heartbeats	map[string]*ConsumerHeartbeat	Active heartbeat tracking per consumer
timeoutThreshold	time.Duration	Maximum time between heartbeats before marking as failed
cleanupInterval	time.Duration	How often to scan for expired heartbeats
groupCoordinator	GroupCoordinatorInterface	Notifies group coordinator of consumer failures
ackTracker	AckTrackerInterface	Retrieves pending messages for failed consumers
cleanupTicker	*time.Ticker	Background timer for periodic cleanup
shutdownCh	chan struct{}	Graceful shutdown coordination

Each consumer maintains detailed heartbeat metadata that enables sophisticated failure detection beyond simple timeout monitoring.

Field	Type	Description
ConsumerID	string	Unique identifier for the consumer
LastHeartbeat	time.Time	Timestamp of most recent heartbeat signal
ConsecutiveMisses	int	Count of sequential missed heartbeats
ProcessingStatus	string	Current processing state (idle, busy, error)
PendingMessageCount	int	Number of unacknowledged messages
AverageProcessingTime	time.Duration	Moving average of message processing time
ErrorRate	float64	Recent error percentage for this consumer

The heartbeat system implements graduated failure detection where consumers progress through suspicion levels before being declared dead. This prevents false positives caused by temporary network hiccups.

1. **Healthy State**: Consumer sends regular heartbeats within expected intervals
2. **Warning State**: Consumer misses one heartbeat but is still considered active
3. **Suspicious State**: Consumer misses multiple consecutive heartbeats, triggering enhanced monitoring
4. **Failed State**: Consumer exceeds timeout threshold, triggering cleanup and reassignment

Health Check Algorithms

Health checks provide point-in-time system health assessment through active probing of system components. Unlike passive monitoring that waits for problems to manifest, health checks actively verify that components can perform their essential functions.

The health check system evaluates multiple dimensions of system health to provide a comprehensive status assessment:

Health Dimension	Check Method	Frequency	Failure Threshold
Connection Pool	Active connection count	30 seconds	>95% utilization
Memory Usage	Process memory statistics	15 seconds	>90% of limit
Disk Space	Available storage check	60 seconds	<10% free space
Message Queue Depth	Pending message count	10 seconds	>1000 pending
Consumer Lag	Processing delay metrics	30 seconds	>5 minutes lag

The health check algorithm follows a standardized assessment procedure:

1. **Collect Metrics**: Gather current measurements for each health dimension
2. **Evaluate Thresholds**: Compare metrics against configured warning and critical thresholds
3. **Calculate Health Score**: Combine individual dimension scores into overall health rating
4. **Generate Health Report**: Create detailed status report with specific recommendations
5. **Trigger Alerts**: Send notifications for any dimensions exceeding critical thresholds
6. **Update Health Status**: Store results for trend analysis and dashboard display

Health checks also perform functional validation by executing representative operations:

- **Connection Test**: Establish new connection and perform basic protocol handshake
- **Publish Test**: Send test message to dedicated health check topic
- **Subscribe Test**: Verify delivery of test messages to health check consumer
- **Persistence Test**: Write test record to WAL and verify successful read-back
- **Recovery Test**: Validate ability to read recent WAL entries and reconstruct state

> **Design Insight: Detection Layering**
>
> Effective failure detection requires multiple detection layers with different response times and accuracy characteristics. Fast but potentially inaccurate detection (heartbeat timeouts) triggers initial response, while slower but more accurate detection (health checks) confirms failures and guides recovery strategies. This layering prevents both slow response to real failures and excessive false alarms.

Recovery Strategies

Recovery strategies define how the system responds to detected failures and restores normal operation. The choice of recovery strategy depends on failure scope, data consistency requirements, and availability priorities.

Graceful Degradation Approaches

Graceful degradation allows the system to continue operating at reduced capacity when components fail, rather than experiencing complete outages. This approach prioritizes availability over full functionality.

Degradation Level	Trigger Condition	Operational Impact	Available Functions
Consumer Throttling	High error rates	Reduced throughput	Publish, limited consume
Read-Only Mode	Disk space low	No new messages	Message delivery only
Emergency Mode	Memory critical	Essential only	Existing connections only
Maintenance Mode	Manual trigger	Planned degradation	Administrative access

Consumer throttling activates when error rates exceed acceptable thresholds. The system reduces message assignment rate to problematic consumers while maintaining full service for healthy consumers. This prevents cascading failures while allowing problem consumers time to recover.

Read-only mode triggers when storage resources approach exhaustion. The broker stops accepting new messages but continues delivering existing messages from memory queues. This preserves system availability for message consumption while preventing data loss from storage failures.

Emergency mode represents the most severe degradation level where the broker maintains only essential functions. New connections are rejected, message publishing is suspended, and only existing connections continue receiving messages from memory. This mode preserves system stability during resource crises.

The degradation level calculation follows a systematic assessment process:

1. **Resource Assessment**: Measure current utilization across all system resources
2. **Threshold Evaluation**: Compare measurements against configured degradation triggers
3. **Impact Analysis**: Determine which functions can be safely disabled at current resource levels
4. **Transition Planning**: Calculate safe transition path to target degradation level
5. **Notification**: Alert connected clients about impending service level changes
6. **Graceful Transition**: Gradually reduce functionality while maintaining data consistency

Automatic Healing Mechanisms

Automatic healing enables the system to recover from failures without manual intervention. These mechanisms must balance aggressive recovery with system stability to avoid flapping behaviors.

Healing Mechanism	Trigger	Recovery Action	Backoff Strategy
Consumer Restart	Process failure	Launch replacement	Exponential backoff
Connection Retry	Network failure	Reconnect with backoff	Linear backoff
Message Reassignment	Timeout	Move to healthy consumer	Immediate
Log Compaction	Disk pressure	Background cleanup	Schedule-based
Memory Cleanup	Memory pressure	Garbage collection	Progressive

Consumer restart mechanisms detect failed consumer processes and automatically launch replacement instances. This requires integration with process supervision systems and careful handling of consumer group membership during transitions.

Connection retry mechanisms handle transient network failures by automatically re-establishing connections with appropriate backoff delays. The retry algorithm must distinguish between transient and permanent failures to avoid infinite retry loops.

Message reassignment provides rapid recovery from consumer failures by moving pending messages to healthy consumers. This mechanism must coordinate with consumer group rebalancing to ensure consistent message assignment.

The automatic healing process follows a structured recovery workflow:

1. **Failure Classification**: Determine the type and scope of detected failure
2. **Recovery Eligibility**: Verify that automatic recovery is appropriate for this failure type
3. **Resource Validation**: Confirm sufficient resources are available for recovery operations
4. **Recovery Strategy Selection**: Choose the most appropriate healing mechanism based on failure characteristics
5. **Recovery Execution**: Implement recovery actions with appropriate error handling
6. **Recovery Verification**: Confirm that recovery actions successfully restored functionality
7. **Monitoring Enhancement**: Increase monitoring frequency for recently recovered components

Consistency Resolution Protocols

When failures occur during message processing, the system may enter inconsistent states where different components have conflicting views of message status. Consistency resolution protocols restore system coherence through systematic state reconciliation.

Inconsistency Type	Detection Method	Resolution Strategy	Consistency Level
Duplicate Messages	Message ID tracking	Deduplication	Exactly-once
Lost Acknowledgments	Timeout scanning	Conservative redelivery	At-least-once
Split-Brain Groups	Generation numbers	Leader election	Strong consistency
Orphaned Messages	Recovery scanning	State reconstruction	Eventual consistency

The consistency resolution algorithm prioritizes data preservation over operational efficiency. When in doubt, the system assumes messages were not processed and triggers redelivery rather than risking message loss.

Message deduplication handles scenarios where network failures cause duplicate message delivery. The system maintains a sliding window of recent message IDs and discards duplicates within the window period.

Lost acknowledgment resolution occurs when consumers process messages but acknowledgments are lost due to network failures. The system uses conservative timeouts and redelivery policies to ensure messages are not lost even if this occasionally causes duplicate processing.

Consumer group split-brain resolution uses generation numbers to detect and resolve scenarios where network partitions cause multiple consumers to claim the same group membership. The resolution process:

1. **Partition Detection**: Identify conflicting group membership claims through generation number mismatches
2. **State Collection**: Gather group state from all accessible consumers and recent WAL entries
3. **Authoritative State Selection**: Choose the most recent consistent group configuration based on generation numbers and timestamps
4. **Conflict Resolution**: Reassign messages that were processed by multiple consumers during the split-brain period
5. **Group Reconstruction**: Rebuild consumer group with unified membership and clear message assignments
6. **Validation**: Verify that all consumers agree on the resolved group configuration

> **Architecture Decision: Conservative Consistency**
>
> - **Context**: Message processing failures can leave the system in inconsistent states where message status is uncertain
> - **Options Considered**:
>   - Optimistic approach (assume success unless proven otherwise)
>   - Conservative approach (assume failure unless proven otherwise)
>   - Consensus-based approach (require majority agreement on message status)
> - **Decision**: Conservative approach with at-least-once delivery semantics
> - **Rationale**: Message loss is typically more damaging than duplicate processing for most applications. Conservative redelivery ensures no messages are lost even if some are processed multiple times.
> - **Consequences**: Simplified recovery logic and stronger durability guarantees, but applications must handle duplicate messages idempotently

Error Handling State Machine

The system implements a comprehensive state machine that governs error handling transitions and ensures appropriate responses to different failure scenarios.

Current State	Error Event	Next State	Recovery Action
Normal	Consumer Timeout	Degraded	Reassign messages, continue service
Normal	Disk Full	Emergency	Enter read-only mode, alert operators
Normal	Memory Warning	Throttled	Reduce accept rate, increase cleanup
Degraded	Consumer Recovery	Normal	Resume full message assignment
Degraded	Additional Failures	Emergency	Further reduce functionality
Emergency	Resource Recovery	Degraded	Gradually restore functions
Emergency	Operator Intervention	Maintenance	Planned recovery coordination

The error handling state machine ensures that system responses escalate appropriately as failure conditions worsen and de-escalate safely as conditions improve. State transitions include hysteresis to prevent rapid oscillation between states due to temporary fluctuations in system health.

Each state transition triggers specific recovery protocols and notification procedures to ensure that failures are addressed systematically and stakeholders are kept informed of system status changes.

⚠️ Pitfall: Thundering Herd During Recovery

When the broker recovers from a failure, all disconnected clients may attempt to reconnect simultaneously, overwhelming the recovering system. This thundering herd effect can cause the broker to fail again just as it's trying to recover.

Why it's wrong: The recovery process assumes the system can handle normal connection load immediately upon restart, but the system may still be rebuilding state from the WAL or performing other resource-intensive recovery operations.

How to fix: Implement connection admission control during recovery phases. Use exponential backoff for client reconnection attempts and limit the rate of new connection acceptance while the broker completes its recovery process. Clients should also implement jittered retry delays to spread reconnection attempts over time.

⚠️ Pitfall: Cascading Timeout Failures

When one component becomes slow due to resource pressure, it can cause timeout failures in dependent components, which then trigger their own recovery procedures, creating a cascade of failures throughout the system.

Why it's wrong: Aggressive timeout values that work well under normal conditions become liability during stress conditions. Each timeout-triggered recovery action consumes additional resources, making the original resource pressure worse.

How to fix: Implement adaptive timeout mechanisms that increase timeout values when system stress is detected. Use circuit breaker patterns to prevent failed calls from propagating through the system. Monitor timeout success rates and automatically adjust timeout values based on recent system performance.

Implementation Guidance

The error handling system requires careful integration across all message queue components. This guidance provides practical approaches for implementing robust error detection and recovery mechanisms.

Technology Recommendations

Component	Simple Option	Advanced Option
Health Monitoring	HTTP endpoints with JSON status	Prometheus metrics with Grafana
Error Tracking	Structured logging with levels	Distributed tracing with OpenTelemetry
Timeout Management	Standard library timers	Custom timeout scheduler with priorities
Circuit Breakers	Simple counter-based	Hystrix-style with statistics
Process Supervision	Manual restart scripts	systemd or Docker health checks

Recommended Module Structure

```
internal/
  errorhandling/
    detector/
      timeout_detector.go      ← timeout-based failure detection
      heartbeat_tracker.go     ← heartbeat monitoring system
      health_checker.go        ← active health verification
    recovery/
      auto_healer.go          ← automatic failure recovery
      degradation_manager.go   ← graceful degradation control
      consistency_resolver.go  ← state consistency restoration
    circuit/
      breaker.go              ← circuit breaker implementation
      adaptive_timeouts.go     ← dynamic timeout adjustment
  monitoring/
    metrics_collector.go       ← system metrics gathering
    alert_manager.go          ← alerting and notification
```

GO

Infrastructure Starter Code

```go
// Package errorhandling provides comprehensive failure detection and recovery
// mechanisms for the message queue system.
package errorhandling

import (
    "context"
    "sync"
    "time"
)

// SystemHealth represents overall system health status
type SystemHealth struct {
    Status          HealthStatus          `json:"status"`
    LastCheck       time.Time             `json:"lastCheck"`
    ComponentHealth map[string]ComponentHealth `json:"components"`
    Recommendations []string              `json:"recommendations"`
    UptimeSeconds   int64                 `json:"uptimeSeconds"`
}

type HealthStatus string

const (
    HealthHealthy    HealthStatus = "healthy"
    HealthDegraded   HealthStatus = "degraded"
    HealthCritical   HealthStatus = "critical"
    HealthUnknown    HealthStatus = "unknown"
)

type ComponentHealth struct {
    Status      HealthStatus `json:"status"`
    LastCheck   time.Time    `json:"lastCheck"`
    ErrorCount  int64        `json:"errorCount"`
    Latency     time.Duration `json:"latency"`
    Message     string       `json:"message,omitempty"`
}

// TimeoutConfig defines timeout settings for various operations
type TimeoutConfig struct {
    ConnectionHeartbeat time.Duration
```

```
    MessageAck         time.Duration
    ProcessingTimeout  time.Duration
    RebalanceTimeout   time.Duration
    HealthCheck        time.Duration
    AdaptiveEnabled    bool
    MinTimeout         time.Duration
    MaxTimeout         time.Duration
}

// DefaultTimeoutConfig returns conservative timeout settings
func DefaultTimeoutConfig() TimeoutConfig {
    return TimeoutConfig{
        ConnectionHeartbeat: 30 * time.Second,
        MessageAck:         60 * time.Second,
        ProcessingTimeout:  300 * time.Second,
        RebalanceTimeout:   120 * time.Second,
        HealthCheck:        15 * time.Second,
        AdaptiveEnabled:    true,
        MinTimeout:         5 * time.Second,
        MaxTimeout:         600 * time.Second,
    }
}

// FailureClassifier categorizes failures for appropriate response selection
type FailureClassifier struct {
    patterns map[string]FailurePattern
    mutex    sync.RWMutex
}

type FailurePattern struct {
    Type        FailureType
    Severity    FailureSeverity
    Recovery    RecoveryStrategy
    Retryable   bool
}

type FailureType string
```

```go
type FailureSeverity string

type RecoveryStrategy string


const (

    FailureTypeNetwork   FailureType = "network"

    FailureTypeConsumer  FailureType = "consumer"

    FailureTypeBroker    FailureType = "broker"

    FailureTypeStorage   FailureType = "storage"

)


const (

    SeverityLow      FailureSeverity = "low"

    SeverityMedium   FailureSeverity = "medium"

    SeverityHigh     FailureSeverity = "high"

    SeverityCritical FailureSeverity = "critical"

)


const (

    RecoveryReconnect    RecoveryStrategy = "reconnect"

    RecoveryReassign     RecoveryStrategy = "reassign"

    RecoveryRestart      RecoveryStrategy = "restart"

    RecoveryDegrade      RecoveryStrategy = "degrade"

    RecoveryEmergency    RecoveryStrategy = "emergency"

)


// NewFailureClassifier creates a failure classifier with default patterns

func NewFailureClassifier() *FailureClassifier {

    fc := &FailureClassifier{

        patterns: make(map[string]FailurePattern),

    }



    // Initialize default failure patterns

    fc.patterns["connection_reset"] = FailurePattern{

        Type:        FailureTypeNetwork,

        Severity:    SeverityLow,

        Recovery:    RecoveryReconnect,

        Retryable:   true,
```

```
    }

    fc.patterns["consumer_timeout"] = FailurePattern{
        Type:        FailureTypeConsumer,
        Severity:    SeverityMedium,
        Recovery:    RecoveryReassign,
        Retryable:   true,
    }

    fc.patterns["disk_full"] = FailurePattern{
        Type:        FailureTypeStorage,
        Severity:    SeverityCritical,
        Recovery:    RecoveryEmergency,
        Retryable:   false,
    }

    return fc
}

// CircuitBreaker implements circuit breaker pattern for failure isolation
type CircuitBreaker struct {
    maxFailures    int
    resetTimeout   time.Duration
    currentState   CircuitState
    failureCount   int
    lastFailureTime time.Time
    mutex          sync.RWMutex
}

type CircuitState string

const (
    CircuitClosed    CircuitState = "closed"
    CircuitOpen      CircuitState = "open"
    CircuitHalfOpen  CircuitState = "half-open"
)
```

```go
// NewCircuitBreaker creates a circuit breaker with specified parameters
func NewCircuitBreaker(maxFailures int, resetTimeout time.Duration) *CircuitBreaker {
    return &CircuitBreaker{
        maxFailures:  maxFailures,
        resetTimeout: resetTimeout,
        currentState: CircuitClosed,
    }
}

// Execute runs the provided function with circuit breaker protection
func (cb *CircuitBreaker) Execute(fn func() error) error {
    if !cb.allowRequest() {
        return ErrCircuitOpen
    }

    err := fn()
    cb.recordResult(err)
    return err
}

var ErrCircuitOpen = errors.New("circuit breaker is open")
```

