Objective

Build a CLI-based background job queue system called queuectl.
This system should manage background jobs with worker processes, handle retries using exponential backoff, and maintain a Dead Letter Queue (DLQ) for permanently failed jobs.
 Problem Overview
You need to implement a minimal, production-grade job queue system that supports:

Enqueuing and managing background jobs
Running multiple worker processes
Retrying failed jobs automatically with exponential backoff
Moving jobs to a Dead Letter Queue after exhausting retries
Persistent job storage across restarts
All operations accessible through a CLI interface
 Job Specification
Each job must contain at least the following fields:

{
"id": "unique-job-id",
"command": "echo 'Hello World'",
"state": "pending",
"attempts": 0,
"max_retries": 3,
"created_at": "2025-11-04T10:30:00Z",
"updated_at": "2025-11-04T10:30:00Z"
}
 Job Lifecycle
StateDescriptionpendingWaiting to be picked up by a workerprocessingCurrently being executedcompletedSuccessfully executedfailedFailed, but retryabledeadPermanently failed (moved to DLQ) CLI Commands
Your tool must support the following commands:
CategoryCommand ExampleDescriptionEnqueuequeuectl enqueue '{"id":"job1","command":"sleep 2"}'Add a new job to the queueWorkersqueuectl worker start --count 3Start one or more workersqueuectl worker stopStop running workers gracefullyStatusqueuectl statusShow summary of all job states & active workersList Jobsqueuectl list --state pendingList jobs by stateDLQqueuectl dlq list / queuectl dlq retry job1View or retry DLQ jobsConfigqueuectl config set max-retries 3Manage configuration (retry, backoff, etc.)⚙ System Requirements
Job ExecutionEach worker must execute the specified command (e.g. sleep 2, echo hello, etc.)
Exit codes should determine success or failure.
Commands that fail or are not found should trigger retries.

Retry & BackoffFailed jobs retry automatically.
Implement exponential backoff:
delay = base ^ attempts seconds
Move to DLQ after max_retries.
PersistenceJob data must persist across restarts.
Use file storage (JSON) or SQLite/embedded DB or anything which you think is best for this usecase.
Worker ManagementMultiple workers can process jobs in parallel.
Prevent duplicate processing (locking required).
Implement graceful shutdown (finish current job before exit).
ConfigurationAllow configurable retry count and backoff base via CLI.
 Expected Test Scenarios
Candidates are expected to ensure the following:

Basic job completes successfully.
Failed job retries with backoff and moves to DLQ.
Multiple workers process jobs without overlap.
Invalid commands fail gracefully.
Job data survives restart.
 Must-Have Deliverables
Your submission must include:

✅ Working CLI application (queuectl)
✅ Persistent job storage
✅ Multiple worker support
✅ Retry mechanism with exponential backoff
✅ Dead Letter Queue
✅ Configuration management
✅ Clean CLI interface (commands & help texts)
✅ Comprehensive README.md
✅ Code structured with clear separation of concerns
✅ At least minimal testing or script to validate core flows
 README Expectations
Your README.md should cover:

Setup Instructions — How to run locally
Usage Examples — CLI commands with example outputs
Architecture Overview — Job lifecycle, data persistence, worker logic
Assumptions & Trade-offs — Decisions made, any simplifications
Testing Instructions — How to verify functionality
 Evaluation Criteria
CriteriaWeightDescriptionFunctionality40%Core features (enqueue, worker, retry, DLQ)Code Quality20%Structure, readability, maintainabilityRobustness20%Handles edge cases and concurrency safelyDocumentation10%Clear setup and usage instructionsTesting10%Demonstrates correctness and reliability Bonus Features (Optional)
Extra credit will be given for:

Job timeout handling
Job priority queues
Scheduled/delayed jobs (run_at)
Job output logging
Metrics or execution stats
Minimal web dashboard for monitoring,I WANT TO DO THIS IN JAVA YoUR TASK IS TO EXPLAIN THIS PROJECT MAKE ME A LLD AND HLD

Now I'll create comprehensive high-level and low-level design documentation for the queuectl background job queue system.