Core Logic Skeleton Code

```go
// HeartbeatTracker manages consumer liveness monitoring and failure detection
type HeartbeatTracker struct {
    heartbeats       map[string]*ConsumerHeartbeat
    timeoutThreshold time.Duration
    cleanupInterval  time.Duration
    groupCoordinator GroupCoordinatorInterface
    ackTracker       AckTrackerInterface
    cleanupTicker    *time.Ticker
    shutdownCh       chan struct{}
    mutex            sync.RWMutex
}

// ProcessHeartbeat updates the heartbeat timestamp for a consumer and evaluates health status
func (ht *HeartbeatTracker) ProcessHeartbeat(consumerID string, status string) error {
    // TODO 1: Acquire write lock to update heartbeat state safely
    // TODO 2: Retrieve or create ConsumerHeartbeat entry for the consumer
    // TODO 3: Update LastHeartbeat timestamp to current time
    // TODO 4: Reset ConsecutiveMisses counter since we received a heartbeat
    // TODO 5: Update ProcessingStatus with the provided status information
    // TODO 6: If consumer was previously marked as failed, trigger recovery procedures
    // TODO 7: Log heartbeat reception for debugging and monitoring
    // Hint: Use time.Now() for current timestamp, consider atomic operations for counters
}

// CheckForTimeouts scans all tracked consumers and identifies those exceeding timeout thresholds
func (ht *HeartbeatTracker) CheckForTimeouts() []string {
    // TODO 1: Acquire read lock to safely access heartbeat data
    // TODO 2: Calculate current time and timeout threshold
    // TODO 3: Iterate through all tracked consumers in heartbeats map
    // TODO 4: For each consumer, compare LastHeartbeat against timeout threshold
    // TODO 5: If timeout exceeded, increment ConsecutiveMisses counter
    // TODO 6: If ConsecutiveMisses exceeds failure threshold, mark as failed
    // TODO 7: Collect all failed consumer IDs into result slice
    // TODO 8: Return slice of failed consumer IDs for cleanup processing
    // Hint: Use time.Since() to calculate elapsed time, be careful about time zones
}
```

```go
// CleanupUnhealthyConsumer removes a failed consumer and reassigns its pending messages
func (ht *HeartbeatTracker) CleanupUnhealthyConsumer(consumerID string) error {
    // TODO 1: Acquire write lock to modify tracking state
    // TODO 2: Retrieve pending messages for the consumer from ackTracker
    // TODO 3: Remove consumer from heartbeat tracking map
    // TODO 4: Notify group coordinator about consumer failure for rebalancing
    // TODO 5: Reassign pending messages to other healthy consumers in the group
    // TODO 6: Log consumer cleanup action with details about reassigned messages
    // TODO 7: Update metrics to reflect consumer failure and cleanup completion
    // Hint: Coordinate with GroupCoordinator.OnConsumerFailure() and AckTracker.CleanupConsumer()
}

// AutoHealer implements automatic recovery mechanisms for various failure types
type AutoHealer struct {
    classifier       *FailureClassifier
    recoveryStrategies map[RecoveryStrategy]RecoveryHandler
    backoffStrategy  BackoffStrategy
    maxRetries       int
    retryState       map[string]*RetryState
    mutex           sync.RWMutex
}

// AttemptRecovery tries to automatically recover from a detected failure
func (ah *AutoHealer) AttemptRecovery(failure FailureEvent) error {
    // TODO 1: Use failure classifier to determine failure type and appropriate recovery strategy
    // TODO 2: Check retry state to see if this failure has been attempted before
    // TODO 3: If max retries exceeded, escalate to manual intervention
    // TODO 4: Calculate backoff delay based on previous retry attempts
    // TODO 5: Select appropriate recovery handler based on failure classification
    // TODO 6: Execute recovery handler with timeout and error handling
    // TODO 7: Update retry state based on recovery success or failure
    // TODO 8: Schedule verification check to ensure recovery was successful
    // Hint: Use context.WithTimeout for recovery operations, implement exponential backoff
}

// DegradationManager controls graceful degradation of system functionality
type DegradationManager struct {
```

```
    currentLevel     DegradationLevel
    thresholds       map[string]float64
    functions        map[string]bool
    resourceMonitor  ResourceMonitor
    transitionDelay  time.Duration
    mutex           sync.RWMutex
}

// EvaluateDegradationNeed assesses current system state and determines if degradation is necessary
func (dm *DegradationManager) EvaluateDegradationNeed() DegradationLevel {
    // TODO 1: Collect current resource utilization metrics from resource monitor
    // TODO 2: Compare each metric against configured degradation thresholds
    // TODO 3: Calculate overall system stress score based on multiple metrics
    // TODO 4: Determine target degradation level based on stress score
    // TODO 5: Consider hysteresis to prevent rapid oscillation between levels
    // TODO 6: Return recommended degradation level for system transition
    // Hint: Use weighted scoring for different metrics, implement hysteresis with separate thresholds
}

// TransitionToDegradationLevel safely changes system degradation level
func (dm *DegradationManager) TransitionToDegradationLevel(targetLevel DegradationLevel) error {
    // TODO 1: Validate that target degradation level is valid and achievable
    // TODO 2: Calculate transition path from current level to target level
    // TODO 3: Notify all affected components about impending degradation change
    // TODO 4: Disable functions that are not available at target degradation level
    // TODO 5: Wait for transition delay to allow graceful client adaptation
    // TODO 6: Update current degradation level and enable remaining functions
    // TODO 7: Verify successful transition and log degradation change
    // Hint: Use graceful shutdown for disabled functions, notify clients via protocol messages
}
```

Milestone Checkpoint

After implementing error handling mechanisms:

Milestone 1 Checkpoint: Connection failure handling

- Start broker and connect multiple clients
- Kill client processes unexpectedly
- Verify: Broker detects disconnections within heartbeat timeout and cleans up resources
- Expected: Connection count decreases, no memory leaks from abandoned connections

Milestone 2 Checkpoint: Consumer failure recovery

- Create consumer group with multiple members
- Send messages and kill one consumer during processing
- Verify: Pending messages reassigned to other consumers, group rebalances automatically
- Expected: All messages eventually acknowledged, failed consumer removed from group

Milestone 3 Checkpoint: Persistence failure handling

- Fill disk to capacity during message ingestion
- Verify: Broker enters read-only mode, existing messages still delivered
- Expected: New publishes rejected with appropriate error, no data corruption

Milestone 4 Checkpoint: Dead letter queue handling

- Send poison messages that cause consistent consumer failures
- Verify: Messages moved to DLQ after max retry attempts
- Expected: Consumer group continues processing other messages, poison messages isolated

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Consumers marked failed but still responsive	Heartbeat timeout too aggressive	Check network latency, consumer processing time	Increase heartbeat timeout, implement adaptive timeouts
Messages redelivered multiple times	Acknowledgment timeout too short	Monitor consumer processing duration	Increase ack timeout or implement processing status reporting
System oscillates between degradation levels	Hysteresis thresholds too close	Graph degradation level over time	Increase gap between up/down thresholds, add stabilization delay
Recovery attempts never succeed	Failure classification incorrect	Check failure classifier patterns	Update failure patterns, implement manual override capability
Circuit breaker stuck open	Reset timeout too long or failures continue	Monitor circuit breaker state transitions	Adjust reset timeout, fix underlying failure cause

Testing Strategy

> **Milestone(s):** All milestones (1-4) — comprehensive testing approaches that verify functionality and catch regressions throughout the entire development process

Mental Model: The Quality Assurance Laboratory

Think of testing a message queue like running a comprehensive quality control laboratory for a pharmaceutical company. Just as pharmaceutical testing has multiple stages — from testing individual chemical compounds in isolation, to testing drug interactions in controlled environments, to running clinical trials with real patients under chaotic real-world conditions — our message queue testing strategy has corresponding layers.

Unit testing is like testing individual chemical compounds in isolation — we verify that each component (Topic Manager, Consumer Groups, Acknowledgment Tracker) behaves correctly when given controlled inputs, with all dependencies replaced by predictable test doubles. **Integration testing** is like testing drug interactions — we verify that when multiple components work together, they produce the expected combined effects without harmful side effects. **Milestone verification** is like clinical trial checkpoints — we verify that after each development phase, the system delivers the promised therapeutic benefits to real users. **Chaos testing** is like stress-testing drugs under extreme conditions — we deliberately inject failures to ensure our system maintains safety and efficacy even when things go wrong.

The key insight is that each testing layer catches different types of problems. Unit tests catch logic errors in individual algorithms. Integration tests catch communication failures between components. Milestone verification catches requirements misunderstandings. Chaos testing catches resilience failures that only emerge under stress. Just as pharmaceutical companies can't skip any testing phase without risking patient safety, we can't skip any testing layer without risking production failures.

Unit Testing Approach

Unit testing in our message queue system focuses on testing individual components in complete isolation from their dependencies. The fundamental principle is **dependency injection with mock objects** — every component receives its dependencies through constructor parameters rather than creating them internally, allowing us to substitute predictable test doubles for complex real dependencies.

Core Testing Infrastructure

Our unit testing approach requires a robust mocking infrastructure that can simulate the behavior of each component's dependencies. The key is creating **behavioral contracts** that define exactly how components interact, then implementing both real and mock versions that satisfy the same contract.

Component Under Test	Primary Dependencies	Mock Requirements	Test Focus Areas
`TopicManager`	WAL Logger, Metrics Collector, Connection Manager	Controlled WAL responses, predictable metrics recording, simulated connection failures	Topic creation, wildcard matching, subscriber fanout, cleanup policies
`GroupCoordinator`	Topic Manager, Heartbeat Tracker, WAL Logger	Deterministic consumer assignments, controllable heartbeat timeouts, WAL persistence verification	Round-robin assignment, sticky assignment, rebalancing triggers, member lifecycle
`AckTracker`	WAL Logger, Group Coordinator, Poison Detector	WAL write verification, consumer load simulation, poison detection triggers	Message tracking, timeout detection, redelivery logic, load balancing
`FlowController`	Lag Monitor, Throttle Manager, Metrics Collector	Controlled lag measurements, predictable throttling responses, metrics verification	Capacity assessment, backpressure application, throttling strategies
`DLQManager`	Storage Interface, Topic Manager, Ack Tracker	Database operation simulation, topic creation responses, ack tracking integration	Message archival, replay planning, statistics collection

Mock Design Patterns

The most critical aspect of unit testing message queue components is creating mocks that can simulate complex temporal behaviors. Real message processing involves timing-dependent interactions — heartbeat timeouts, acknowledgment deadlines, rebalancing coordination. Our mocks must control the flow of time to make these interactions deterministic.

Time-Controlled Mocks use a `MockTimeProvider` interface that allows tests to advance simulated time on demand. Instead of using `time.Now()` directly, components accept a `TimeProvider` interface. Tests inject a `MockTimeProvider` that starts at a fixed time and advances only when the test calls `AdvanceTime(duration)`.

```go
type TimeProvider interface {
    Now() time.Time
    After(duration time.Duration) <-chan time.Time
}
```

State-Recording Mocks capture every interaction for later verification. For example, `MockWALLogger` records every `AppendRecord` call with its parameters, allowing tests to verify that persistence occurred with the correct data and in the correct order.

Failure-Injection Mocks can be programmed to fail in specific ways at specific times. A `MockGroupCoordinator` might fail the third `JoinGroup` call with a specific error, allowing us to test how `AckTracker` handles coordination failures during rebalancing.

Component-Specific Testing Strategies

Topic Manager Unit Tests focus on the core responsibilities of topic lifecycle management and message routing. The critical test categories include:

1. **Topic Creation and Retention**: Verify that topics are created with correct retention policies and that cleanup occurs according to time and size limits

2. **Wildcard Subscription Matching**: Exhaustively test the `matchPattern` logic with hierarchical topic names and complex wildcard patterns
3. **Subscriber Fanout**: Verify that messages are delivered to all matching subscribers and that failed deliveries don't prevent other deliveries
4. **Concurrent Publisher Handling**: Test that multiple publishers can write to the same topic simultaneously without corruption
5. **Memory Leak Prevention**: Verify that disconnected subscribers are cleaned up and that abandoned topics are eventually garbage collected

Consumer Group Unit Tests concentrate on the assignment algorithms and rebalancing coordination. Key test scenarios include:

1. **Assignment Strategy Correctness**: Verify round-robin and sticky assignment produce the expected message distribution across consumers
2. **Rebalancing State Management**: Test that rebalancing locks prevent concurrent modifications and that generation numbers prevent split-brain scenarios
3. **Member Lifecycle Handling**: Verify proper state transitions when consumers join, leave voluntarily, or timeout
4. **Load Balancing Verification**: Test that message assignments consider consumer capacity and current load
5. **Coordination Protocol**: Verify the multi-step rebalancing protocol handles failures at each step

Acknowledgment Tracker Unit Tests focus on message lifecycle management and redelivery logic. Critical areas include:

1. **Timeout Detection Accuracy**: Verify that `CheckTimeouts` identifies exactly the messages that have exceeded their deadlines
2. **Redelivery Assignment**: Test that timed-out messages are reassigned to available consumers with appropriate backoff
3. **Poison Message Detection**: Verify that messages exceeding retry limits are correctly identified and routed to dead letter processing
4. **Consumer Load Balancing**: Test that message assignments consider current consumer load and processing capacity
5. **State Consistency**: Verify that acknowledgment processing maintains correct state even under concurrent operations

Test Data Management

Unit tests require carefully constructed test data that exercises edge cases without becoming unmaintainable. The key is using **test data builders** that create valid objects with sensible defaults while allowing targeted modifications for specific test scenarios.

```go
type MessageBuilder struct {
    message *Message
}

func NewTestMessage() *MessageBuilder {
    return &MessageBuilder{
        message: &Message{
            ID: "test-" + generateUniqueID(),
            Topic: "test.topic",
            Payload: []byte("test payload"),
            Timestamp: testTimeProvider.Now().Unix(),
            RetryCount: 0,
            MaxRetries: 3,
        },
    }
}

func (b *MessageBuilder) WithTopic(topic string) *MessageBuilder {
    b.message.Topic = topic
    return b
}

func (b *MessageBuilder) WithRetryCount(count int) *MessageBuilder {
    b.message.RetryCount = count
    return b
}
```

This pattern allows tests to create realistic test data concisely while making the test's intent clear through the builder method calls.

Integration Testing Scenarios

Integration testing verifies that components work correctly together, focusing on the **interfaces between components** and the **end-to-end data flows** that span multiple components. Unlike unit tests that isolate components, integration tests use real implementations of all components but control external dependencies like disk storage and network connections.

Integration Test Infrastructure

Integration tests require a **controlled environment** that provides real component interactions while maintaining deterministic behavior. This involves creating a test harness that manages component lifecycle and provides controlled external dependencies.

External Dependency	Test Implementation	Control Mechanisms	Failure Injection
File System	In-memory file system or temp directories	Controlled disk space limits, configurable fsync behavior	Disk full errors, permission denied, corruption
Network Connections	Local TCP connections or in-process channels	Controllable latency, bandwidth limits	Connection drops, partial reads, timeout
System Clock	Mock time provider with manual advancement	Deterministic timeout behavior, controlled time progression	Clock skew, time jumps
Operating System	Containerized environment or process isolation	Resource limits, signal handling	Process crashes, resource exhaustion

The integration test harness manages **component startup and shutdown** in the correct order, ensuring that dependencies are available before dependent components initialize. It also provides **test lifecycle management** — setting up clean state before each test and ensuring complete cleanup afterward.

End-to-End Message Flows

The most critical integration tests verify complete message flows from producer to consumer, including all persistence, acknowledgment, and error handling steps. These tests exercise the coordination between all major components.

Basic Publish-Subscribe Flow tests the fundamental message routing:

1. Producer connects and publishes message to topic
2. `TopicManager` receives message and triggers `WALLogger.AppendRecord`
3. After successful persistence, `TopicManager` identifies matching subscribers
4. Message is delivered to all subscribers via their connections
5. Subscribers send acknowledgments back through protocol handler
6. `AckTracker` processes acknowledgments and marks message as complete

The test verifies that each step occurs in the correct order and that failure at any step prevents subsequent steps from executing.

Consumer Group Distribution Flow tests the coordination between group management and message delivery:

1. Multiple consumers join the same consumer group
2. `GroupCoordinator` assigns consumers to partitions using round-robin strategy
3. Producer publishes multiple messages to the topic
4. Messages are distributed among group members according to assignment
5. Each consumer acknowledges their assigned messages
6. Verify that no message is delivered to multiple group members

Rebalancing Coordination Flow tests the complex multi-step rebalancing process:

1. Consumer group has stable membership with assigned messages
2. New consumer joins the group
3. `GroupCoordinator` initiates rebalancing by incrementing generation number
4. Existing consumers stop receiving new assignments
5. `AckTracker` waits for pending acknowledgments to complete or timeout
6. New assignment strategy redistributes work among all members
7. Message delivery resumes with new assignments

This test verifies that rebalancing doesn't cause message loss or duplication and that the system returns to stable state after membership changes.

Component Interaction Patterns

Integration tests focus heavily on **component coordination patterns** — the specific sequences of method calls and state changes that occur when components collaborate to handle complex operations.

Persistence-Before-Delivery Pattern ensures that messages are durably stored before any delivery confirmation:

1. `TopicManager.PublishMessage` calls `WAL.AppendRecord` with message data
2. `WAL.AppendRecord` writes to disk and calls `fsync` for durability
3. Only after successful persistence does `TopicManager` proceed with fanout delivery
4. If persistence fails, the publish operation returns error and no delivery occurs

Coordination-With-Timeout Pattern handles operations that require coordination between multiple components with failure detection:

1. Component A initiates operation and starts timeout timer
2. Component A sends coordination messages to Components B and C
3. Components B and C respond with confirmation or rejection
4. If all confirmations received before timeout, operation proceeds
5. If timeout expires or any rejection received, operation aborts with cleanup

State-Handoff Pattern manages ownership transfer of resources between components:

1. Component A identifies resource that needs ownership transfer
2. Component A notifies Component B of pending transfer with transaction ID
3. Component B acknowledges readiness and prepares to accept ownership
4. Component A transfers resource state and confirms transfer completion
5. Component B acknowledges successful transfer and assumes full ownership
6. Component A removes resource from its state

These patterns require precise timing and error handling that can only be verified through integration testing with real component interactions.

Data Consistency Verification

Integration tests must verify that the system maintains **data consistency** across all components even when operations fail partially. This requires testing scenarios where operations succeed in some components but fail in others.

Crash Recovery Consistency tests verify that the system can restore consistent state after unexpected shutdown:

1. Start system with multiple active consumer groups and pending messages
2. Publish several messages and allow partial processing
3. Simulate broker crash during message processing
4. Restart broker and verify `RecoveryFlow` restores correct state
5. Verify that no messages are lost or duplicated
6. Verify that consumer group assignments are correctly restored

Rebalancing Consistency tests verify that consumer group state remains consistent even when rebalancing is interrupted:

1. Initiate consumer group rebalancing during active message processing
2. Simulate failures at different points in the rebalancing sequence
3. Verify that the system either completes rebalancing or rolls back to previous stable state
4. Verify that no messages are assigned to multiple consumers
5. Verify that no messages are orphaned without any assigned consumer

Milestone Verification Checkpoints

Each project milestone represents a significant increase in system capability and complexity. Milestone verification ensures that the implementation actually delivers the promised functionality and that the foundation is solid before proceeding to the next milestone.

Milestone 1: Core Queue & Wire Protocol

Acceptance Criteria Verification for Milestone 1 focuses on the fundamental publish-subscribe functionality and binary protocol implementation.

Verification Category	Test Scenario	Expected Behavior	Failure Indicators
TCP Connection Handling	Multiple clients connect simultaneously	Server accepts all connections and maintains separate state	Connection refused, shared state corruption
Binary Protocol Parsing	Send malformed frames with invalid headers	Server rejects invalid frames with proper error codes	Server crash, silent data corruption
Topic Message Storage	Publish 1000 messages to same topic rapidly	All messages stored in order without loss	Missing messages, incorrect ordering
Subscriber Fanout	Multiple subscribers to same topic	Each subscriber receives all published messages	Missing deliveries, duplicate deliveries
Wildcard Subscriptions	Subscribe to `sensors.*.temperature` pattern	Receives messages from `sensors.room1.temperature` and `sensors.room2.temperature`	Pattern matching failures, incorrect routing

Protocol Compliance Testing verifies that the binary protocol implementation handles all edge cases correctly:

1. **Frame Boundary Detection**: Send messages split across multiple TCP packets to verify frame reconstruction
2. **Maximum Frame Size Enforcement**: Send frames exceeding `MaxFrameSize` limit to verify rejection
3. **Protocol Magic Verification**: Send frames with incorrect magic bytes to verify protocol version checking
4. **Partial Read Handling**: Use slow network connections to verify handling of partial TCP reads
5. **Connection Cleanup**: Abruptly disconnect clients to verify proper cleanup of subscription state

Performance Baseline Establishment measures system performance under normal conditions to detect regressions in future milestones:

- Message throughput: Minimum 10,000 messages/second with single publisher and subscriber
- Subscription latency: Maximum 1ms from publish to subscriber notification
- Memory usage: Linear growth with number of stored messages, no memory leaks
- Connection capacity: Support at least 1,000 concurrent client connections

Milestone 2: Consumer Groups & Acknowledgment

Consumer Group Functionality Verification tests the round-robin distribution and acknowledgment mechanisms:

Test Scenario	Setup	Expected Result	Verification Method
Round-Robin Distribution	3 consumers in group, publish 9 messages	Each consumer receives exactly 3 messages	Track message assignments per consumer
Acknowledgment Tracking	Consumer receives message but doesn't ACK	Message redelivered after timeout	Monitor redelivery events
NACK Handling	Consumer sends NACK for received message	Message immediately reassigned to different consumer	Verify reassignment timing
Consumer Group Rebalancing	Add 4th consumer to group with 3 active members	Work redistributed among 4 consumers	Monitor assignment changes
Failed Consumer Detection	Consumer stops sending heartbeats	Consumer removed from group, work reassigned	Track heartbeat timeouts

Rebalancing Coordination Testing verifies the complex multi-step rebalancing process:

1. **Rebalance Trigger Detection**: Join/leave events trigger rebalancing within configurable timeout
2. **Generation Number Coordination**: All consumers acknowledge new generation before proceeding
3. **Assignment Redistribution**: New assignments minimize message reassignment for sticky strategy
4. **Inflight Message Handling**: Pending messages are properly reassigned during rebalancing
5. **Split-Brain Prevention**: Multiple simultaneous rebalances are serialized correctly

Acknowledgment Reliability Testing focuses on the reliability guarantees:

- **Exactly-Once Processing**: No message is processed by multiple consumers in the same group
- **At-Least-Once Delivery**: No message is lost due to consumer failures or timeouts
- **Poison Message Handling**: Messages exceeding retry limit are moved to dead letter processing
- **Timeout Accuracy**: Message timeouts occur within ±100ms of configured deadline

Milestone 3: Persistence & Backpressure

Persistence Durability Verification tests that the append-only log provides actual crash recovery:

Recovery Scenario	Test Setup	Expected Recovery	Verification Steps
Clean Shutdown Recovery	Normal shutdown with pending messages	All pending messages restored	Compare pre/post shutdown state
Crash During Write	Kill broker during WAL write operation	Partial writes detected and handled	Check log integrity after restart
Corrupted Log Recovery	Introduce corruption in log file	System detects corruption, recovers valid portion	Monitor corruption detection logs
Large Log Recovery	1M+ messages in log file	Recovery completes in reasonable time	Measure recovery performance

Backpressure Mechanism Testing verifies flow control under various load conditions:

1. **Consumer Lag Detection**: Slow consumer processing triggers lag monitoring alerts
2. **Producer Throttling**: Throttling reduces producer throughput to match consumer capacity
3. **Adaptive Throttling**: Throttling intensity adjusts based on lag trend direction
4. **Throttling Release**: Throttling automatically releases when consumer lag decreases
5. **Cascading Prevention**: Backpressure doesn't cause upstream service failures

Retention Policy Enforcement tests that old messages are cleaned up correctly:

- **Time-Based Retention**: Messages older than retention period are removed
- **Size-Based Retention**: Log size doesn't exceed configured maximum
- **Message-Count Retention**: Topic doesn't exceed maximum message count
- **Retention During Recovery**: Retention policies are enforced during crash recovery

Milestone 4: Dead Letter Queue & Monitoring

Dead Letter Queue Processing Verification tests poison message handling:

DLQ Scenario	Message Condition	Expected DLQ Behavior	Verification Method
Retry Exhaustion	Message fails processing 5 times	Moved to DLQ with failure history	Check DLQ contents
DLQ Message Inspection	Administrative query for failed messages	Returns message content and failure reason	API response validation
Message Replay	Replay DLQ message to original topic	Message reprocessed successfully	Monitor processing outcome
DLQ Retention	Old DLQ messages exceed retention	Messages purged automatically	Check DLQ size limits

Monitoring API Functionality tests the administrative and observability interfaces:

1. **Queue Depth Metrics**: API returns accurate count of pending messages per topic
2. **Consumer Lag Monitoring**: API reports processing delay for each consumer group
3. **Throughput Measurement**: API provides message/second rates for publishers and consumers
4. **Error Rate Tracking**: API reports failure rates and error categories
5. **Health Check Accuracy**: API health endpoint reflects actual system component status

End-to-End System Verification tests complete functionality across all milestones:

- **Producer → Consumer Flow**: Message published, persisted, delivered, acknowledged, and removed

- **Failed Message Recovery**: Failed message retried, eventually moved to DLQ, then successfully replayed
- **System Restart Resilience**: Complete system restart with full state recovery and continued operation
- **Performance Under Load**: System maintains functionality under high message volume and many concurrent clients

Chaos Testing Scenarios

Chaos testing deliberately injects failures into a running system to verify that error handling, recovery mechanisms, and resilience features actually work under realistic failure conditions. The goal is to uncover failure modes that only emerge when multiple components fail simultaneously or when failures occur at particularly unfortunate timing.

Failure Classification and Injection

Chaos testing requires a systematic approach to failure injection that covers all major failure categories and timing scenarios. The key insight is that real-world failures are often **combinations** of simple failures that occur at the worst possible times.

Failure Category	Specific Failure Types	Injection Methods	Expected System Response
Network Failures	Connection drops, partitions, high latency	Traffic shaping, firewall rules, proxy delays	Reconnection, timeout handling, degraded mode
Process Failures	Broker crash, consumer crash, OOM kill	Process signals, resource limits, container kills	Crash recovery, consumer rebalancing, state restoration
Storage Failures	Disk full, corruption, permission denied	Filesystem manipulation, FUSE overlays	Error handling, graceful degradation, alerting
Resource Exhaustion	Memory exhaustion, CPU saturation, fd limits	Resource cgroups, load injection	Backpressure activation, throttling, load shedding
Timing Failures	Clock skew, delayed operations, race conditions	Clock manipulation, artificial delays	Timeout handling, consistency maintenance

Network Partition Scenarios test the system's behavior when components cannot communicate:

1. **Producer-Broker Partition**: Producer cannot reach broker but consumers can
2. **Consumer-Broker Partition**: Consumers isolated while producers continue sending
3. **Split-Brain Partition**: Broker cluster partitioned with active nodes on both sides
4. **Intermittent Partition**: Network flaps between connected and partitioned states

Cascading Failure Scenarios test whether isolated failures propagate through the system:

1. **Consumer Death Spiral**: One slow consumer causes backpressure affecting other consumers
2. **Memory Exhaustion Cascade**: Large message backlog causes OOM, triggering more failures
3. **Rebalancing Storm**: Consumer failures trigger rapid rebalancing cycles
4. **Storage Cascade**: Disk full condition causes crashes, leading to more disk usage during recovery

Temporal Failure Injection

The most revealing chaos tests inject failures at **specific timing windows** when the system is performing complex coordinated operations. These tests uncover race conditions and coordination bugs that are nearly impossible to find through normal testing.

Rebalancing Disruption Tests inject failures during the multi-step consumer group rebalancing process:

1. **Coordinator Failure During Rebalancing**: Kill group coordinator after rebalance initiation but before completion
2. **Consumer Crash During Assignment**: Crash consumer immediately after receiving new assignment but before acknowledgment
3. **Network Partition During Coordination**: Partition network during consumer group coordination phase
4. **Clock Skew During Timeout**: Introduce clock skew during rebalancing timeout detection

Message Processing Disruption Tests target the critical acknowledgment and redelivery windows:

1. **Broker Crash After Delivery Before ACK**: Crash broker immediately after delivering message but before receiving acknowledgment
2. **Consumer Crash During Processing**: Kill consumer while message is being processed but before ACK sent

3. **Network Failure During ACK**: Drop network connection exactly when acknowledgment is being transmitted
4. **Storage Failure During Persistence**: Introduce disk error during WAL write for acknowledgment record

Recovery Verification Under Stress

Chaos testing must verify not just that the system survives failures, but that **recovery mechanisms work correctly** even when the system is under stress or experiencing multiple concurrent failures.

Recovery Performance Degradation tests measure how recovery time and success rate change under various stress conditions:

Stress Condition	Recovery Operation	Baseline Performance	Degraded Performance Threshold
High Message Volume	Crash recovery from WAL	30 seconds for 1M messages	Must complete within 5 minutes
Many Consumer Groups	Consumer group rebalancing	2 seconds for 100 consumers	Must complete within 30 seconds
Large Topic Count	Topic metadata recovery	5 seconds for 10K topics	Must complete within 60 seconds
Memory Pressure	DLQ message replay	100 messages/second	Must maintain >10 messages/second

Concurrent Failure Handling tests verify that the system can handle multiple simultaneous failures:

1. **Broker + Consumer Failures**: Crash broker and half of consumers simultaneously
2. **Storage + Network Failures**: Introduce disk errors and network partitions concurrently
3. **Resource + Process Failures**: Exhaust memory while crashing consumer processes
4. **Timing + Coordination Failures**: Introduce clock skew during consumer failures

State Consistency Under Chaos verifies that the system maintains data consistency even when experiencing multiple failures:

- **Message Delivery Guarantees**: No message loss or duplication despite broker crashes during delivery
- **Consumer Group Consistency**: Group membership and assignment state remains coherent despite coordination failures
- **Persistence Consistency**: WAL state remains valid and recoverable despite storage failures
- **Acknowledgment Consistency**: Message acknowledgment state is consistent across crashes and recoveries

Chaos Testing Infrastructure

Implementing effective chaos testing requires infrastructure that can inject failures precisely and monitor system behavior comprehensively. The chaos testing framework must be able to coordinate complex failure scenarios across multiple system components.

Failure Orchestration System manages the timing and coordination of multiple failure injections:

```go
type ChaosScenario struct {
    Name string
    Duration time.Duration
    FailureSequence []FailureEvent
    ExpectedOutcomes []OutcomeAssertion
    RecoveryValidation []ValidationStep
}

type FailureEvent struct {
    Timestamp time.Duration  // relative to scenario start
    TargetComponent string
    FailureType FailureType
    Parameters map[string]interface{}
    Duration time.Duration
}
```

System State Monitoring captures detailed system state before, during, and after failure injection to verify that recovery was successful:

- **Message Flow Tracking**: Record all messages published, delivered, and acknowledged
- **Consumer Group State**: Monitor group membership, assignments, and rebalancing events
- **Resource Utilization**: Track memory, CPU, disk, and network usage throughout scenarios
- **Error Rate Monitoring**: Measure error rates and recovery times for all components

Failure Impact Assessment analyzes the results of chaos testing to identify areas needing resilience improvements:

1. **Failure Blast Radius**: How many components were affected by each injected failure
2. **Recovery Time Distribution**: Statistical analysis of recovery times across multiple test runs
3. **Data Loss Assessment**: Any instances of message loss or corruption during failures
4. **Availability Impact**: Percentage of time system was unavailable during failure scenarios

Implementation Guidance

Technology Recommendations

Testing Layer	Simple Option	Advanced Option	Recommendation
Unit Test Framework	Go's built-in `testing` package	Testify with assertions and mocks	Start with built-in, add Testify for complex mocks
Integration Test Environment	Docker Compose with real containers	Kubernetes test clusters	Docker Compose for development, K8s for CI
Chaos Testing	Manual failure injection scripts	Chaos Monkey, Litmus, Gremlin	Start with scripts, evolve to Chaos Monkey
Test Data Management	JSON fixtures with Go embed	Property-based testing with Rapid	JSON fixtures for deterministic tests
Mock Framework	Hand-written interfaces and structs	GoMock with code generation	Hand-written for learning, GoMock for scale

Recommended File Structure

```
message-queue/
├── cmd/
│   ├── broker/main.go                    ← broker entry point
│   └── chaos-tester/main.go              ← chaos testing tool
├── internal/
│   ├── components/
│   │   ├── topic/
│   │   │   ├── manager.go                ← topic manager implementation
│   │   │   ├── manager_test.go           ← unit tests
│   │   │   └── manager_integration_test.go ← integration tests
│   │   ├── groups/
│   │   │   ├── coordinator.go
│   │   │   ├── coordinator_test.go
│   │   │   └── coordinator_integration_test.go
│   │   └── ack/
│   │       ├── tracker.go
│   │       ├── tracker_test.go
│   │       └── tracker_integration_test.go
│   ├── testing/
│   │   ├── mocks/                        ← reusable mock implementations
│   │   │   ├── wal_logger.go
│   │   │   ├── time_provider.go
│   │   │   └── connection_manager.go
│   │   ├── builders/                     ← test data builders
│   │   │   ├── message_builder.go
│   │   │   ├── consumer_builder.go
│   │   │   └── topic_builder.go
│   │   ├── harness/                      ← integration test infrastructure
│   │   │   ├── broker_harness.go
│   │   │   ├── client_harness.go
│   │   │   └── cluster_harness.go
│   │   └── chaos/                        ← chaos testing framework
│   │       ├── failure_injector.go
│   │       ├── scenario_runner.go
│   │       └── state_validator.go
│   └── testdata/
│       ├── configs/                      ← test configuration files
│       ├── scenarios/                    ← chaos test scenarios
│       └── fixtures/                     ← test message data
├── test/
│   ├── integration/                      ← full system integration tests
│   │   ├── publish_subscribe_test.go
│   │   ├── consumer_groups_test.go
│   │   ├── persistence_test.go
│   │   └── dlq_test.go
│   ├── chaos/                           ← chaos testing scenarios
│   │   ├── network_partition_test.go
│   │   ├── process_failure_test.go
│   │   └── recovery_stress_test.go
│   └── performance/                      ← performance benchmarks
│       ├── throughput_test.go
│       └── latency_test.go
└── scripts/
    ├── run_unit_tests.sh                ← test execution scripts
    ├── run_integration_tests.sh
    ├── run_chaos_tests.sh
    └── performance_benchmark.sh
```

Core Testing Infrastructure Code

Mock Time Provider for Deterministic Testing

```go
// internal/testing/mocks/time_provider.go

package mocks

import (
    "sync"
    "time"
)

// MockTimeProvider allows tests to control the flow of time
type MockTimeProvider struct {
    currentTime time.Time
    timers      map[string]*MockTimer
    mutex       sync.RWMutex
}

// NewMockTimeProvider creates a time provider starting at the given time
func NewMockTimeProvider(startTime time.Time) *MockTimeProvider {
    return &MockTimeProvider{
        currentTime: startTime,
        timers:      make(map[string]*MockTimer),
    }
}

// Now returns the current simulated time
func (m *MockTimeProvider) Now() time.Time {
    m.mutex.RLock()
    defer m.mutex.RUnlock()
    return m.currentTime
}

// After creates a timer that fires after the given duration
func (m *MockTimeProvider) After(duration time.Duration) <-chan time.Time {
    m.mutex.Lock()
    defer m.mutex.Unlock()

    timer := &MockTimer{
        fireTime: m.currentTime.Add(duration),
        ch:       make(chan time.Time, 1),
```

```
    }

    timerID := generateTimerID()
    m.timers[timerID] = timer
    return timer.ch
}

// AdvanceTime moves the simulated time forward and fires any ready timers
func (m *MockTimeProvider) AdvanceTime(duration time.Duration) {
    m.mutex.Lock()
    defer m.mutex.Unlock()

    m.currentTime = m.currentTime.Add(duration)

    // Fire any timers that should trigger
    for id, timer := range m.timers {
        if !timer.fired && !timer.fireTime.After(m.currentTime) {
            timer.ch <- timer.fireTime
            timer.fired = true
            delete(m.timers, id)
        }
    }
}

type MockTimer struct {
    fireTime time.Time
    ch       chan time.Time
    fired    bool
}

func generateTimerID() string {
    // Implementation omitted for brevity
    return "timer-" + randomString(8)
}
```

WAL Logger Mock with Recording Capabilities

```go
// internal/testing/mocks/wal_logger.go

package mocks

import (
    "encoding/json"
    "errors"
    "sync"
)

// MockWALLogger records all operations for verification
type MockWALLogger struct {
    records    []WALRecord
    nextOffset int64
    failures   map[int64]error  // offset -> error to return
    mutex      sync.RWMutex
}

// NewMockWALLogger creates a new mock WAL logger
func NewMockWALLogger() *MockWALLogger {
    return &MockWALLogger{
        records:  make([]WALRecord, 0),
        failures: make(map[int64]error),
    }
}

// AppendRecord records the operation and returns the next offset
func (m *MockWALLogger) AppendRecord(recordType WALRecordType, data json.RawMessage) (int64, error) {
    m.mutex.Lock()
    defer m.mutex.Unlock()

    // Check if we should inject a failure
    if err, exists := m.failures[m.nextOffset]; exists {
        return 0, err
    }

    record := WALRecord{
        RecordType: recordType,
        Data:      data,
```

```
        Timestamp: time.Now().Unix(),
    }

    m.records = append(m.records, record)
    offset := m.nextOffset
    m.nextOffset++

    return offset, nil
}

// ReadRecord retrieves a record by offset
func (m *MockWALLogger) ReadRecord(offset int64) (*WALRecord, error) {
    m.mutex.RLock()
    defer m.mutex.RUnlock()

    if offset < 0 || offset >= int64(len(m.records)) {
        return nil, errors.New("offset out of range")
    }

    return &m.records[offset], nil
}

// SetFailureAt configures the mock to fail at a specific offset
func (m *MockWALLogger) SetFailureAt(offset int64, err error) {
    m.mutex.Lock()
    defer m.mutex.Unlock()
    m.failures[offset] = err
}

// GetRecordCount returns the number of records written
func (m *MockWALLogger) GetRecordCount() int {
    m.mutex.RLock()
    defer m.mutex.RUnlock()
    return len(m.records)
}

// GetRecordsByType returns all records of a specific type
```

```go
func (m *MockWALLogger) GetRecordsByType(recordType WALRecordType) []WALRecord {
    m.mutex.RLock()
    defer m.mutex.RUnlock()

    var results []WALRecord
    for _, record := range m.records {
        if record.RecordType == recordType {
            results = append(results, record)
        }
    }
    return results
}
```

Unit Test Examples

Topic Manager Wildcard Subscription Testing

```go
// internal/components/topic/manager_test.go

func TestTopicManager_WildcardMatching(t *testing.T) {

    tests := []struct {

        name           string

        pattern        string

        topic          string

        shouldMatch    bool

        description    string

    }{

        {

            name:        "single_wildcard_match",

            pattern:     "sensors.*.temperature",

            topic:       "sensors.room1.temperature",

            shouldMatch: true,

            description: "Single wildcard should match one segment",

        },

        {

            name:        "single_wildcard_no_match_multiple_segments",

            pattern:     "sensors.*.temperature",

            topic:       "sensors.floor1.room1.temperature",

            shouldMatch: false,

            description: "Single wildcard should not match multiple segments",

        },

        {

            name:        "multi_wildcard_match",

            pattern:     "sensors.#",

            topic:       "sensors.floor1.room1.temperature",

            shouldMatch: true,

            description: "Multi wildcard should match any remaining segments",

        },

        {

            name:        "complex_pattern_match",

            pattern:     "sensors.*.#",

            topic:       "sensors.room1.temperature.celsius",

            shouldMatch: true,

            description: "Combined wildcards should work together",
```

```
        },
    }


    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            // TODO: Initialize TopicManager with mock dependencies
            // TODO: Create subscription with wildcard pattern
            // TODO: Test pattern matching against various topic names
            // TODO: Verify that only matching topics receive messages
            // TODO: Test that subscription cleanup works correctly
        })
    }
}
```

Integration Test Skeleton

Consumer Group Rebalancing Integration Test

```go
// internal/components/groups/coordinator_integration_test.go
func TestGroupCoordinator_RebalancingFlow(t *testing.T) {
    // TODO: Set up integration test harness with real components
    // TODO: Create consumer group with 3 initial members
    // TODO: Publish messages and verify round-robin distribution
    // TODO: Add 4th consumer and trigger rebalancing
    // TODO: Verify that work is redistributed correctly
    // TODO: Remove consumer and verify rebalancing again
    // TODO: Verify no message loss or duplication during rebalancing
    // TODO: Clean up all resources
}
```

Chaos Testing Framework

Failure Injection Infrastructure

```go
// internal/testing/chaos/failure_injector.go

package chaos


import (

    "context"

    "time"

)


// FailureInjector manages coordinated failure injection

type FailureInjector struct {

    targets map[string]FailureTarget

    active  map[string]*ActiveFailure

}


// FailureTarget defines a component that can have failures injected

type FailureTarget interface {

    InjectFailure(failureType string, parameters map[string]interface{}) error

    RecoverFromFailure(failureType string) error

    GetHealthStatus() HealthStatus

}


// InjectScenario executes a complete chaos testing scenario

func (f *FailureInjector) InjectScenario(ctx context.Context, scenario *ChaosScenario) (*ScenarioResult, error) {

    // TODO: Validate scenario definition and target availability

    // TODO: Record initial system state for comparison

    // TODO: Execute failure sequence according to timeline

    // TODO: Monitor system behavior during failures

    // TODO: Trigger recovery and measure recovery time

    // TODO: Validate final system state and data consistency

    // TODO: Generate comprehensive scenario report



    return nil, nil

}
```

Milestone Verification Scripts

Milestone 1 Verification Checkpoint

```bash
#!/bin/bash

# scripts/verify_milestone_1.sh

echo "Verifying Milestone 1: Core Queue & Wire Protocol"

# TODO: Start broker in background

# TODO: Test basic TCP connection establishment

# TODO: Send PUBLISH command and verify response

# TODO: Test SUBSCRIBE command with fanout delivery

# TODO: Verify wildcard subscription matching

# TODO: Test connection cleanup on disconnect

# TODO: Measure basic performance metrics

# TODO: Generate verification report

echo "Milestone 1 verification complete"
```

Expected Behavior After Milestone 1

- Broker accepts multiple TCP connections without interference
- Binary protocol correctly parses all command types
- Messages are delivered to all matching subscribers
- Wildcard patterns match hierarchical topic names correctly
- Disconnected clients are cleaned up automatically
- System handles at least 1,000 concurrent connections
- Message throughput exceeds 10,000 messages/second

Performance Benchmarking

```go
// test/performance/throughput_test.go

func BenchmarkMessageThroughput(b *testing.B) {

    // TODO: Set up broker with optimized configuration

    // TODO: Create multiple concurrent publishers

    // TODO: Measure messages/second under various loads

    // TODO: Track memory usage and connection counts

    // TODO: Verify that performance scales with resources

}
```

Debugging Guide

> **Milestone(s):** All milestones (1-4) — comprehensive debugging strategies that help diagnose and resolve issues throughout the entire development process, from basic TCP connectivity through complex persistence and recovery scenarios

Mental Model: The Digital Detective

Think of debugging a message queue like being a detective investigating a complex case. Every symptom is a clue, every log entry is evidence, and every metric is a witness statement. Just as a detective follows a methodical process — examining the crime scene, interviewing witnesses, analyzing evidence, and testing theories — debugging requires systematic investigation.