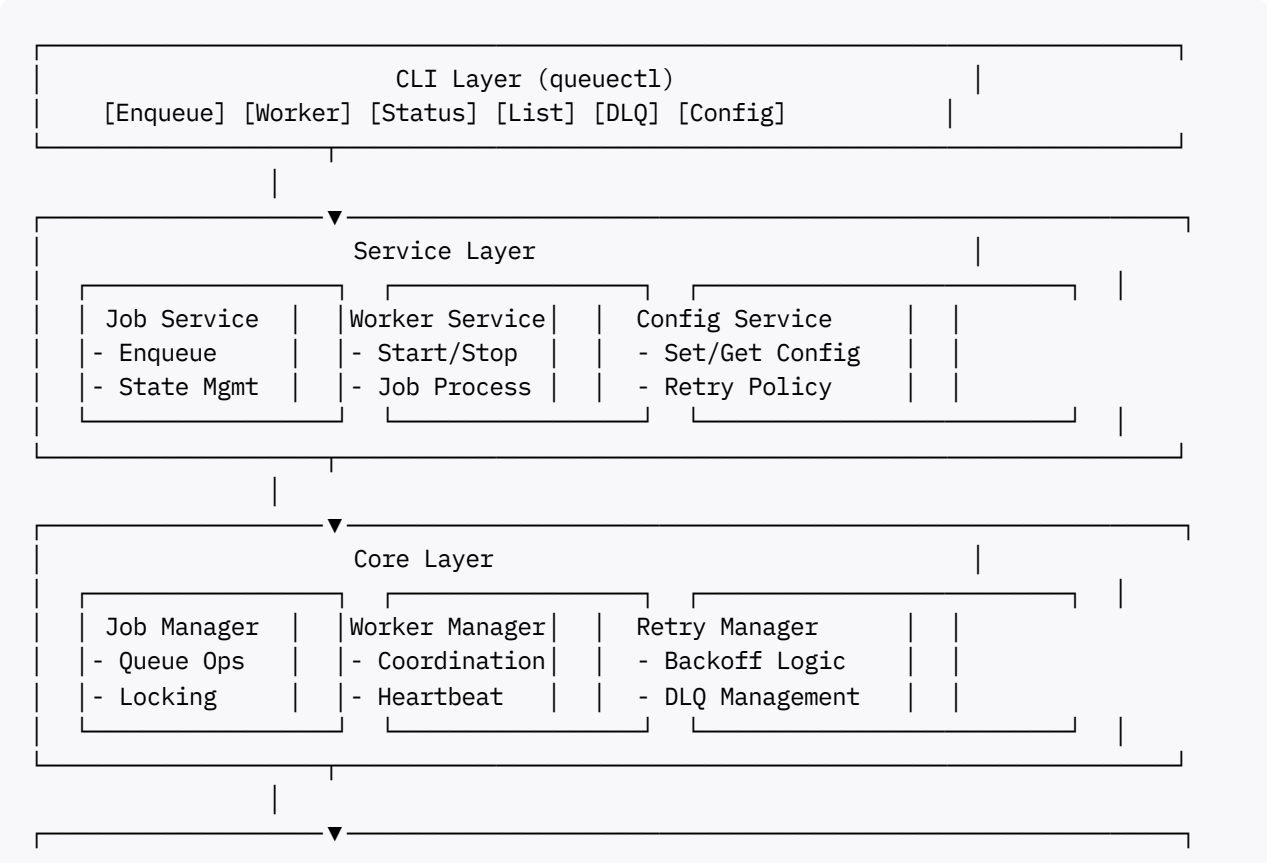
QueueCTL: CLI-Based Background Job Queue System - Design Document

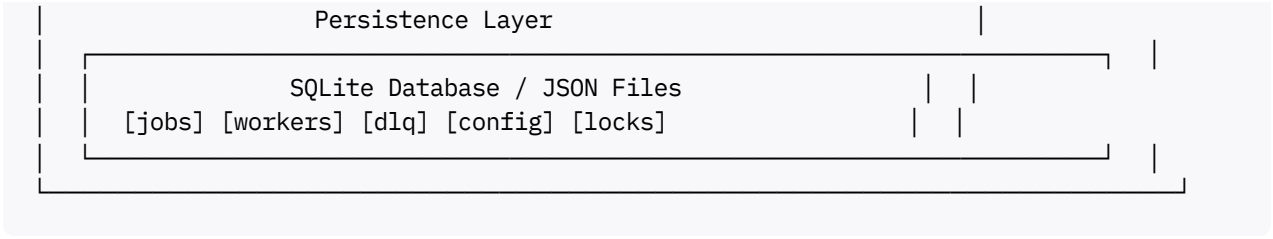
High-Level Design (HLD)

System Overview

QueueCTL is a production-grade, CLI-based job queue system that manages background task execution with multiple worker processes, automatic retry mechanisms using exponential backoff, and a Dead Letter Queue (DLQ) for handling permanently failed jobs. The system provides persistent storage across restarts and offers complete job lifecycle management through an intuitive command-line interface.[1] [2] [3] [4]

Architecture Diagram





Key Components

1. CLI Layer (CommandLine Interface)[3] [4]

- Built using Picocli framework for robust command parsing
- Handles user interactions and command routing
- Provides help text and command validation
- Subcommands: enqueue, worker, status, list, dlq, config

2. Service Layer

- **JobService**: Manages job enqueueing, state transitions, and retrieval
- **WorkerService**: Controls worker lifecycle (start/stop/status)
- **ConfigService**: Handles system configuration management
- **DLQService**: Manages Dead Letter Queue operations

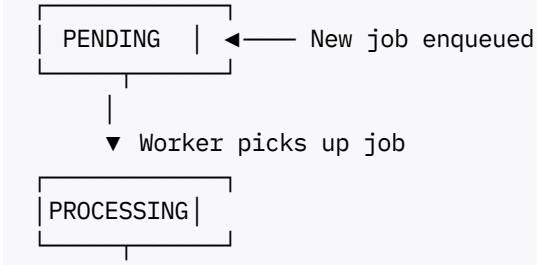
3. Core Layer[2] [1]

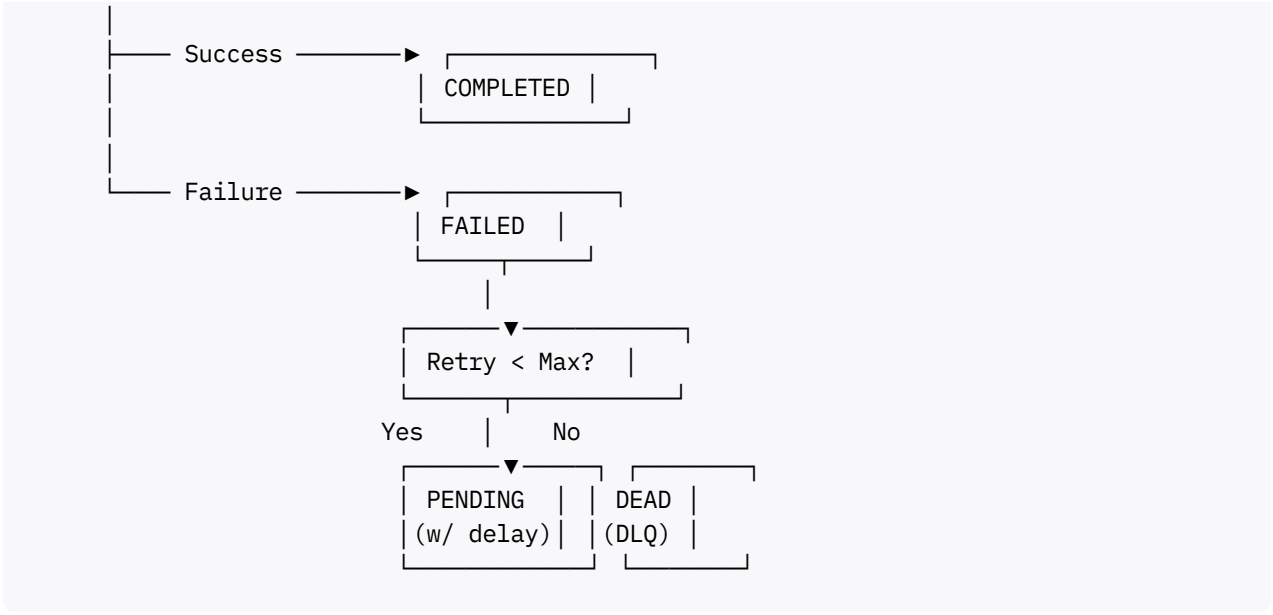
- **JobManager**: Central job queue operations and state management
- **WorkerManager**: Worker pool coordination and job assignment
- **RetryManager**: Implements exponential backoff and retry logic[5] [6]
- **LockManager**: Prevents duplicate job processing[7] [8]