The message queue presents multiple "crime scenes": network connections, message routing, persistence files, and consumer state. Each component leaves traces of its activities through logs, metrics, and state changes. A skilled debugger learns to read these traces like a detective reads evidence, building a timeline of events that led to the problem.

The key insight is that message queue issues are rarely isolated — they're usually symptoms of systemic problems. A "lost message" might actually be a consumer crash that prevented acknowledgment. A "slow consumer" might be caused by disk I/O blocking the persistence layer. Like solving a mystery, the obvious explanation is often wrong, and the real cause requires deeper investigation.

Connection and Protocol Issues

Connection and protocol problems are the most common issues learners encounter when implementing the TCP-based wire protocol. These issues manifest immediately when clients attempt to connect and exchange messages with the broker.

TCP Connection Problems

The fundamental challenge with TCP connections is that they're stateful and can fail in numerous ways. Unlike HTTP's request-response model, message queue connections are long-lived and bidirectional, creating more opportunities for failure.

Socket Binding and Listening Issues

The first category of connection problems occurs during broker startup when establishing the listening socket. These issues prevent any clients from connecting and are typically caused by port conflicts or permission restrictions.

Symptom	Likely Cause	Diagnosis	Resolution
"Address already in use" during startup	Another process using the port	`netstat -tlnp | grep :8080` to find process	Kill conflicting process or change port
"Permission denied" binding to port	Attempting to bind to privileged port (< 1024)	Check if port < 1024 and running as non-root	Use port >= 1024 or run with elevated privileges
Broker starts but clients can't connect	Listening on wrong interface (localhost vs 0.0.0.0)	Check bind address in configuration	Bind to 0.0.0.0 for all interfaces or specific IP
Intermittent connection failures	OS socket limits reached	Check `ulimit -n` and active connections	Increase ulimit or implement connection pooling

Client Connection Establishment

Once the broker is listening, clients must successfully establish TCP connections and complete the protocol handshake. Connection establishment can fail due to network issues, broker overload, or client configuration problems.

Symptom	Likely Cause	Diagnosis	Resolution
"Connection refused" from client	Broker not listening on expected port	Verify broker is running: `telnet hostname port`	Check broker startup logs and port configuration
Connection hangs during handshake	Client or broker blocking on incomplete frame	Check if both ends are reading/writing correctly	Implement proper frame boundary detection
"Connection reset by peer"	Broker rejecting connection due to overload	Check broker connection count and limits	Implement connection limits with backoff retry
Random connection drops	Network issues or broker resource exhaustion	Monitor connection duration and system resources	Add connection heartbeat and resource monitoring

Protocol Frame Parsing Issues

The binary wire protocol introduces complexity around message framing and parsing. TCP is a stream protocol, so applications must handle partial reads and frame boundaries correctly.

Partial TCP Reads

TCP provides no guarantee that a single `read()` call will return a complete message frame. The operating system may deliver data in smaller chunks, requiring applications to buffer partial frames until complete.

⚠️ **Pitfall: Assuming Complete Frame Reception** Many learners write code like `conn.Read(buffer)` and assume the buffer contains a complete frame. This works in local testing but fails in production when network conditions cause TCP to segment data. The fix is to implement proper frame accumulation that continues reading until the expected frame size is received.

Frame Parsing State	Description	Next Action
Reading Magic Number	First 4 bytes of `FrameHeader`	Continue reading until 4 bytes accumulated
Reading Header	Bytes 5-12 of header	Continue reading until full 12-byte header
Reading Payload	Variable-length payload based on header	Read exactly `PayloadLength` bytes from header
Frame Complete	Full frame assembled	Process frame and reset to reading next magic number

The correct approach is to maintain a connection-level buffer that accumulates partial data across multiple TCP reads:

> **Critical Design Principle**: Always read in a loop until the expected data is fully received. TCP is a stream protocol that makes no guarantees about read boundaries.

Frame Boundary Detection

Without proper framing, the broker cannot determine where one message ends and the next begins. This is especially problematic when multiple clients are sending messages rapidly over the same connection.

Error Pattern	Symptom	Cause	Fix
Frame sync lost	"Invalid magic number" errors	Misaligned frame reading after partial read	Reset connection and resynchronize on magic number
Payload truncation	"Unexpected EOF" during message processing	Reading fewer bytes than `PayloadLength` specifies	Loop until full payload read or connection closes
Frame corruption	Checksum failures or malformed commands	Buffer overflow or thread safety issues	Add proper bounds checking and synchronization
Memory exhaustion	Broker crashes with large frame claims	Client sending invalid `PayloadLength` values	Enforce `MaxFrameSize` limit and reject oversized frames

Protocol Version Compatibility

Version mismatches between client and broker can cause subtle failures that are difficult to diagnose. The protocol version is embedded in the `FrameHeader` but may not be checked consistently.

Version Issue	Detection	Resolution
Client using newer protocol version	Unknown frame types in broker logs	Implement version negotiation or graceful degradation
Broker using newer protocol version	Client failing to parse responses	Return version compatibility error with supported versions
Mixed protocol versions	Intermittent parsing failures	Log protocol version for each connection and audit compatibility

Connection Lifecycle Management

Long-lived connections require careful lifecycle management, including graceful shutdown, connection cleanup, and resource reclamation.

Connection Cleanup on Disconnect

When clients disconnect unexpectedly, the broker must clean up all associated state to prevent resource leaks. This includes removing subscriptions, releasing pending messages, and updating consumer group membership.

⚠️ **Pitfall: Incomplete Cleanup Leading to Ghost Consumers** A common issue is forgetting to clean up consumer group membership when a connection drops. This leaves "ghost consumers" that are assigned messages but never acknowledge them, causing head-of-line blocking. The fix is to implement comprehensive cleanup in the connection close handler.

Cleanup Task	Component Responsible	Failure Impact
Remove topic subscriptions	`TopicManager`	Messages delivered to closed connection
Update consumer group membership	`GroupCoordinator`	Ghost consumers preventing message delivery
Return pending messages	`AckTracker`	Messages stuck in limbo awaiting acknowledgment
Close heartbeat tracking	`HeartbeatTracker`	Consumer reported as healthy when disconnected
Release connection resources	`Handler`	Memory and file descriptor leaks

Heartbeat and Keepalive

Detecting failed connections quickly is essential for maintaining system responsiveness. TCP keepalive operates at the transport layer but may not detect application-level failures promptly enough for message queue requirements.

The broker implements application-level heartbeats through the `HeartbeatTracker` component. Clients must send periodic heartbeat frames, and the broker marks consumers as unhealthy if heartbeats are missed.

Heartbeat Parameter	Typical Value	Tuning Considerations
Heartbeat Interval	30 seconds	Balance between responsiveness and network overhead
Missed Heartbeat Threshold	3 consecutive misses	Allow for temporary network issues
Cleanup Delay	90 seconds total	Prevent premature cleanup during network partitions
Heartbeat Frame Size	Minimal (header only)	Reduce bandwidth consumption

Message Delivery Issues

Message delivery problems are often the most challenging to debug because they involve multiple system components and can have subtle timing dependencies. These issues typically manifest as lost messages, duplicate deliveries, or incorrect message ordering.

Lost Messages

Message loss can occur at several points in the delivery pipeline, from initial publication through final acknowledgment. The challenge is determining exactly where messages disappear and why.

Messages Never Reach Topic Queue

The first potential loss point is between the client sending a `PUBLISH` command and the message being enqueued in the topic's message queue. This typically indicates problems with the protocol handler or topic manager.

Loss Scenario	Detection Method	Root Cause Analysis
Protocol parsing failure	Check broker logs for parse errors	Malformed frame or unsupported command type
Topic creation failure	Verify topic exists in topic manager state	Insufficient permissions or disk space for topic metadata
Message validation failure	Look for validation errors in logs	Invalid message format or size limits exceeded
Memory allocation failure	System OOM logs or broker crash	Message too large or system memory exhausted

Messages Lost During Fanout

After successful enqueueing, messages must be delivered to all topic subscribers. Loss during fanout typically indicates issues with the subscription index or delivery mechanism.

⚠️ **Pitfall: Race Conditions During Subscription Changes** A subtle issue occurs when subscribers are added or removed while messages are being delivered. If the fanout algorithm snapshots the subscriber list at the wrong time, messages may be delivered to the old subscriber set. The fix is to use consistent subscriber snapshots or implement copy-on-write subscription lists.

Fanout Issue	Symptom	Diagnostic Steps
Subscription index corruption	Some subscribers never receive messages	Dump subscription index and verify all expected subscribers present
Wildcard matching failure	Messages not delivered to pattern subscribers	Test wildcard patterns against topic names manually
Connection failure during delivery	Messages acknowledged but not received	Check connection state for each subscriber during delivery
Delivery timeout	Slow subscribers blocking fanout	Monitor delivery latency per subscriber

Messages Lost in Consumer Groups

Consumer group delivery adds another layer of complexity where messages must be assigned to specific group members. Loss here typically indicates problems with the group coordinator or assignment strategy.

Consumer Group Loss	Diagnosis	Resolution
Messages assigned to disconnected consumer	Check consumer heartbeat status during assignment	Implement consumer health checking before assignment
Round-robin assignment skipping consumers	Verify assignment counter state	Audit assignment strategy implementation for off-by-one errors
Message assignment race condition	Multiple consumers receiving same message	Add atomic assignment with unique message IDs
Consumer group rebalancing drops messages	Messages in-flight during rebalancing disappear	Implement message reassignment during rebalance

Duplicate Messages

Duplicate delivery is often a consequence of reliability mechanisms like acknowledgment timeouts and redelivery. While "at-least-once" delivery semantics permit duplicates, excessive duplication can overwhelm consumers.

Acknowledgment Timeout Duplicates

The most common source of duplicates is acknowledgment timeouts that trigger message redelivery. This is expected behavior, but tuning timeout values incorrectly can cause excessive duplication.

Timeout Configuration	Too Short	Too Long	Optimal Setting
Acknowledgment Deadline	Frequent false positive timeouts	Delayed failure detection	2-3x average processing time
Redelivery Interval	Duplicate storms under load	Slow recovery from failures	Exponential backoff starting at 1 second
Consumer Health Check	Premature consumer marking as failed	Ghost consumers holding messages	Based on heartbeat interval

Rebalancing Duplicates

Consumer group rebalancing can cause messages to be delivered to multiple consumers if not handled carefully. This occurs when message assignments are not properly coordinated during membership changes.

⚠️ **Pitfall: Message Double-Assignment During Rebalancing** During rebalancing, a message might be in-flight to one consumer while being reassigned to another. If both consumers process the message successfully, it results in duplication. The fix is to implement a "generation number" system where consumers include their rebalance generation in acknowledgments, allowing the broker to ignore acknowledgments from the wrong generation.

Rebalancing Phase	Duplicate Risk	Mitigation Strategy
Member Addition	New member receives messages already assigned	Drain existing assignments before adding member
Member Removal	Messages reassigned while still being processed	Use acknowledgment generations to detect stale acks
Assignment Strategy Change	Different assignment logic creates overlaps	Implement assignment strategy versioning

Message Ordering Issues

Maintaining message order becomes challenging in a concurrent system with multiple publishers, fanout delivery, and consumer groups. Order violations can cause application-level correctness problems.

Publisher-Level Ordering

Multiple publishers sending to the same topic can interleave messages in unexpected ways. While this is generally acceptable, some applications require ordering within a publisher's message stream.

Ordering Problem	Cause	Solution Approach
Messages from same publisher out of order	Concurrent publishing or network reordering	Add sequence numbers to messages from publishers
Fast publisher overtaking slow publisher	Different network paths or processing times	Implement publisher-level queuing if ordering required
Timestamp-based ordering incorrect	Clock skew between publishers	Use broker-assigned timestamps for ordering

Consumer Group Ordering

Consumer groups distribute messages across multiple consumers for parallel processing, which inherently breaks total ordering. However, applications sometimes need ordering guarantees within subsets of messages.

⚠️ **Pitfall: Head-of-Line Blocking in Consumer Groups** If one consumer in a group becomes slow or fails, it can block processing of messages assigned to it while other consumers remain idle. This violates the goal of parallel processing. The solution is to implement consumer health monitoring and reassign messages from failed consumers to healthy ones.

Ordering Requirement	Implementation Approach	Trade-offs
No ordering required	Standard round-robin distribution	Maximum parallelism and throughput
Ordering within key groups	Hash-based assignment to consumers	Reduces parallelism but maintains key ordering
Global ordering	Single consumer processes all messages	Eliminates parallelism benefits
Approximate ordering	Time-window based ordering	Good balance of ordering and performance

Performance and Memory Issues

Performance problems in message queues typically manifest as high latency, low throughput, or excessive resource consumption. These issues often compound under load and may not appear during development testing.

Memory Leaks

Memory leaks in a message broker can be particularly dangerous because they accumulate over time and can cause sudden system failures when memory is exhausted.

Topic and Subscription Leaks

Topics and subscriptions that are created but never cleaned up represent one of the most common sources of memory leaks. This is especially problematic with dynamic topic creation and temporary subscriptions.

⚠️ **Pitfall: Accumulating Dead Topics** Applications often create topics dynamically based on user sessions or temporary workflows. If these topics are never explicitly deleted, they accumulate in the broker's memory along with their message queues and subscription lists. The fix is to implement topic garbage collection based on inactivity timeouts.

Memory Leak Source	Detection Method	Prevention Strategy
Abandoned topics with no subscribers	Monitor topic subscriber count over time	Implement topic TTL based on last activity
Subscription registrations never cleaned up	Track subscription count per topic	Clean up subscriptions on connection close
Wildcard subscriptions accumulating	Monitor wildcard subscription index size	Implement subscription deduplication
Consumer group metadata never removed	Track consumer group count and membership	Remove empty consumer groups after timeout

Message Queue Growth

Message queues can grow unbounded if consumers cannot keep up with producers or if acknowledgment tracking accumulates unacknowledged messages.

Queue Growth Issue	Monitoring Metric	Intervention Strategy
Slow consumer causing backlog	Queue depth per topic	Implement backpressure to throttle producers
Unacknowledged messages accumulating	Pending message count per consumer	Reduce acknowledgment timeout or remove slow consumers
Dead letter queue growing	DLQ message count	Implement DLQ retention policy with automatic purging
Message retention policy not enforced	Total message count across all topics	Audit retention policy enforcement and fix cleanup logic

Connection and Client State Leaks

Each client connection maintains state in multiple system components. Failure to clean up this state completely can cause gradual memory growth.

State Category	Cleanup Location	Leak Impact
Connection metadata	`Handler.connections` map	Memory growth proportional to connection churn
Consumer heartbeat tracking	`HeartbeatTracker.heartbeats` map	Ghost consumers reported as healthy
Pending message assignments	`AckTracker.pendingMessages` map	Messages never timeout or get reassigned
Consumer group membership	`GroupCoordinator.groups` map	Incorrect load balancing and assignment

Slow Consumer Detection and Handling

Slow consumers can cause system-wide performance degradation by creating backpressure that affects other consumers and publishers. Early detection and appropriate handling are crucial for maintaining system health.

Consumer Performance Monitoring

The broker must continuously monitor consumer performance to identify slow or failed consumers before they impact system performance.

Performance Metric	Calculation	Warning Threshold	Critical Threshold
Message Processing Rate	Messages acked per minute	< 50% of average group rate	< 25% of average group rate
Acknowledgment Latency	Time from delivery to acknowledgment	> 2x average group latency	> 5x average group latency
Queue Depth Growth	Rate of queue growth for consumer	> 100 messages/minute	> 1000 messages/minute
Heartbeat Latency	Time between heartbeat messages	> 2x heartbeat interval	> 5x heartbeat interval

Slow Consumer Mitigation Strategies

When slow consumers are detected, the broker has several options for maintaining system performance while giving consumers a chance to recover.

> **Design Principle**: Graceful degradation is better than system-wide failure. A slow consumer should impact only its own message processing, not affect other consumers or the entire system.

Mitigation Strategy	When to Apply	Implementation
Increased acknowledgment timeout	Temporary performance issues	Adaptively increase timeout based on recent performance
Consumer group rebalancing	Consumer consistently slow	Remove consumer from group and redistribute its messages
Message reassignment	Consumer stopped responding	Move pending messages to healthy consumers
Consumer circuit breaker	Repeated failures or timeouts	Stop assigning new messages until consumer recovers

Throughput Bottlenecks

Message queue throughput can be limited by various system components. Identifying the bottleneck component is crucial for effective performance tuning.

Protocol Layer Bottlenecks

The TCP protocol handler can become a bottleneck if it cannot parse and process frames quickly enough to keep up with client demand.

Bottleneck Indicator	Likely Cause	Optimization Approach
High CPU usage in frame parsing	Inefficient parsing algorithm	Optimize binary parsing, consider zero-copy techniques
Many connections in blocking read state	Single-threaded connection handling	Implement connection pooling or async I/O
Frame processing queue backing up	Slow command processing	Separate frame parsing from command execution
Memory allocations during parsing	Frequent buffer allocations	Use buffer pools and avoid allocation in hot paths

Topic Manager Bottlenecks

The topic manager handles all message routing and subscription management, making it a potential bottleneck under high load.

⚠️ **Pitfall: Single-Threaded Topic Manager** Many learners implement the topic manager as a single goroutine or thread that processes all topic operations sequentially. This becomes a bottleneck as the number of topics and message volume grows. The fix is to implement per-topic concurrency or use lock-free data structures for topic management.

Topic Manager Issue	Performance Impact	Scaling Solution
Global lock for all topic operations	All message routing serialized	Use per-topic locks or lock-free concurrent maps
Linear search for wildcard subscriptions	O(n) complexity per message	Build inverted index for wildcard patterns
Synchronous fanout delivery	Slow subscriber blocks all deliveries	Implement asynchronous delivery with buffering
Subscription index rebuilding	Periodic performance pauses	Use incremental index updates instead of rebuilds

Persistence Layer Bottlenecks

The Write-Ahead Log can become a significant bottleneck, especially if fsync is called synchronously for every message.

Persistence Bottleneck	Symptom	Optimization
Synchronous fsync per message	High write latency and low throughput	Implement group commit with periodic fsync
Single log file for all topics	Write contention and serialization	Use per-topic log files or partitioned logs
Inefficient record serialization	High CPU usage during writes	Optimize serialization format and use binary encoding
Log compaction blocking writes	Periodic throughput pauses	Implement background compaction with write-ahead buffering

Persistence and Recovery Issues

Persistence-related problems can be the most dangerous because they can lead to permanent data loss or system unavailability. These issues often only manifest under specific failure conditions or after extended operation.

Log Corruption Detection and Recovery

Append-only logs can become corrupted due to hardware failures, incomplete writes, or software bugs. Detecting corruption early and recovering gracefully is essential for data integrity.

Write-Time Corruption Prevention

The most effective approach to corruption is preventing it during write operations through careful implementation of atomic writes and data validation.

Corruption Source	Prevention Method	Recovery Strategy
Partial writes during crash	Use write + fsync + rename pattern for atomic updates	Detect partial records by length mismatch and truncate
Concurrent write access	Single writer with proper locking	Rebuild from valid records before corruption point
Hardware bit flips	Include checksums in record headers	Skip corrupted records and continue with next valid record
Software bugs writing invalid data	Validate records before writing	Use backup writer to maintain secondary log

Read-Time Corruption Detection

During log reading, whether for normal operation or crash recovery, the system must detect and handle corrupted records gracefully.

⚠️ **Pitfall: Failing Recovery Due to Single Corrupted Record** Many learners write recovery code that fails completely if any log record is corrupted. This can make the entire system unrecoverable due to a single bad record. The fix is to implement record-level error handling that skips corrupted records and continues processing valid ones.

Corruption Type	Detection Method	Handling Strategy
Invalid magic number	Check first 4 bytes of record	Skip bytes until next valid magic number found
Checksum mismatch	Recompute and compare checksum	Log error and skip record, continue with next
Invalid record length	Length exceeds remaining file size	Truncate file at last valid record
Malformed record data	JSON parsing or field validation fails	Skip record but preserve for manual inspection

Log Compaction Issues

Log compaction removes obsolete records to reclaim disk space, but it introduces complexity around maintaining data consistency during the compaction process.

Compaction Problem	Impact	Solution Approach
Compaction running during crash recovery	Incomplete state reconstruction	Disable compaction during recovery phase
Lost records during compaction	Data loss or inconsistent state	Use atomic log file replacement with verification
Compaction performance impact	System slowdown during compaction	Implement incremental compaction with rate limiting
Corrupted compacted log	Entire log becomes unusable	Maintain backup of original log until compaction verified

Crash Recovery Failures

The crash recovery process must rebuild the complete system state from the Write-Ahead Log. Recovery failures can prevent the broker from starting, making the system completely unavailable.

Recovery State Reconstruction

During recovery, the system must process WAL records in chronological order to rebuild topics, consumer groups, and pending message state. This process can fail in several ways.

Recovery Failure	Cause	Resolution
Incomplete topic state	Missing topic creation records	Implement topic auto-creation during recovery
Consumer group inconsistency	Group membership changes without proper logging	Rebuild group membership from consumer heartbeat records
Orphaned pending messages	Messages delivered but consumer state lost	Return orphaned messages to topic queues
Inconsistent message acknowledgments	ACK records without corresponding message records	Skip orphaned ACK records and continue recovery

Recovery Performance Issues

Large WAL files can make recovery extremely slow, potentially violating availability requirements. The recovery process must be optimized for both correctness and performance.

⚠️ **Pitfall: Linear Recovery Time with Log Size** Basic recovery implementations that process every record sequentially can take hours to recover from large logs. The fix is to implement checkpoint-based recovery that only processes records since the last checkpoint, combined with periodic background checkpointing.

Recovery Optimization	Benefit	Implementation Complexity
Checkpoint-based recovery	Skip processing most historical records	Medium - requires consistent state snapshots
Parallel recovery processing	Process independent records concurrently	High - requires dependency analysis
Recovery progress reporting	Operator visibility into recovery status	Low - add progress logging
Recovery validation	Verify recovered state consistency	Medium - requires state validation logic

Disk Space Management

Message brokers can consume large amounts of disk space, especially with retention policies that keep messages for extended periods. Running out of disk space can cause immediate system failure.

Storage Growth Monitoring

Proactive monitoring of disk space usage patterns helps prevent sudden space exhaustion and enables capacity planning.

Storage Metric	Monitoring Approach	Alert Threshold
Total disk usage	Monitor broker data directory size	> 80% of available space
Log growth rate	Track bytes written per hour/day	Sudden 10x increase in growth rate
Retention policy effectiveness	Messages purged vs messages added	Retention not keeping up with ingestion
DLQ growth	Dead letter queue size over time	> 1GB or growing faster than 1MB/day

Emergency Space Reclamation

When disk space becomes critically low, the system may need to take emergency actions to continue operating while more space is provisioned.

> **Operational Principle**: Availability is often more important than durability. It's better to lose some messages through aggressive cleanup than to make the entire system unavailable due to disk exhaustion.

Emergency Action	Risk Level	When to Apply
Reduce retention time temporarily	Medium - may lose important messages	< 95% disk usage
Purge oldest DLQ messages	Low - these are already failed messages	< 98% disk usage
Stop accepting new messages	High - impacts availability	< 99% disk usage
Emergency log compaction	Medium - may impact performance	Critical disk space remaining

Debugging Tools and Techniques

Effective debugging requires a systematic approach with the right tools and techniques. The complexity of message queue systems demands sophisticated debugging strategies that go beyond simple print statements.

Logging Strategies

Structured logging is essential for debugging distributed message queue systems where problems span multiple components and may be intermittent.

Component-Level Logging

Each system component should implement comprehensive logging that captures both normal operation and error conditions. The logging must be detailed enough to reconstruct the sequence of events that led to a problem.

Component	Critical Events to Log	Log Level	Information to Include
Protocol Handler	Connection establish/close, frame parsing errors	INFO/ERROR	Client address, connection ID, frame type, error details
Topic Manager	Topic creation, message publishing, subscription changes	INFO	Topic name, message ID, subscriber count, operation result
Consumer Groups	Group membership changes, rebalancing events	INFO	Group name, consumer ID, assignment changes, rebalance reason
Acknowledgment Tracker	Message assignments, acknowledgments, timeouts	DEBUG	Message ID, consumer ID, timeout values, retry count
Persistence Layer	WAL writes, fsync operations, recovery events	INFO	Record type, offset, record count, operation duration

Structured Log Format

Using structured logging formats like JSON enables powerful querying and analysis of log data. Each log entry should include consistent metadata that enables correlation across components.

⚠️ **Pitfall: Inconsistent Log Correlation IDs** Without consistent correlation IDs across components, it's nearly impossible to trace a single message's journey through the system. The fix is to generate a unique correlation ID when a message enters the system and include it in all related log entries.

Log Field	Purpose	Example Value
timestamp	Precise event timing	"2024-01-15T10:30:45.123Z"
component	Source component name	"topic_manager", "ack_tracker"
level	Log severity level	"INFO", "WARN", "ERROR"
correlation_id	Cross-component tracing	"msg_123456789"
event_type	Specific event category	"message_published", "consumer_timeout"
details	Event-specific data	{"topic": "orders", "consumer_id": "worker-3"}

Performance-Aware Logging

Logging can significantly impact system performance if not implemented carefully. High-throughput message brokers must balance logging detail with performance requirements.

Logging Strategy	Performance Impact	When to Use
Synchronous logging	High - blocks operation until log written	Only for critical errors
Asynchronous logging	Low - operations continue while logging happens	Normal and debug information
Sampling-based logging	Very low - only logs subset of events	High-frequency debug events
Conditional logging	Low - logging disabled unless debugging	Detailed tracing information

State Inspection Tools

The ability to inspect system state during operation is crucial for diagnosing problems that may not be evident from logs alone.

Runtime State Dumps

Administrative commands that dump current system state provide snapshots for problem analysis. These dumps should be comprehensive but also structured for easy analysis.

State Category	Information to Include	Access Method
Active Topics	Topic names, subscriber counts, queue depths, retention settings	HTTP endpoint `/admin/topics`
Consumer Groups	Group names, member lists, assignment distributions, rebalance status	HTTP endpoint `/admin/groups`
Pending Messages	Message IDs, assigned consumers, timeout deadlines, retry counts	HTTP endpoint `/admin/pending`
Connection Status	Client addresses, connection durations, heartbeat status, protocol versions	HTTP endpoint `/admin/connections`

Real-Time Monitoring Interfaces

Beyond static state dumps, real-time monitoring provides continuous visibility into system behavior. This is especially valuable for diagnosing intermittent problems or performance issues.

⚠️ **Pitfall: Monitoring Overhead Affecting Performance** Comprehensive monitoring can itself become a performance bottleneck if implemented inefficiently. The fix is to use efficient data structures for metrics collection and implement sampling for high-frequency events.

Monitoring Interface	Update Frequency	Typical Usage
Metrics Dashboard	Every 10 seconds	General system health monitoring
Live Connection View	Every 2 seconds	Debugging connection issues
Message Flow Tracer	Real-time	Following specific message through system
Performance Profiler	On-demand	Identifying performance bottlenecks

Memory and Resource Analysis

Message brokers must carefully manage memory and system resources to maintain stable operation under varying loads.

Memory Usage Profiling

Understanding memory allocation patterns helps identify leaks and optimize resource usage. Different categories of memory usage have different debugging approaches.

Memory Category	Profiling Approach	Common Issues
Message buffers	Track allocation and deallocation rates	Buffer pools not being reused
Topic metadata	Monitor growth over time	Topics not being garbage collected
Connection state	Count active connections and associated data	Connection cleanup not working
Persistence buffers	Profile WAL write operations	Buffers not being released after writes

Resource Leak Detection

Systematic approaches to detecting resource leaks help prevent gradual system degradation that can be difficult to diagnose after it becomes severe.

Resource Type	Detection Method	Prevention Strategy
Memory leaks	Monitor process memory growth over time	Implement resource pooling and explicit cleanup
File descriptor leaks	Check open file count via `/proc/pid/fd`	Ensure files are closed in defer statements or finally blocks
Goroutine leaks	Monitor goroutine count in Go runtime	Use context cancellation and proper channel cleanup
Database connections	Monitor connection pool usage	Implement connection timeouts and pool limits

Network and Protocol Debugging

Network-related issues can be subtle and intermittent, requiring specialized debugging techniques that operate at the protocol level.

Protocol Traffic Analysis

Capturing and analyzing actual network traffic helps diagnose protocol implementation issues that may not be evident from application logs.

Analysis Tool	Use Case	Information Provided
tcpdump/wireshark	Protocol implementation debugging	Raw packet contents and timing
netstat	Connection state analysis	TCP connection states and buffer usage
ss (socket statistics)	Advanced socket inspection	Detailed socket state and performance metrics
Application-level tracing	Message-level protocol debugging	Frame boundaries, command parsing, response generation

Connection State Debugging

TCP connection issues often require understanding the underlying socket state and network conditions.

⚠️ **Pitfall: Ignoring TCP Window Scaling and Buffer Issues** Network performance problems are often caused by TCP window scaling or buffer size mismatches rather than application-level issues. The fix is to monitor TCP performance metrics and tune socket buffer sizes appropriately.

TCP State Issue	Symptom	Debugging Approach
Connection hanging	Client appears frozen	Check TCP retransmission statistics and network connectivity
Slow data transfer	High latency despite low network load	Monitor TCP window sizes and buffer utilization
Frequent disconnections	Clients repeatedly reconnecting	Analyze TCP reset causes and keepalive settings
Port exhaustion	Connection refused errors	Monitor local port usage and adjust ephemeral port range

Implementation Guidance

This section provides practical tools and techniques for implementing effective debugging in your message queue system.

Technology Recommendations

Debugging Category	Simple Option	Advanced Option
Structured Logging	Standard library with JSON formatting	Dedicated logging framework (logrus, zap for Go)
Metrics Collection	In-memory counters with HTTP endpoint	Time-series database (Prometheus + Grafana)
State Inspection	JSON REST endpoints	Administrative web interface
Protocol Debugging	Custom packet logging	Wireshark with custom dissector
Performance Profiling	Built-in runtime profilers	Dedicated APM solution
Error Tracking	Log file analysis	Centralized error reporting service

File Structure for Debugging Infrastructure

```
internal/debug/
  logger.go              ← Structured logging setup
  metrics.go             ← Metrics collection
  admin_api.go           ← Administrative endpoints
  state_dumper.go        ← System state inspection
  profiler.go            ← Performance profiling
  protocol_tracer.go     ← Protocol-level debugging
```

Structured Logging Implementation

```go
// logger.go - Complete structured logging implementation

package debug

import (
    "context"
    "encoding/json"
    "io"
    "os"
    "time"
)

type Logger struct {
    output io.Writer
    level  LogLevel
}

type LogLevel int

const (
    DEBUG LogLevel = iota
    INFO
    WARN
    ERROR
)

type LogEntry struct {
    Timestamp     time.Time         `json:"timestamp"`
    Level         string            `json:"level"`
    Component     string            `json:"component"`
    EventType     string            `json:"event_type"`
    CorrelationID string            `json:"correlation_id,omitempty"`
    Message       string            `json:"message"`
    Details       map[string]interface{} `json:"details,omitempty"`
}

func NewLogger(output io.Writer, level LogLevel) *Logger {
    return &Logger{
        output: output,
```

```
        level:  level,
    }
}

func (l *Logger) Log(ctx context.Context, level LogLevel, component, eventType, message string, details
map[string]interface{}) {
    if level < l.level {
        return
    }

    entry := LogEntry{
        Timestamp:     time.Now().UTC(),
        Level:         levelToString(level),
        Component:     component,
        EventType:     eventType,
        Message:       message,
        Details:       details,
    }

    // Extract correlation ID from context if present
    if corrID := ctx.Value("correlation_id"); corrID != nil {
        entry.CorrelationID = corrID.(string)
    }

    data, _ := json.Marshal(entry)
    l.output.Write(append(data, '\n'))
}

func levelToString(level LogLevel) string {
    switch level {
    case DEBUG: return "DEBUG"
    case INFO: return "INFO"
    case WARN: return "WARN"
    case ERROR: return "ERROR"
    default: return "UNKNOWN"
    }
}
```

Administrative API for State Inspection

```go
// admin_api.go - Complete administrative interface

package debug

import (
    "encoding/json"
    "net/http"
    "strconv"
)

type AdminAPI struct {
    topicManager TopicManagerInterface
    groupCoordinator GroupCoordinatorInterface
    ackTracker AckTrackerInterface
}

func NewAdminAPI(tm TopicManagerInterface, gc GroupCoordinatorInterface, at AckTrackerInterface) *AdminAPI {
    return &AdminAPI{
        topicManager: tm,
        groupCoordinator: gc,
        ackTracker: at,
    }
}

func (api *AdminAPI) SetupRoutes(mux *http.ServeMux) {
    mux.HandleFunc("/admin/topics", api.handleTopics)
    mux.HandleFunc("/admin/groups", api.handleConsumerGroups)
    mux.HandleFunc("/admin/pending", api.handlePendingMessages)
    mux.HandleFunc("/admin/connections", api.handleConnections)
}

func (api *AdminAPI) handleTopics(w http.ResponseWriter, r *http.Request) {
    // TODO: Extract topic list from TopicManager
    // TODO: For each topic, collect: name, subscriber count, queue depth, retention policy
    // TODO: Include topic creation time and last activity timestamp
    // TODO: Return as JSON array with proper error handling
}

func (api *AdminAPI) handleConsumerGroups(w http.ResponseWriter, r *http.Request) {
```

```
    // TODO: Get all consumer groups from GroupCoordinator

    // TODO: For each group, collect: name, member count, assignment distribution

    // TODO: Include rebalancing status and last rebalance time

    // TODO: Show pending message count per consumer

}
```

Protocol Debugging Tools

```go
// protocol_tracer.go - Protocol-level debugging utilities

package debug

import (
    "fmt"
    "net"
    "time"
)

type ProtocolTracer struct {
    enabled bool
    logger  *Logger
}

func NewProtocolTracer(logger *Logger) *ProtocolTracer {
    return &ProtocolTracer{
        enabled: true,
        logger:  logger,
    }
}

func (pt *ProtocolTracer) TraceFrameReceived(conn net.Conn, frameType uint8, payload []byte) {
    if !pt.enabled {
        return
    }

    // TODO: Log frame type, payload size, connection info
    // TODO: Include frame parsing timing
    // TODO: Optionally dump payload hex for debugging
    // TODO: Correlate with connection ID for multi-connection debugging
}

func (pt *ProtocolTracer) TraceFrameSent(conn net.Conn, frameType uint8, payload []byte) {
    // TODO: Similar to TraceFrameReceived but for outbound frames
    // TODO: Include response timing (time since request received)
}
```

Performance Profiling Integration

```go
// profiler.go - Performance monitoring and profiling

package debug


import (

    "context"

    "runtime"

    "time"

)


type PerformanceProfiler struct {

    metrics *MetricsCollector

    logger  *Logger

}


func NewPerformanceProfiler(metrics *MetricsCollector, logger *Logger) *PerformanceProfiler {

    return &PerformanceProfiler{

        metrics: metrics,

        logger:  logger,

    }

}


func (pp *PerformanceProfiler) ProfileOperation(ctx context.Context, operation string, fn func() error) error {

    start := time.Now()



    // TODO: Record start time and operation name

    // TODO: Execute the operation function

    // TODO: Record completion time and success/failure

    // TODO: Update metrics with operation duration

    // TODO: Log slow operations (>100ms) with details

    // TODO: Return original error from operation



    return nil // placeholder

}


func (pp *PerformanceProfiler) CollectRuntimeMetrics() {

    // TODO: Collect Go runtime statistics (goroutines, memory, GC)

    // TODO: Record current connection counts
```

```
    // TODO: Update metrics with system resource usage

    // TODO: Run this periodically in background goroutine

}
```

Milestone Checkpoint for Debugging

After implementing debugging infrastructure:

1. **Verify Structured Logging**: Start the broker and publish a message. You should see structured JSON logs showing the message flow through components.
2. **Test Administrative API**: Use curl to access `/admin/topics` and verify you get topic information in JSON format.
3. **Validate Protocol Tracing**: Enable protocol tracing and observe frame-level debugging output when clients connect and send commands.
4. **Check Performance Profiling**: Run a load test and verify operation timing is being recorded and slow operations are logged.

Common Debugging Scenarios

Scenario	Investigation Steps	Tools to Use
Messages disappearing	Check logs for correlation_id, verify topic subscriptions, inspect pending message state	Structured logs, admin API
High memory usage	Profile memory allocations, check for growing maps, verify cleanup on disconnect	Runtime profiler, admin API
Slow message delivery	Trace protocol operations, check consumer group assignments, profile component performance	Protocol tracer, performance profiler
Connection issues	Enable protocol debugging, check TCP connection state, verify heartbeat processing	Protocol tracer, netstat

Future Extensions

> **Milestone(s):** All milestones (1-4) — potential enhancements that build upon the complete message queue implementation, demonstrating how good architectural design enables future growth

Mental Model: The Growing City Infrastructure

Think of our message queue as a city's infrastructure that has grown from a small town to a thriving metropolis. When the city was small, a single post office (our single broker) could handle all mail delivery efficiently. But as the population grows, we need multiple post offices (broker clustering) working together to serve different neighborhoods while maintaining city-wide coordination.