4. Persistence Layer[9] [10]

- SQLite database for structured data storage
- Tables: jobs, workers, dlq_jobs, config, job_locks
- Ensures data survives application restarts
- Transaction management for data integrity

Job Lifecycle





Design Patterns Applied[11] [12] [13] [14]

1. Command Pattern

- Each CLI command encapsulates request as object
- Decouples command invocation from execution
- Enables queuing, logging, and undo operations

2. Worker Pattern[15] [16]

- Master-Worker architecture for parallel processing
- Multiple workers process jobs concurrently
- Worker pool managed by WorkerManager

3. Strategy Pattern

- Different retry strategies (exponential backoff)
- Configurable command execution strategies
- Pluggable persistence backends

4. Repository Pattern

- JobRepository, WorkerRepository, ConfigRepository
- Abstract data access layer
- Easy to switch persistence mechanisms

5. Observer Pattern

- Job state change notifications
- Worker status monitoring
- Event-driven architecture for logging

Data Flow

Job Enqueue Flow:

```
User → CLI → JobService → JobManager → JobRepository → SQLite
                                  ↓
                            Create Job Record
                            State: PENDING
```

Job Processing Flow:[1] [2]

```
Worker Thread → WorkerManager → JobManager → Acquire Lock
                                        ↓
                                  Update State: PROCESSING
                                        ↓
                                  Execute Command
                                        ↓
                                        |
                              ┌─────────┴─────────┐
                         Success                 Failure
                              ↓                     ↓
                    Update: COMPLETED    Calculate Backoff → Retry
                                                    ↓
                                          Max Retries? → DLQ (DEAD)
```

Concurrency & Locking Strategy[8] [17] [7]

1. Job-Level Locking

- Pessimistic locking using database transactions
- `SELECT FOR UPDATE` prevents duplicate processing
- Lock acquired before state change to PROCESSING

2. Worker Coordination[16] [15]

- Each worker runs in separate thread
- Worker heartbeat mechanism for failure detection
- Graceful shutdown coordination via shutdown hooks[18] [19] [20]

3. File-Based Locking (Optional)

- FileLock for inter-process coordination
- Prevents multiple queuectl instances conflicts
- Lock file: `.queuectl.lock`

Retry & Backoff Strategy[6] [21] [5]

Exponential Backoff Formula:[22]

```
delay = base^attempts (in seconds)
```

Example with base=2:

- Attempt 1: 2^1 = 2 seconds
- Attempt 2: 2^2 = 4 seconds
- Attempt 3: 2^3 = 8 seconds
- Max retries: 3 (configurable)

Implementation Details:

- Retry scheduled by updating job's `next_retry_at` timestamp
- Workers skip jobs with future `next_retry_at`
- Jitter optional to prevent thundering herd[5]

Dead Letter Queue (DLQ)[23] [24] [25]

Purpose:

- Store permanently failed jobs after exhausting retries
- Enable manual investigation and replay
- Prevent main queue clogging[23]

DLQ Operations:

- Automatic move after max_retries exceeded
- List DLQ jobs with failure reasons
- Retry individual jobs from DLQ
- Bulk retry or purge operations

Configuration Management

Configurable Parameters:

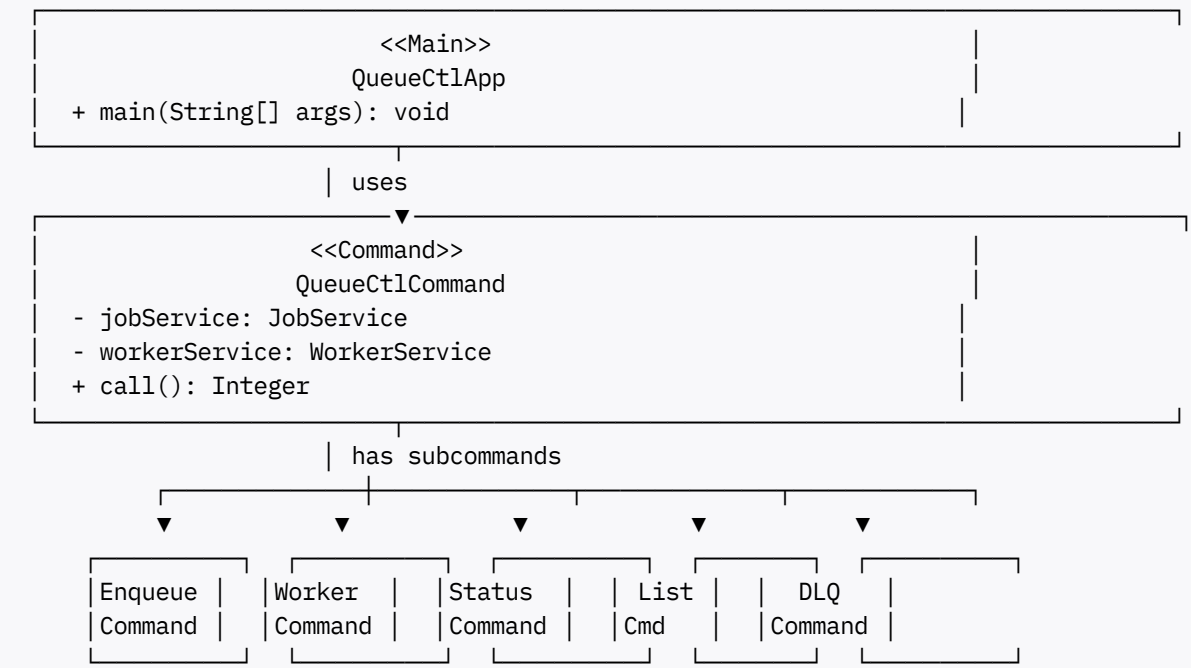
```
max-retries: 3              # Maximum retry attempts
backoff-base: 2             # Exponential backoff base
worker-count: 3             # Default worker count
worker-poll-interval: 1000  # Poll interval in ms
graceful-shutdown-timeout: 30 # Seconds to wait for workers
```

High-Level Technology Stack

- **Language**: Java 17+
- **CLI Framework**: Picocli 4.x[4] [3]
- **Database**: SQLite with JDBC
- **Concurrency**: Java ExecutorService, ThreadPool[26] [16]
- **Build Tool**: Maven/Gradle
- **Logging**: SLF4J + Logback
- **Testing**: JUnit 5, Mockito

Low-Level Design (LLD)

Class Diagram



Core Classes

1. Job Entity

```
package com.queuectl.model;

import java.time.Instant;
import java.util.UUID;

public class Job {
    private String id;
    private String command;
```

```
    private JobState state;
    private int attempts;
    private int maxRetries;
    private Instant createdAt;
    private Instant updatedAt;
    private Instant nextRetryAt;
    private String errorMessage;
    private String output;

    public enum JobState {
        PENDING,
        PROCESSING,
        COMPLETED,
        FAILED,
        DEAD
    }

    // Constructor
    public Job(String command, int maxRetries) {
        this.id = UUID.randomUUID().toString();
        this.command = command;
        this.state = JobState.PENDING;
        this.attempts = 0;
        this.maxRetries = maxRetries;
        this.createdAt = Instant.now();
        this.updatedAt = Instant.now();
    }

    // Getters and Setters
    public String getId() { return id; }
    public void setId(String id) { this.id = id; }

    public String getCommand() { return command; }
    public void setCommand(String command) { this.command = command; }

    public JobState getState() { return state; }
    public void setState(JobState state) {
        this.state = state;
        this.updatedAt = Instant.now();
    }

    public int getAttempts() { return attempts; }
    public void incrementAttempts() { this.attempts++; }

    public int getMaxRetries() { return maxRetries; }
    public void setMaxRetries(int maxRetries) { this.maxRetries = maxRetries; }

    public Instant getCreatedAt() { return createdAt; }
    public Instant getUpdatedAt() { return updatedAt; }

    public Instant getNextRetryAt() { return nextRetryAt; }
    public void setNextRetryAt(Instant nextRetryAt) {
        this.nextRetryAt = nextRetryAt;
    }

    public String getErrorMessage() { return errorMessage; }
```

```
    public void setErrorMessage(String errorMessage) {
        this.errorMessage = errorMessage;
    }

    public String getOutput() { return output; }
    public void setOutput(String output) { this.output = output; }

    public boolean canRetry() {
        return attempts < maxRetries;
    }

    public boolean isReadyForProcessing() {
        if (state != JobState.PENDING) return false;
        if (nextRetryAt == null) return true;
        return Instant.now().isAfter(nextRetryAt);
    }
}
```