Similarly, as our simple routing system (topics with wildcards) served us well initially, a growing city needs more sophisticated transportation systems — express lanes for priority mail (content-based routing), different delivery services for different types of packages (exchange types), and high-speed bulk transport systems (batching and compression) to handle increased volume efficiently.

The beauty of good infrastructure design is that these improvements can be added incrementally without tearing down what already works. Each extension builds upon the solid foundation we've established, following the same architectural principles that made our single-node system reliable and maintainable.

Broker Clustering

The most significant architectural evolution for our message queue involves transforming from a single-node design to a distributed cluster. This extension addresses the fundamental limitations of single-node architectures: limited throughput capacity, single point of failure, and inability to scale beyond the resources of one machine.

Mental Model: Postal Service Network

Imagine transitioning from a single post office serving an entire city to a network of coordinated post offices, each serving specific neighborhoods but working together as one unified postal system. Each post office (broker node) maintains its own local operations while participating in city-wide coordination for mail routing, load balancing, and ensuring no mail is lost even if one office temporarily closes.

The key insight is that customers (producers and consumers) shouldn't need to know which specific post office handles their mail — they interact with "the postal service" as a single entity, while behind the scenes, the offices coordinate to provide seamless service.

Cluster Architecture Design

Our clustering design centers around a **leader-follower replication model** with **topic partitioning** for horizontal scaling. Each topic gets divided into multiple partitions, with each partition replicated across several broker nodes. One broker serves as the partition leader handling all reads and writes, while follower brokers maintain synchronized replicas for fault tolerance.

The cluster maintains several critical components that extend our existing architecture:

Component	Responsibility	Relationship to Existing System
`ClusterCoordinator`	Leader election, member discovery, partition assignment	Extends `GroupCoordinator` concepts to broker-level coordination
`PartitionManager`	Manages partition replicas, handles leader failover	Extends `TopicManager` to work with partitioned topics
`ReplicationEngine`	Synchronizes data between partition replicas	Extends `WAL` with inter-node replication capabilities
`ClientRouter`	Routes client requests to appropriate partition leaders	New component that extends protocol handling
`ConsensusModule`	Coordinates cluster membership and configuration changes	New component implementing Raft consensus algorithm

Leader Election and Consensus

The foundation of our clustering system relies on a **Raft consensus algorithm** implementation that ensures all broker nodes agree on cluster membership, partition leadership, and configuration changes. This builds naturally on our existing `WAL` infrastructure, as Raft itself is fundamentally about maintaining a replicated append-only log across multiple nodes.

> **Decision: Raft Consensus for Cluster Coordination**
>
> - **Context**: Clustered message brokers require coordination for leader election, configuration changes, and partition management. Multiple consensus algorithms exist (Raft, PBFT, Paxos).
> - **Options Considered**: Raft (leader-based, simpler implementation), Multi-Paxos (more complex but battle-tested), PBFT (Byzantine fault tolerant but overkill)
> - **Decision**: Implement Raft consensus algorithm
> - **Rationale**: Raft provides strong consistency guarantees with simpler implementation compared to Paxos. Leader-based approach aligns with our partition leadership model. Our target use case doesn't require Byzantine fault tolerance.
> - **Consequences**: Enables strong consistency for cluster metadata, requires odd number of nodes for fault tolerance, adds complexity but provides foundation for reliable clustering.

The `ConsensusModule` maintains a replicated state machine across all broker nodes. Critical cluster state changes — such as broker joins/leaves, partition reassignments, and configuration updates — get proposed as Raft log entries and committed only after a majority of nodes agree.

Partition Management and Replication

Our existing `TopicManager` evolves into a `PartitionManager` that handles multiple partitions per topic. Each partition operates as an independent message queue with its own `WAL` file and consumer group assignments. The partition key (derived from message headers or content) determines which partition receives each message, ensuring related messages maintain ordering within their partition.

Partition Replication State	Leader Actions	Follower Actions
`Healthy`	Accept writes, replicate to followers, serve reads	Replicate leader's WAL, maintain read replicas, monitor leader health
`UnderReplicated`	Continue serving, alert cluster coordinator	Catch up from leader or other healthy follower
`LeaderFailover`	N/A (leader is down)	Participate in leader election, take over if elected

The `ReplicationEngine` extends our `WAL` component to synchronize log entries across partition replicas. When the partition leader appends a message to its `WAL`, it simultaneously sends replication requests to all followers. The leader waits for acknowledgment from a configurable number of replicas (typically a majority) before confirming the write to the client.

Client Request Routing

Our existing protocol handler gains a `ClientRouter` component that transparently directs client requests to the appropriate partition leaders. Producers specify a partition key (or the router generates one via hash-based distribution), while consumers connect to consumer groups that span all partitions of a topic.

The router maintains a cached view of cluster topology — which brokers are alive, which partitions each broker leads, and how to reach each broker. When cluster topology changes (leader failover, partition reassignment), the router updates its cache and potentially redirects in-flight connections.

Consumer Group Rebalancing Across Partitions

Our `GroupCoordinator` extends to handle consumer groups distributed across multiple partitions and brokers. Instead of assigning individual messages to consumers, the coordinator assigns entire partitions to consumers. Each consumer in a group receives exclusive ownership of one or more partitions, consuming all messages from those partitions in order.

This partition-based assignment provides several advantages: consumers maintain message ordering within their assigned partitions, rebalancing becomes simpler (moving entire partitions rather than individual messages), and consumer failure recovery is faster (reassign partitions rather than redelivering individual messages).

Data Migration and Cluster Expansion

Adding new brokers to an existing cluster requires careful partition reassignment to maintain data availability during migration. The cluster coordinator orchestrates this process by:

1. Creating new partition replicas on the new brokers
2. Synchronizing these replicas with existing partition data
3. Promoting new replicas to leader status for some partitions
4. Decommissioning old replicas from overloaded brokers
5. Updating cluster routing tables atomically

Our existing `WAL` compaction mechanisms become critical during cluster expansion, as new brokers need efficient ways to catch up with potentially large amounts of historical data.

Advanced Routing Features

Beyond basic topic-based routing with wildcard subscriptions, production message queues often require more sophisticated routing mechanisms. These advanced features build upon our existing `TopicManager` component while introducing new routing abstractions that provide greater flexibility for complex messaging patterns.

Mental Model: Intelligent Mail Sorting System

Imagine upgrading from a simple postal system where mail gets routed based only on zip codes (topic names) to an intelligent sorting facility that can examine package contents, sender information, delivery priorities, and recipient preferences to make sophisticated routing decisions. Letters marked "urgent" go to express delivery, packages containing fragile items get special handling, and mail for VIP customers gets priority processing — all while maintaining the same simple interface for senders and receivers.

Content-Based Routing Implementation

Content-based routing allows messages to be delivered based on message payload content or header values, not just topic names. This extends our `WildcardSubscription` concept to support complex filtering expressions that examine message attributes.

Our routing system introduces a new `MessageFilter` abstraction that evaluates expressions against message content:

Filter Type	Example Expression	Matches Messages Where
`HeaderFilter`	`priority = "high" AND region = "us-west"`	Message headers contain specific key-value pairs
`PayloadFilter`	`$.order.amount > 1000`	JSON payload contains values meeting criteria
`CompositeFilter`	`topic LIKE "orders.*" AND priority = "urgent"`	Multiple conditions combine with boolean logic

The `ContentRouter` component extends our existing `TopicManager` to evaluate these filters during message fanout. Instead of simply matching topic names against wildcard patterns, the router evaluates each subscriber's filter expression against incoming messages.

Exchange Types and Routing Keys

Exchange types introduce the concept of pluggable routing algorithms, similar to RabbitMQ's exchange model. Our system defines several exchange types that can be mixed and matched per topic:

Exchange Type	Routing Behavior	Use Cases
`DirectExchange`	Exact routing key match	Point-to-point messaging, RPC patterns
`TopicExchange`	Wildcard pattern matching on routing keys	Hierarchical topic routing, complex subscriptions
`FanoutExchange`	Broadcast to all bound queues	Event broadcasting, cache invalidation
`HeadersExchange`	Route based on message headers	Content-based routing, metadata filtering

The `ExchangeManager` component implements this routing logic as a pluggable system where each topic can specify its exchange type and routing rules. This provides much greater flexibility than our current simple topic-based routing while maintaining backward compatibility.

Routing Key Hierarchies

Our current wildcard subscription system (using `*` and `#` patterns) extends to support more sophisticated hierarchical routing keys. These routing keys use dot-notation paths like `orders.electronics.laptops.urgent` that can be matched with pattern expressions like `orders.*.*.urgent` or `orders.electronics.#`.

The enhanced routing system supports:

- **Multi-level wildcards**: `orders.#` matches `orders.electronics.laptops` and `orders.books.fiction.scifi`
- **Single-level wildcards**: `orders.*.urgent` matches `orders.electronics.urgent` but not `orders.electronics.laptops.urgent`
- **Exact matching**: `orders.electronics.laptops` matches only that specific routing key
- **Compound patterns**: `orders.{electronics,books}.*.urgent` matches urgent orders for electronics or books

Message Transformation Pipelines

Advanced routing often requires message transformation — modifying, enriching, or filtering message content as it flows through the system. Our design introduces a `TransformationPipeline` concept where messages can pass through a series of transformation stages before reaching their final destinations.

Transformation Type	Purpose	Implementation
`MessageEnricher`	Add metadata or context information	Lookup additional data and merge into headers
`ContentFilter`	Remove sensitive information	Parse payload and redact specified fields
`FormatConverter`	Transform between data formats	Convert JSON to Avro, XML to JSON, etc.
`MessageSplitter`	Split compound messages into components	Extract array elements into separate messages
`MessageAggregator`	Combine related messages	Collect messages by correlation ID

These transformations integrate with our existing `Message` structure by creating new messages with transformed payloads while preserving delivery tracking and acknowledgment behavior.

Performance Optimizations

As message queue systems handle increasing throughput and scale, several performance optimization techniques become essential. These optimizations build upon our existing architecture while introducing new mechanisms for handling high-volume, low-latency messaging scenarios.

Mental Model: Highway Traffic Optimization

Think of our message queue performance optimizations like upgrading from city streets to modern highway systems. Individual cars (messages) traveling one-by-one through stop signs (individual processing) works fine for low traffic, but as volume increases, we need on-ramps that merge multiple cars into traffic flow (batching), HOV lanes for priority traffic (message prioritization), and efficient cargo trucks that carry multiple deliveries at once (compression and zero-copy techniques).

Each optimization maintains the same destination guarantees while dramatically improving throughput and reducing resource consumption under heavy load.

Message Batching Strategies

Batching groups multiple small operations into larger, more efficient operations. Our current system processes each message individually through the `PUBLISH` → `WAL` append → fanout delivery cycle. Batching optimizes this by processing multiple messages together at each stage.

The `BatchProcessor` component extends our existing `TopicManager` to collect messages into batches based on configurable criteria:

Batching Strategy	Trigger Condition	Trade-offs
`TimeBatching`	Collect messages for fixed time window (e.g., 10ms)	Lower latency variance, predictable max delay
`SizeBatching`	Batch when size reaches threshold (e.g., 1MB)	Better throughput, variable latency
`CountBatching`	Batch when message count reaches limit (e.g., 100 messages)	Simple implementation, balanced performance
`AdaptiveBatching`	Dynamic adjustment based on current load	Optimal performance but complex tuning

Our `WAL` component particularly benefits from batching, as it can write multiple log records in a single `fsync` operation rather than forcing disk synchronization for each individual message. This dramatically reduces I/O overhead while maintaining durability guarantees.

The batching system maintains our delivery guarantees by treating the entire batch as an atomic unit — either all messages in the batch are successfully processed, or none are confirmed to producers.

Message Compression Techniques

Compression reduces network bandwidth and storage requirements by encoding message payloads more efficiently. Our system implements compression at multiple levels:

Compression Level	Scope	Algorithm	CPU vs Space Trade-off
`MessageCompression`	Individual message payload	LZ4 (fast) or zstd (efficient)	Low CPU overhead, moderate space savings
`BatchCompression`	Entire message batch	zstd or gzip	Higher CPU overhead, better compression ratios
`LogCompression`	WAL file segments	zstd with dictionary	Best compression, highest CPU cost

The `CompressionManager` component handles compression transparently — producers send uncompressed messages, the broker compresses for storage and transmission, and consumers receive decompressed messages. This maintains API compatibility while providing performance benefits.

Compression decisions can be made dynamically based on message content. Text-heavy messages (JSON, XML) compress much better than binary data (images, encrypted content), so the system can skip compression for binary payloads to avoid CPU waste.

Zero-Copy Message Handling

Zero-copy techniques eliminate unnecessary memory copying operations during message processing. Our current implementation copies message data multiple times: from network buffer to protocol parser, from parser to message structure, from message structure to WAL writer, and from WAL to subscriber connections.

The `ZeroCopyProcessor` implements several optimization techniques:

Technique	Benefit	Implementation Complexity
`DirectBufferAccess`	Eliminate message parsing copies	Reference byte ranges in network buffers
`MemoryMappedFiles`	Eliminate WAL read/write copies	Map WAL files directly into process memory
`SendFileOptimization`	Eliminate subscriber delivery copies	Use kernel sendfile() system calls
`BufferPooling`	Eliminate allocation overhead	Reuse pre-allocated byte buffers

Zero-copy optimizations require careful buffer lifetime management to ensure data remains valid throughout the entire processing pipeline. The system uses reference counting and buffer pinning to prevent premature buffer reuse.

Asynchronous I/O and Pipeline Parallelism

Our current synchronous processing model — receive message → write WAL → fanout delivery → send acknowledgment — creates a sequential bottleneck. **Pipeline parallelism** overlaps these stages to process multiple messages simultaneously.

The `PipelineProcessor` decomposes message handling into parallel stages:

1. **Network I/O Stage**: Asynchronously receive messages from multiple connections
2. **Parsing Stage**: Parse protocol frames and validate message structure
3. **Persistence Stage**: Write messages to WAL with batching and compression
4. **Routing Stage**: Determine subscriber destinations using parallel filter evaluation
5. **Delivery Stage**: Asynchronously deliver to subscriber connections

Each stage runs in parallel with configurable buffer sizes between stages. This pipeline design dramatically increases throughput while maintaining message ordering guarantees within each topic partition.

Memory Management Optimizations

High-throughput message processing generates significant garbage collection pressure. Our optimizations focus on reducing allocation overhead and improving memory locality:

Optimization	Technique	Benefit
`ObjectPooling`	Reuse `Message`, `Consumer`, and buffer objects	Eliminate allocation overhead
`OffHeapStorage`	Store message payloads outside garbage collector scope	Reduce GC pressure
`CompactDataStructures`	Use arrays instead of maps where possible	Improve cache locality
`PreallocationStrategies`	Pre-size collections based on load patterns	Avoid dynamic resizing overhead

The `MemoryManager` component coordinates these optimizations, monitoring memory usage patterns and adjusting allocation strategies based on current load characteristics.

Decision: Incremental Performance Optimization Approach

- **Context**: Message queues can benefit from numerous performance optimizations, but implementing all simultaneously creates complexity and debugging challenges.
- **Options Considered**: Implement all optimizations together (faster performance but higher risk), implement incrementally with benchmarking (slower but safer), implement only when needed (reactive approach)
- **Decision**: Implement performance optimizations incrementally with comprehensive benchmarking between each change
- **Rationale**: Allows measurement of each optimization's individual impact, enables rollback if optimization introduces bugs, provides learning opportunities to understand performance characteristics
- **Consequences**: Longer development timeline but much higher confidence in system performance and stability. Enables data-driven optimization decisions rather than premature optimization.

Benchmarking and Performance Monitoring

Performance optimizations require sophisticated measurement capabilities to verify their effectiveness. Our system includes comprehensive benchmarking infrastructure that extends our existing `MetricsCollector`:

Benchmark Type	Measurement Focus	Key Metrics
`ThroughputBenchmark`	Messages processed per second	Peak throughput, sustained throughput, throughput variance
`LatencyBenchmark`	End-to-end message delivery time	P50, P95, P99 latencies, latency distribution
`ResourceBenchmark`	CPU, memory, disk, network utilization	Resource efficiency, bottleneck identification
`ScalabilityBenchmark`	Performance under varying load	Linear scaling factors, breaking points

The benchmarking system creates controlled load scenarios that simulate realistic usage patterns while measuring the impact of each optimization. This data-driven approach ensures that optimizations provide measurable benefits and don't introduce performance regressions.

Implementation Guidance

The future extensions described above represent significant architectural enhancements that build upon the solid foundation established in the core implementation. While these extensions add substantial complexity, they follow the same design principles and patterns established in the base system.

Technology Recommendations

Extension Area	Simple Option	Advanced Option
Clustering Consensus	Hashicorp Raft library (pre-built, battle-tested)	Custom Raft implementation (educational, full control)
Content Routing	JSONPath expressions with go-jsonpath	Custom expression language with ANTLR parser
Message Compression	Standard library gzip/lz4	Facebook Zstandard with custom dictionaries
Benchmarking	Go's built-in testing.B framework	Dedicated load testing with JMeter/Locust
Inter-Node Communication	gRPC with Protocol Buffers (type safety)	Custom TCP protocol (minimal overhead)

Recommended Extension Architecture

These extensions should be implemented as separate modules that integrate with the existing system through well-defined interfaces:

```
project-root/
  cmd/
    server/main.go              ← enhanced to support clustering
    benchmark/benchmark.go      ← performance testing utilities
  internal/
    core/                       ← existing core components
      protocol/
      topic/
      consumer/
      persistence/
    clustering/                 ← new clustering extensions
      coordinator.go            ← cluster membership and leadership
      replication.go           ← partition replication engine
      consensus.go             ← Raft consensus implementation
    routing/                    ← new advanced routing
      content_filter.go        ← content-based filtering
      exchange_manager.go      ← exchange type implementations
      transformation.go        ← message transformation pipelines
    optimization/               ← new performance optimizations
      batch_processor.go       ← message batching logic
      compression.go           ← compression algorithms
      zerocopy.go             ← zero-copy optimizations
      memory_manager.go        ← memory management
```

Core Extension Interfaces

The extension modules should integrate with the existing system through these interface contracts:

```go
// ClusteringExtension extends the base broker with multi-node capabilities
type ClusteringExtension interface {
    // JoinCluster connects this broker to an existing cluster
    // Returns cluster member ID and current topology
    JoinCluster(seedNodes []string) (memberID string, topology *ClusterTopology, error)

    // TODO 1: Establish connections to seed nodes using gRPC or TCP
    // TODO 2: Request cluster membership through Raft consensus
    // TODO 3: Receive initial partition assignments and replica data
    // TODO 4: Start replication streams for assigned partitions
    // TODO 5: Register this node as available for partition leadership
}

// AdvancedRouter extends TopicManager with content-based routing
type AdvancedRouter interface {
    // CreateContentSubscription registers a subscriber with message filtering
    // Filter expressions can examine headers, payload, or routing keys
    CreateContentSubscription(consumerID string, filter *MessageFilter) error

    // TODO 1: Parse filter expression into executable condition tree
    // TODO 2: Register subscriber with topic manager
    // TODO 3: Store filter condition for message evaluation
    // TODO 4: Optimize filter evaluation order for performance
}

// PerformanceOptimizer extends message processing with batching and compression
type PerformanceOptimizer interface {
    // EnableBatching configures message batching with specified strategy
    // Batching groups messages for more efficient processing
    EnableBatching(strategy BatchingStrategy, config BatchConfig) error

    // TODO 1: Create batch collector with specified strategy (time/size/count)
    // TODO 2: Modify message processing pipeline to collect batches
    // TODO 3: Update WAL writer to handle batch operations
    // TODO 4: Ensure atomic batch processing for delivery guarantees
}
```

Clustering Implementation Skeleton

The clustering extension requires careful coordination between multiple distributed components:

```go
// ClusterCoordinator manages cluster membership and partition assignments
type ClusterCoordinator struct {
    nodeID        string
    raftNode      *raft.Raft
    topology      *ClusterTopology
    partitionMgr  *PartitionManager
}

// HandleNodeJoin processes a new broker joining the cluster
func (c *ClusterCoordinator) HandleNodeJoin(joinRequest *JoinRequest) (*JoinResponse, error) {
    // TODO 1: Validate joining node credentials and compatibility
    // TODO 2: Propose cluster membership change through Raft consensus
    // TODO 3: Wait for majority agreement on membership change
    // TODO 4: Calculate new partition assignments including new node
    // TODO 5: Return cluster topology and initial partition assignments
    // TODO 6: Trigger rebalancing to distribute load to new node
}

// ReplicatePartition synchronizes partition data to follower nodes
func (c *ClusterCoordinator) ReplicatePartition(partitionID string, logEntries []*WALRecord) error {
    // TODO 1: Identify all follower nodes for this partition
    // TODO 2: Send replication requests to followers in parallel
    // TODO 3: Wait for acknowledgment from majority of followers
    // TODO 4: Handle follower failures by promoting new replicas
    // TODO 5: Update partition health status based on replication success
}
```

Content-Based Routing Implementation

Advanced routing builds upon the existing wildcard subscription infrastructure:

```go
// MessageFilter evaluates complex conditions against message content
type MessageFilter struct {
    Expression string                   // Filter expression (e.g., "priority > 5 AND region = 'us-west'")
    Compiled   *CompiledExpression      // Parsed and optimized filter
}

// ContentRouter extends TopicManager with advanced filtering capabilities
type ContentRouter struct {
    topicManager    *TopicManager        // Existing topic management
    filterEvaluator *FilterEvaluator     // Expression evaluation engine
    subscriptions   map[string][]*ContentSubscription  // Per-topic filter subscriptions
}

// EvaluateSubscriptions determines which subscribers should receive a message
func (r *ContentRouter) EvaluateSubscriptions(msg *Message) ([]*Consumer, error) {
    // TODO 1: Get basic topic subscribers using existing wildcard matching
    // TODO 2: Get content subscribers for this topic from subscription registry
    // TODO 3: Evaluate each content filter against message headers and payload
    // TODO 4: Combine wildcard subscribers with matching content subscribers
    // TODO 5: Return deduplicated list of target subscribers
}
```

Performance Optimization Implementation

Batching optimization requires careful coordination with existing message processing:

```go
// BatchProcessor groups individual messages for more efficient processing
type BatchProcessor struct {
    strategy      BatchingStrategy      // Time/size/count-based batching
    currentBatch  *MessageBatch        // Accumulating message batch
    flushTimer    *time.Timer          // Timer for time-based flushing
    walLogger     WALLogger            // Enhanced WAL with batch support
}

// CollectMessage adds a message to the current batch, flushing if needed
func (p *BatchProcessor) CollectMessage(msg *Message) error {
    // TODO 1: Add message to current batch buffer
    // TODO 2: Check if batch is ready for flushing (size/count/time)
    // TODO 3: If ready, flush current batch to WAL and subscribers
    // TODO 4: Create new empty batch for subsequent messages
    // TODO 5: Start/restart flush timer for time-based batching
}

// FlushBatch processes accumulated messages as atomic unit
func (p *BatchProcessor) FlushBatch(batch *MessageBatch) error {
    // TODO 1: Write all batch messages to WAL in single fsync operation
    // TODO 2: Perform topic routing for all messages in batch
    // TODO 3: Group subscriber deliveries by connection for efficiency
    // TODO 4: Send batch delivery notifications to all subscribers
    // TODO 5: Update metrics for batch size and processing latency
}
```

Extension Integration Points

These extensions integrate with the existing system at well-defined points:

1. **Protocol Handler Integration**: Extensions register new command types (e.g., `JOIN_CLUSTER`, `CONTENT_SUBSCRIBE`) that get routed to appropriate extension handlers
2. **Topic Manager Integration**: Advanced routing extends topic subscription logic while preserving existing wildcard functionality
3. **WAL Integration**: Clustering and batching extend WAL operations (replication streams, batch writes) while maintaining existing durability guarantees
4. **Metrics Integration**: All extensions report performance and health metrics through the existing `MetricsCollector` interface

Milestone Verification for Extensions

Each extension should be implemented and verified incrementally:

Clustering Extension Checkpoint:

- Start three broker nodes in cluster mode: `./broker --cluster --seed-nodes=node1:9001,node2:9002`
- Verify leader election: exactly one node should show as cluster leader in logs

- Test partition assignment: publish messages and verify they distribute across cluster nodes
- Test failover: stop leader node and verify new leader election within 30 seconds

Advanced Routing Checkpoint:

- Create content subscription: `CONTENT_SUBSCRIBE orders priority > 5`
- Publish test messages with various priority values
- Verify only high-priority messages reach content subscriber
- Test filter performance: 1000 messages/second should maintain sub-10ms routing latency

Performance Optimization Checkpoint:

- Enable message batching with 10ms time windows
- Measure baseline throughput vs batched throughput (expect 2-5x improvement)
- Monitor memory usage during high-throughput tests (should remain stable)
- Verify delivery guarantees preserved (no message loss or duplication)

These extensions represent natural evolution paths for the message queue system. Each builds upon the architectural foundation established in the core implementation while adding sophisticated capabilities needed for production-scale deployments. The key insight is that good initial design makes these extensions possible without fundamental rewrites — they enhance and extend rather than replace the existing system.

Implementation Guidance

Technology Recommendations

Extension Area	Simple Option	Advanced Option
Clustering Consensus	Hashicorp Raft library + gRPC transport	Custom Raft with optimized TCP protocol
Content Routing	JSONPath with github.com/oliveagle/jsonpath	Custom expression parser with ANTLR
Message Compression	Standard library compress/gzip	Facebook Zstandard with dictionaries
Performance Profiling	Built-in go test -bench + pprof	Custom metrics with Prometheus/Grafana
Inter-Node Communication	gRPC with Protocol Buffers	Custom binary protocol over TCP

Recommended Extension Architecture

Extensions should be implemented as separate modules that integrate through well-defined interfaces:

```
project-root/
  cmd/
    server/main.go              ← enhanced with extension loading
    cluster-node/main.go        ← clustered broker executable
    benchmark/main.go           ← performance testing utility
  internal/
    extensions/                 ← new extension framework
      clustering/
        coordinator.go          ← cluster membership management
        replication.go          ← partition replication engine
        consensus.go            ← Raft consensus implementation
        partition_manager.go    ← distributed partition handling
      routing/
        content_router.go       ← content-based message filtering
        exchange_manager.go     ← pluggable routing algorithms
        transform_pipeline.go   ← message transformation chain
        filter_evaluator.go     ← expression evaluation engine
      optimization/
        batch_processor.go      ← message batching coordination
        compression_mgr.go      ← compression algorithm selection
        zerocopy_buffers.go     ← zero-copy buffer management
        memory_pool.go          ← object pooling and reuse
  pkg/
    clustering/                 ← public clustering APIs
      client.go                 ← cluster-aware client library
      topology.go               ← cluster membership types
    routing/                    ← public routing APIs
      filters.go                ← message filter expressions
      exchanges.go              ← exchange type definitions
```

Core Extension Framework

```go
package extensions

import (
    "context"
    "github.com/message-queue/internal/core"
)

// Extension represents a pluggable system enhancement
type Extension interface {
    // Name returns the unique extension identifier
    Name() string

    // Initialize sets up the extension with broker dependencies
    Initialize(ctx context.Context, broker core.BrokerInterface) error

    // Start begins extension operation (called after all extensions initialized)
    Start(ctx context.Context) error

    // Stop gracefully shuts down the extension
    Stop(ctx context.Context) error

    // Health returns current extension health status
    Health() ExtensionHealth
}

// ExtensionRegistry manages loading and lifecycle of extensions
type ExtensionRegistry struct {
    extensions map[string]Extension
    broker     core.BrokerInterface
    logger     *Logger
}

// LoadExtension registers and initializes an extension
func (r *ExtensionRegistry) LoadExtension(ext Extension) error {
    // TODO 1: Validate extension name is unique and not already loaded
    // TODO 2: Call extension Initialize() with broker interface
    // TODO 3: Register extension in internal registry map
```

```
    // TODO 4: Log successful extension loading with name and version

    return nil

}
```

Clustering Extension Implementation

```go
package clustering

import (
    "context"
    "time"
    "github.com/hashicorp/raft"
)

// ClusteringExtension implements distributed broker coordination
type ClusteringExtension struct {
    nodeID          string
    raftNode        *raft.Raft
    topology        *ClusterTopology
    partitionMgr    *PartitionManager
    replicationMgr  *ReplicationManager
}

// Initialize sets up clustering with existing broker components
func (c *ClusteringExtension) Initialize(ctx context.Context, broker core.BrokerInterface) error {
    // TODO 1: Extract broker's topic manager and WAL for integration
    // TODO 2: Create Raft configuration with peer discovery
    // TODO 3: Initialize partition manager with topic data migration
    // TODO 4: Set up replication streams between cluster nodes
    // TODO 5: Register cluster-specific protocol commands (JOIN_CLUSTER, etc.)
    return nil
}

// JoinCluster connects this node to existing cluster or creates new cluster
func (c *ClusteringExtension) JoinCluster(seedNodes []string) (*ClusterMembership, error) {
    // TODO 1: If no seed nodes provided, bootstrap new single-node cluster
    // TODO 2: Otherwise, contact seed nodes to request cluster membership
    // TODO 3: Participate in Raft consensus to add this node to cluster
    // TODO 4: Receive current partition assignments and begin replication
    // TODO 5: Update local topology view and start partition management
    return nil, nil
}
```

```go
// PartitionManager handles distributed topic partitions
type PartitionManager struct {
    partitions   map[string]*PartitionReplica  // partitionID -> replica info
    assignments  map[string][]string           // partitionID -> replica nodeIDs
    localNode    string
    clusterTopo  *ClusterTopology
}

// CreatePartition sets up a new partition with specified replication factor
func (p *PartitionManager) CreatePartition(topicName string, partitionID string, replicationFactor int) error {
    // TODO 1: Select replica nodes based on cluster topology and load balancing
    // TODO 2: Create partition data structures on leader node (this node)
    // TODO 3: Send CreateReplica commands to selected follower nodes
    // TODO 4: Wait for replica creation confirmation from majority of nodes
    // TODO 5: Update cluster topology with new partition assignment
    // TODO 6: Begin replicating any existing topic messages to new partition
    return nil
}
```

Content-Based Routing Extension

```go
package routing

import (
    "encoding/json"
    "github.com/oliveagle/jsonpath"
)

// ContentRoutingExtension adds message filtering and transformation
type ContentRoutingExtension struct {
    filterEvaluator  *FilterEvaluator
    transformPipeline *TransformationPipeline
    subscriptions    map[string][]*ContentSubscription
    exchangeManager  *ExchangeManager
}

// MessageFilter represents a content-based subscription filter
type MessageFilter struct {
    Expression     string                   // Human-readable filter expression
    CompiledFilter *CompiledFilterExpression // Optimized evaluation tree
    FilterType     FilterType               // Header, Payload, or Composite
}

// FilterEvaluator efficiently evaluates filter expressions against messages
type FilterEvaluator struct {
    headerFilters  map[string]*HeaderFilterSet    // Pre-indexed header filters
    payloadFilters []*PayloadFilterExpression     // JSONPath-based payload filters
    cache          map[string]bool                // Filter result cache
}

// EvaluateMessage determines if message matches filter criteria
func (f *FilterEvaluator) EvaluateMessage(msg *core.Message, filter *MessageFilter) (bool, error) {
    // TODO 1: Check filter result cache for recent identical evaluations
    // TODO 2: For header filters, perform fast map lookup against message headers
    // TODO 3: For payload filters, parse JSON and evaluate JSONPath expressions
    // TODO 4: For composite filters, evaluate sub-expressions and combine with boolean logic
    // TODO 5: Cache evaluation result for performance optimization
    return false, nil
```

```
}


// CreateContentSubscription registers consumer with message filtering
func (c *ContentRoutingExtension) CreateContentSubscription(consumerID string, topicPattern string, filter string) error {
    // TODO 1: Parse and validate filter expression syntax
    // TODO 2: Compile filter into optimized evaluation structure
    // TODO 3: Register subscription with topic manager for base topic matching
    // TODO 4: Store content filter for additional evaluation during fanout
    // TODO 5: Add subscription to per-topic filter index for efficiency
    return nil
}
```

Performance Optimization Extension

```go
package optimization

import (
    "sync"
    "time"
    "unsafe"
)

// PerformanceExtension coordinates various throughput and latency optimizations
type PerformanceExtension struct {
    batchProcessor   *BatchProcessor
    compressionMgr   *CompressionManager
    memoryManager    *MemoryManager
    zeroCopyEngine   *ZeroCopyEngine
}

// BatchProcessor groups messages for more efficient processing
type BatchProcessor struct {
    strategy        BatchingStrategy
    currentBatch    *MessageBatch
    batchTimeout    *time.Timer
    maxBatchSize    int
    maxBatchDelay   time.Duration
    flushChannel    chan *MessageBatch
}

// CollectMessage adds message to current batch, flushing if thresholds exceeded
func (b *BatchProcessor) CollectMessage(msg *core.Message) error {
    // TODO 1: Add message to current batch buffer (thread-safe)
    // TODO 2: Check if batch size threshold reached (e.g., 100 messages)
    // TODO 3: Check if batch memory threshold reached (e.g., 1MB total)
    // TODO 4: If any threshold exceeded, flush current batch immediately
    // TODO 5: Otherwise, ensure timeout timer is running for time-based flush
    return nil
}

// FlushBatch processes accumulated messages as single atomic operation
```

```go
func (b *BatchProcessor) FlushBatch(batch *MessageBatch) error {
    // TODO 1: Write all batch messages to WAL in single fsync operation
    // TODO 2: Perform topic routing once per unique topic in batch
    // TODO 3: Group subscriber notifications by network connection
    // TODO 4: Send batch notifications to minimize network round-trips
    // TODO 5: Update throughput and latency metrics for entire batch
    return nil
}

// CompressionManager selects optimal compression for different message types
type CompressionManager struct {
    algorithms    map[string]CompressionAlgorithm  // name -> implementation
    selector      *CompressionSelector             // chooses algorithm per message
    dictionaries  map[string][]byte               // compression dictionaries
}

// CompressMessage selects appropriate algorithm and compresses payload
func (c *CompressionManager) CompressMessage(msg *core.Message) (*CompressedMessage, error) {
    // TODO 1: Analyze message payload to determine data type (JSON, binary, etc.)
    // TODO 2: Select compression algorithm based on payload characteristics
    // TODO 3: Apply compression using selected algorithm and any relevant dictionary
    // TODO 4: Return compressed message with algorithm metadata for decompression
    // TODO 5: Update compression ratio metrics for algorithm performance tracking
    return nil, nil
}

// ZeroCopyEngine eliminates unnecessary memory copying in message pipeline
type ZeroCopyEngine struct {
    bufferPool    *sync.Pool               // Reusable byte buffers
    mmapFiles     map[string][]byte        // Memory-mapped WAL files
    directBuffers map[*core.Connection][]byte // Connection-specific buffers
}

// ProcessMessageZeroCopy handles message without intermediate copying
func (z *ZeroCopyEngine) ProcessMessageZeroCopy(conn *core.Connection, frameData []byte) error {
    // TODO 1: Parse message header directly from network buffer (no copying)
    // TODO 2: Create message struct with pointers into original buffer
```

```
    // TODO 3: Write to WAL using memory-mapped file (kernel handles copying)

    // TODO 4: For fanout, use sendfile() system call to copy directly between sockets

    // TODO 5: Pin network buffer until all subscribers receive message

    return nil
}
```

Benchmarking and Performance Testing

```go
// cmd/benchmark/main.go

package main

import (
    "context"
    "flag"
    "fmt"
    "time"
    "github.com/message-queue/pkg/client"
)

// BenchmarkSuite runs comprehensive performance tests
type BenchmarkSuite struct {
    brokerAddr      string
    testDuration    time.Duration
    messageSize     int
    producerCount   int
    consumerCount   int
    results         *BenchmarkResults
}

// RunThroughputBenchmark measures maximum sustained message throughput
func (b *BenchmarkSuite) RunThroughputBenchmark() *ThroughputResult {
    // TODO 1: Create specified number of producer goroutines
    // TODO 2: Each producer publishes messages at maximum rate for test duration
    // TODO 3: Create consumer goroutines to acknowledge all messages
    // TODO 4: Measure total messages published and consumed
    // TODO 5: Calculate throughput (messages/second) and resource utilization
    return nil
}

// RunLatencyBenchmark measures end-to-end message delivery latency
func (b *BenchmarkSuite) RunLatencyBenchmark() *LatencyResult {
    // TODO 1: Publish messages with timestamp in payload at steady rate
    // TODO 2: Consumers measure delivery latency by comparing timestamps
    // TODO 3: Collect latency measurements for statistical analysis
    // TODO 4: Calculate P50, P95, P99 latency percentiles
```

```
    // TODO 5: Generate latency histogram for distribution analysis
    return nil
}

func main() {
    brokerAddr := flag.String("broker", "localhost:9001", "Broker address")
    duration := flag.Duration("duration", 60*time.Second, "Test duration")
    messageSize := flag.Int("msg-size", 1024, "Message payload size")
    producers := flag.Int("producers", 10, "Number of producer connections")
    consumers := flag.Int("consumers", 10, "Number of consumer connections")
    flag.Parse()

    suite := &BenchmarkSuite{
        brokerAddr:    *brokerAddr,
        testDuration:  *duration,
        messageSize:   *messageSize,
        producerCount: *producers,
        consumerCount: *consumers,
    }

    fmt.Println("Running message queue performance benchmark...")

    // Run baseline performance test
    baselineResults := suite.RunThroughputBenchmark()
    fmt.Printf("Baseline throughput: %.2f messages/second\n", baselineResults.MessagesPerSecond)

    // Run latency characterization
    latencyResults := suite.RunLatencyBenchmark()
    fmt.Printf("Latency P95: %v, P99: %v\n", latencyResults.P95, latencyResults.P99)

    // TODO: Add more sophisticated benchmark scenarios
    // - Gradual load increase to find breaking point
    // - Mixed message sizes to test batching effectiveness
    // - Consumer group rebalancing under load
    // - Network partition simulation for clustering tests
}
```

Extension Integration Testing

```go
// internal/extensions/integration_test.go

package extensions_test

import (
    "context"
    "testing"
    "time"
    "github.com/message-queue/internal/extensions/clustering"
    "github.com/message-queue/internal/extensions/routing"
    "github.com/message-queue/internal/extensions/optimization"
)

// TestClusteringExtensionIntegration verifies cluster coordination works correctly
func TestClusteringExtensionIntegration(t *testing.T) {
    // Create three-node cluster for testing
    nodes := make([]*TestBroker, 3)
    for i := range nodes {
        nodes[i] = NewTestBroker(t, fmt.Sprintf("node%d", i))
        nodes[i].LoadExtension(&clustering.ClusteringExtension{})
    }

    // Bootstrap cluster with first node
    err := nodes[0].clustering.BootstrapCluster()
    if err != nil {
        t.Fatalf("Failed to bootstrap cluster: %v", err)
    }

    // Join remaining nodes to cluster
    for i := 1; i < len(nodes); i++ {
        membership, err := nodes[i].clustering.JoinCluster([]string{nodes[0].Address()})
        if err != nil {
            t.Fatalf("Node %d failed to join cluster: %v", i, err)
        }

        if len(membership.Nodes) != i+1 {
            t.Errorf("Expected %d nodes in cluster, got %d", i+1, len(membership.Nodes))
```

```
        }
    }

    // TODO: Test partition distribution across cluster
    // TODO: Test leader election when current leader fails
    // TODO: Test replication consistency across nodes
    // TODO: Test cluster expansion and contraction
}

// TestContentRoutingIntegration verifies message filtering works with existing pub/sub
func TestContentRoutingIntegration(t *testing.T) {
    broker := NewTestBroker(t, "routing-test")
    broker.LoadExtension(&routing.ContentRoutingExtension{})

    // Create content-based subscription
    filter := `headers.priority = "high" AND headers.region = "us-west"`
    err := broker.routing.CreateContentSubscription("consumer1", "orders.*", filter)
    if err != nil {
        t.Fatalf("Failed to create content subscription: %v", err)
    }

    // Publish messages with different attributes
    testCases := []struct {
        topic    string
        headers  map[string]string
        shouldMatch bool
    }{
        {"orders.electronics", map[string]string{"priority": "high", "region": "us-west"}, true},
        {"orders.books", map[string]string{"priority": "low", "region": "us-west"}, false},
        {"orders.electronics", map[string]string{"priority": "high", "region": "europe"}, false},
    }

    for _, tc := range testCases {
        msg := &core.Message{
            Topic: tc.topic,
            Headers: tc.headers,
```

```
            Payload: []byte("test message"),
        }

        err := broker.PublishMessage(msg)
        if err != nil {
            t.Fatalf("Failed to publish message: %v", err)
        }

        // Verify consumer receives message only when filter matches
        received := broker.WaitForMessage("consumer1", 100*time.Millisecond)
        if tc.shouldMatch && received == nil {
            t.Errorf("Expected consumer to receive message for case %+v", tc)
        } else if !tc.shouldMatch && received != nil {
            t.Errorf("Expected consumer to NOT receive message for case %+v", tc)
        }
    }
}
```

Extension Development Guidelines

When implementing these extensions, follow these key principles:

1. **Incremental Integration**: Each extension should be implementable and testable independently, then integrated with other extensions
2. **Backward Compatibility**: Extensions should not break existing functionality — they enhance rather than replace
3. **Performance Measurement**: Every optimization should include before/after benchmarks proving its effectiveness
4. **Graceful Degradation**: Extensions should detect when they cannot operate (e.g., insufficient cluster nodes) and fall back to base functionality
5. **Configuration Flexibility**: Extensions should be configurable and disableable for different deployment scenarios

The extension framework provides a clean path for evolving the message queue system while maintaining the architectural clarity and reliability established in the base implementation.

Glossary

> **Milestone(s):** All milestones (1-4) — foundational terminology and concepts used throughout the entire message queue implementation

Mental Model: The Technical Dictionary

Think of this glossary as your message queue dictionary — just like learning a foreign language, understanding distributed systems requires mastering specific vocabulary. Each term has precise meaning in our context, and using the wrong term can lead to confusion during implementation. This glossary serves as your reference when reading the design document or debugging issues, ensuring everyone on the team speaks the same technical language.

Core Message Queue Concepts

Term	Definition	Context
Message Queue	An asynchronous communication mechanism where producers send messages to a queue and consumers retrieve them later	The fundamental abstraction our system provides
Producer	An application or service that publishes messages to topics	Also called Publisher in some contexts
Consumer	An application or service that subscribes to topics and processes messages	Receives messages from the broker
Broker	The central server that receives, stores, routes, and delivers messages between producers and consumers	Our main application component
Topic	A named channel or category that groups related messages together	Messages are published to topics and consumers subscribe to topics
Message	A unit of data containing a payload and metadata that flows through the system	Core data structure with ID, topic, payload, headers, and timestamps
Payload	The actual data content of a message, stored as raw bytes	Can contain JSON, XML, binary data, or any serialized format
Headers	Key-value metadata attached to messages for routing and processing hints	Similar to HTTP headers, used for content type, routing keys, etc.

Messaging Patterns

Term	Definition	Usage
Publish/Subscribe (Pub/Sub)	One-to-many messaging pattern where a single message is delivered to all subscribers of a topic	Default behavior for topic subscriptions
Point-to-Point	One-to-one messaging pattern where each message is delivered to exactly one consumer	Implemented through consumer groups
Fan-out	The process of delivering one published message to multiple subscribers	Happens automatically for all topic subscribers
Fan-in	Multiple producers sending messages to the same topic or consumer	Natural result of multiple publishers
Request-Reply	Synchronous communication pattern using correlation IDs to match responses to requests	Not directly implemented but can be built on top
Broadcast	Delivering a message to all connected consumers regardless of subscription	Special case of fan-out to all consumers

Consumer Group Concepts

Term	Definition	Implementation
Consumer Group	A set of consumers that work together to process messages from a topic with load balancing	Messages distributed across group members
Group Coordinator	Component responsible for managing consumer group membership and assignments	Handles joins, leaves, and rebalancing
Assignment Strategy	Algorithm determining how messages are distributed among consumer group members	Round-robin or sticky assignment
Round-Robin	Assignment strategy that distributes messages sequentially across available consumers	Default strategy for fair load distribution
Sticky Assignment	Assignment strategy that minimizes reassignment during rebalancing to preserve consumer state	Reduces reprocessing overhead
Rebalancing	Process of redistributing message assignments when consumer group membership changes	Triggered by joins, leaves, or failures
Generation Number	Version counter that prevents split-brain scenarios during rebalancing	Incremented with each rebalancing cycle
Inflight Messages	Messages that have been assigned to consumers but not yet acknowledged	Tracked for timeout and redelivery

Message Reliability and Acknowledgment

Term	Definition	Purpose
Acknowledgment (ACK)	Positive confirmation from consumer that message was successfully processed	Enables at-least-once delivery guarantee
Negative Acknowledgment (NACK)	Signal from consumer that message processing failed and should be retried	Triggers immediate redelivery to another consumer
At-Least-Once Delivery	Guarantee that messages will be delivered one or more times	Default delivery semantic with acknowledgment
At-Most-Once Delivery	Guarantee that messages will be delivered zero or one times	Lower reliability but no duplicates
Exactly-Once Delivery	Guarantee that messages will be delivered exactly one time	Complex to implement, not provided in our system
Message Timeout	Maximum time allowed for consumer to acknowledge message before redelivery	Prevents messages from being stuck with failed consumers
Redelivery	Reassigning a message to a different consumer after original assignment failed	Happens after timeout or NACK
Poison Message	A message that consistently causes processing failures and gets repeatedly redelivered	Moved to dead letter queue after max retry attempts

Persistence and Durability

Term	Definition	Implementation
Write-Ahead Log (WAL)	Append-only log that records all operations before they are applied to system state	Enables crash recovery and durability
Append-Only Log	Storage structure that only adds new entries and never modifies existing ones	Provides consistency and enables efficient replication
Checkpoint	Point-in-time snapshot of system state that enables faster recovery	Reduces WAL replay time during startup
Log Compaction	Process of removing obsolete records from the WAL to reclaim storage space	Maintains only the latest state for each key
Crash Recovery	Process of rebuilding system state from persistent logs after unexpected shutdown	Replays WAL records to restore broker state
fsync	File system operation that forces buffered data to be written to persistent storage	Ensures durability of acknowledged operations
Durability	Guarantee that acknowledged operations survive system crashes	Achieved through WAL and fsync
State Consolidation	Process of combining multiple log records into final state representation	Used during recovery to rebuild current state

Dead Letter Queue and Error Handling

Term	Definition	Usage
Dead Letter Queue (DLQ)	Special destination for messages that cannot be delivered successfully	Stores poison messages for later inspection
Retry Count	Number of times message delivery has been attempted	Incremented on each NACK or timeout
Max Retries	Maximum number of delivery attempts before sending message to DLQ	Prevents infinite redelivery loops
Replay	Process of reintroducing DLQ messages back into the normal processing flow	Enables recovery after fixing consumer bugs
Message Inspection	Examining DLQ messages to understand failure reasons	Administrative feature for debugging
Retention Policy	Rules determining how long messages are kept in DLQ before deletion	Prevents unbounded storage growth
Failure Reason	Descriptive text explaining why message processing failed	Stored with DLQ message for debugging

Flow Control and Backpressure

Term	Definition	Purpose
Backpressure	Flow control mechanism that slows down producers when consumers cannot keep up	Prevents system overload
Consumer Lag	Time delay between message publication and consumption	Key metric for triggering backpressure
Throttling	Artificially limiting publisher throughput to allow consumers to catch up	Primary backpressure implementation technique
Token Bucket	Rate limiting algorithm using refillable token pool	Controls publisher message rates
Capacity Status	Current health assessment indicating normal, warning, critical, or overloaded states	Determines when to apply backpressure
Lag Monitoring	Continuous tracking of consumer processing performance	Detects when intervention is needed
Flow Controller	Component responsible for implementing backpressure policies	Coordinates between lag monitoring and throttling

Networking and Protocol

Term	Definition	Implementation
Wire Protocol	Binary communication format between clients and broker	Custom protocol with message framing
Message Framing	Technique for preserving message boundaries over TCP streams	Length-prefixed frames with headers
Frame Header	Fixed-size header containing frame metadata like type and payload length	12-byte header with magic number and length
Protocol Multiplexing	Supporting multiple operation types on a single TCP connection	Different frame types for PUBLISH, SUBSCRIBE, ACK, NACK
Partial TCP Reads	TCP delivering data in smaller chunks than requested	Common pitfall requiring careful buffer handling
Connection Cleanup	Removing client state when TCP connection is closed	Prevents resource leaks and orphaned subscriptions
Heartbeat	Periodic messages to detect connection failures	Enables detection of failed consumers
Binary Protocol	Using binary encoding instead of text for efficiency	Chosen over JSON/HTTP for performance

Topic Management and Routing

Term	Definition	Features
Topic Auto-Creation	Automatically creating topics when first message is published	Simplifies client usage
Wildcard Subscription	Pattern-based subscription using * and # wildcards for hierarchical topics	Enables flexible routing patterns
Hierarchical Topic Naming	Organizing topics in path-like structure with forward slashes	Example: sensor/temperature/room1
Subscription Indexing	Optimized data structures for efficient wildcard pattern matching	Improves routing performance
Topic Retention	Policy determining how long messages are kept in topic queues	Time-based and size-based limits
Message Router	Component that determines which subscribers should receive each message	Handles wildcard matching and fanout

Monitoring and Observability

Term	Definition	Measurement
Metrics Collection	Gathering performance and health measurements from system components	Time-series data for monitoring
Time-Series Database	Storage optimized for chronological data with timestamps	Tracks metrics over time
Queue Depth	Number of messages waiting to be consumed in a topic	Key metric for system health
Throughput	Number of messages processed per unit time	Measured in messages/second
Consumer Lag Time	Time difference between message publication and consumption	Critical performance indicator
Error Rate	Percentage of operations that result in failures	Health indicator for system reliability
Health Check	Active verification of component functionality	Automated system health monitoring
Administrative API	HTTP REST endpoints for inspecting and managing broker state	Operational interface for monitoring

System Architecture and Scaling

Term	Definition	Scope
Single-Node Deployment	Running entire message broker on one server instance	Current implementation scope
Horizontal Scaling	Adding more broker nodes to handle increased load	Future clustering capability
Leader-Follower Replication	One leader handles writes while followers maintain synchronized copies	Clustering pattern for high availability
Partition-Based Scaling	Dividing topics into independent partitions for parallel processing	Kafka-style scaling approach
Consensus Algorithm	Protocol ensuring all cluster nodes agree on decisions	Raft or similar for distributed coordination
Split-Brain Prevention	Ensuring consistent state during network partitions	Critical for distributed systems

Testing and Development

Term	Definition	Usage
Unit Testing	Testing individual components in isolation with mocked dependencies	First line of defense for code quality
Integration Testing	Testing component interactions with real implementations	Verifies end-to-end functionality
Chaos Testing	Deliberate failure injection to test system resilience	Validates error handling and recovery
Mock Objects	Test doubles that simulate dependency behavior	Enables isolated component testing
Milestone Verification	Testing that milestone requirements are actually met	Ensures learning objectives are achieved
Test Data Builders	Utilities for creating valid test objects with targeted modifications	Simplifies test setup
Failure Injection	Simulating specific error conditions during testing	Tests error handling paths

Error Handling and Resilience

Term	Definition	Implementation
Graceful Degradation	Continuing to operate with reduced functionality during failures	Maintains core services when components fail
Circuit Breaker	Pattern that prevents calls to failing components	Stops cascading failures
Automatic Healing	Self-recovery mechanisms without manual intervention	Reduces operational overhead
Failure Classification	Categorizing failures by type, severity, and recovery strategy	Enables appropriate response to different failures
Exponential Backoff	Progressively increasing delay between retry attempts	Prevents overwhelming recovering services
Cascading Failure	Failures that propagate through system components	Major risk in distributed systems
Timeout Detection	Identifying failures based on operation duration limits	Primary failure detection mechanism

Performance and Optimization

Term	Definition	Benefit
Zero-Copy Optimization	Eliminating unnecessary memory copying operations	Reduces CPU overhead and improves throughput
Batching	Processing multiple messages together as a unit	Improves efficiency through amortized costs
Memory-Mapped Files	Mapping files directly into process memory space	Faster file I/O with OS-managed caching
Connection Pooling	Reusing TCP connections across multiple operations	Reduces connection establishment overhead
Buffer Pooling	Reusing memory buffers to reduce garbage collection	Improves memory allocation performance
Compression	Reducing message size through encoding algorithms	Saves network bandwidth and storage

Configuration and Operational Terms

Term	Definition	Purpose
Retention Policy	Rules for message lifecycle management including time and size limits	Prevents unbounded resource usage
Timeout Configuration	Settings for various operation timeouts like ACK deadline	Balances reliability with performance
Capacity Thresholds	Limits that trigger backpressure or other protective measures	Prevents system overload
Log Rotation	Managing WAL file size through archival and cleanup	Maintains reasonable disk usage
Administrative Interface	Management API for operational tasks	Enables monitoring and troubleshooting

Implementation-Specific Terms

Term	Definition	Context
Length-Prefixed Framing	Frame format where header contains payload size	Our specific wire protocol design
Protocol Magic Number	Fixed bytes at message start for validation	0x4D515545 in our protocol
WAL Record Types	Different categories of operations stored in write-ahead log	MessagePublished, MessageAcked, ConsumerJoined, etc.
State Transitions	Valid changes between message lifecycle states	Created → Pending → Assigned → Acknowledged
Assignment Freeze	Pausing new message assignments during rebalancing	Ensures consistent group state
Drain Timeout	Maximum time to wait for pending acknowledgments during shutdown	Graceful shutdown behavior

Common Antipatterns and Pitfalls

Term	Definition	Problem
Head-of-Line Blocking	Slow message preventing processing of subsequent messages	Can stall entire queue
Thundering Herd	Many consumers competing for resources simultaneously	Overwhelms system during recovery
Split-Brain	Different parts of system having inconsistent state	Causes data corruption and conflicts
Resource Leak	Gradual consumption of system resources without cleanup	Eventually causes system failure
Scope Creep	Uncontrolled expansion of project requirements	Derails learning objectives
Ghost Consumers	Disconnected consumers still receiving message assignments	Causes message delivery delays

Future Extension Terms

Term	Definition	Application
Content-Based Routing	Message delivery based on payload content rather than just topic	Advanced routing capability
Message Transformation	Modifying message content during routing	Data format conversion
Exchange Types	Different routing algorithms like direct, topic, fanout, headers	RabbitMQ-style routing flexibility
Clustering	Multi-node deployment with coordination and replication	High availability and scalability
Cross-Data Center Replication	Synchronizing message brokers across geographic regions	Disaster recovery and global scale

Industry Standard Protocols and Systems

Term	Definition	Relevance
AMQP	Advanced Message Queuing Protocol - industry standard	Reference for feature comparison
RESP	Redis Serialization Protocol - simple text protocol	Alternative protocol design
Apache Kafka	Distributed streaming platform with high throughput	Inspiration for partitioning and replication
RabbitMQ	Feature-rich message broker with flexible routing	Reference implementation for features
Redis Pub/Sub	Simple publish/subscribe messaging	Lightweight alternative comparison

State and Lifecycle Management

Term	Definition	States
Message States	Lifecycle phases from creation to completion	Created, Pending, Assigned, Acknowledged, DeadLetter
Consumer States	Connection and processing status	Connected, Subscribed, Processing, Disconnected
Group States	Consumer group coordination status	Forming, Stable, Rebalancing
System States	Overall broker health and capacity	Normal, Warning, Critical, Overloaded
Connection States	TCP connection lifecycle	Connecting, Connected, Authenticated, Closing

---

Implementation Guidance

This glossary serves as your reference throughout the development process. When you encounter unfamiliar terms in the design document or during implementation, return to this section for clarification.

Terminology Consistency

Maintain strict consistency in terminology throughout your implementation. Use these exact terms in variable names, comments, and documentation:

Category	Consistent Usage
Components	`TopicManager`, `GroupCoordinator`, `AckTracker`, `FlowController`, `DLQManager`
Data Types	`Message`, `Consumer`, `ConsumerGroup`, `Topic`, `WALRecord`
Operations	`PublishMessage`, `AssignMessage`, `ProcessAck`, `TriggerRebalance`
States	`StatePending`, `StateAssigned`, `DeliveryPending`, `CircuitOpen`
Constants	`PUBLISH`, `SUBSCRIBE`, `ACK`, `NACK`, `ProtocolMagic`

Documentation Standards

When writing code comments and documentation, use these terminology guidelines:

```go
// Good: Uses consistent, precise terminology

// ProcessAck handles message acknowledgment and updates consumer group assignment

func (a *AckTracker) ProcessAck(messageID string, consumerID string) error {

    // Find pending message in acknowledgment tracking table

    // Update message state from StatePending to StateAcknowledged

    // Notify group coordinator of successful processing

}


// Bad: Inconsistent, vague terminology

// HandleAck does ack stuff for messages

func (a *AckTracker) HandleAck(id string, consumer string) error {

    // Process the acknowledgment somehow

    // Update some state

}
```

Logging and Monitoring Vocabulary

Use consistent terminology in log messages and metrics:

```go
// Structured logging with consistent terminology
log.Info("message-published",
    "topic", msg.Topic,
    "message-id", msg.ID,
    "producer-id", msg.ProducerID,
    "payload-size", len(msg.Payload))

log.Warn("consumer-lag-detected",
    "consumer-group", groupName,
    "lag-duration", lagTime,
    "queue-depth", queueDepth,
    "backpressure-status", "warning")
```

Error Message Standards

Use precise terminology in error messages to aid debugging:

```go
// Good: Specific, actionable error messages
fmt.Errorf("message acknowledgment timeout: messageID=%s consumerID=%s deadline=%v",
    messageID, consumerID, deadline)

fmt.Errorf("consumer group rebalancing failed: groupName=%s trigger=%s error=%w",
    groupName, trigger, err)

// Bad: Vague, generic error messages
fmt.Errorf("something went wrong with message processing")
fmt.Errorf("group error: %w", err)
```

Testing Terminology

Use consistent terms in test names and assertions:

```go
// Test names using standard terminology
func TestTopicManager_PublishMessage_FanoutToAllSubscribers(t *testing.T)
func TestGroupCoordinator_TriggerRebalance_RoundRobinAssignment(t *testing.T)
func TestAckTracker_ProcessTimeout_RedeliveryToAvailableConsumer(t *testing.T)

// Assertion messages with precise terminology
assert.Equal(t, StatePending, msg.State, "message should be in pending state after assignment")
assert.True(t, len(group.Members) == expectedCount, "consumer group should have correct member count after rebalancing")
```

Common Terminology Mistakes

Avoid these common inconsistencies:

Instead of	Use	Reason
"client", "connection"	`Consumer` when referring to message processing entity	Distinguishes network connection from logical consumer
"queue", "channel"	`Topic` for message routing destination	Aligns with pub/sub terminology
"worker", "processor"	`Consumer` for message processing entity	Consistent with consumer group concepts
"confirm", "receipt"	`ACK` or `acknowledgment` for positive confirmation	Standard messaging terminology
"failure", "error"	`NACK` for processing failure signal	Distinguishes from system errors
"restart", "retry"	`redelivery` for message reassignment	Specific to message lifecycle