2. JobRepository

```
package com.queuectl.repository;

import com.queuectl.model.Job;
import com.queuectl.model.Job.JobState;
import java.sql.*;
import java.time.Instant;
import java.util.*;

public class JobRepository {
    private final Connection connection;

    public JobRepository(Connection connection) {
        this.connection = connection;
        initializeSchema();
    }

    private void initializeSchema() {
        String sql = """
            CREATE TABLE IF NOT EXISTS jobs (
                id TEXT PRIMARY KEY,
                command TEXT NOT NULL,
                state TEXT NOT NULL,
                attempts INTEGER DEFAULT 0,
                max_retries INTEGER DEFAULT 3,
                created_at TEXT NOT NULL,
                updated_at TEXT NOT NULL,
                next_retry_at TEXT,
                error_message TEXT,
                output TEXT
            );

            CREATE INDEX IF NOT EXISTS idx_state
                ON jobs(state);
            CREATE INDEX IF NOT EXISTS idx_next_retry
                ON jobs(next_retry_at) WHERE state = 'PENDING';
```

```
        """;

        try (Statement stmt = connection.createStatement()) {
            stmt.executeUpdate(sql);
        } catch (SQLException e) {
            throw new RuntimeException("Failed to initialize schema", e);
        }
    }

    public void save(Job job) throws SQLException {
        String sql = """
            INSERT OR REPLACE INTO jobs
            (id, command, state, attempts, max_retries,
             created_at, updated_at, next_retry_at, error_message, output)
            VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
        """;

        try (PreparedStatement pstmt = connection.prepareStatement(sql)) {
            pstmt.setString(1, job.getId());
            pstmt.setString(2, job.getCommand());
            pstmt.setString(3, job.getState().name());
            pstmt.setInt(4, job.getAttempts());
            pstmt.setInt(5, job.getMaxRetries());
            pstmt.setString(6, job.getCreatedAt().toString());
            pstmt.setString(7, job.getUpdatedAt().toString());
            pstmt.setString(8, job.getNextRetryAt() != null ?
                job.getNextRetryAt().toString() : null);
            pstmt.setString(9, job.getErrorMessage());
            pstmt.setString(10, job.getOutput());
            pstmt.executeUpdate();
        }
    }

    public Optional<Job> findById(String id) throws SQLException {
        String sql = "SELECT * FROM jobs WHERE id = ?";

        try (PreparedStatement pstmt = connection.prepareStatement(sql)) {
            pstmt.setString(1, id);
            ResultSet rs = pstmt.executeQuery();

            if (rs.next()) {
                return Optional.of(mapResultSetToJob(rs));
            }
            return Optional.empty();
        }
    }

    public List<Job> findByState(JobState state) throws SQLException {
        String sql = "SELECT * FROM jobs WHERE state = ? ORDER BY created_at";

        try (PreparedStatement pstmt = connection.prepareStatement(sql)) {
            pstmt.setString(1, state.name());
            ResultSet rs = pstmt.executeQuery();

            List<Job> jobs = new ArrayList<>();
            while (rs.next()) {
```

```
                jobs.add(mapResultSetToJob(rs));
            }
            return jobs;
        }
    }

    // Acquire job with pessimistic locking
    public Optional<Job> acquireNextPendingJob() throws SQLException {
        connection.setAutoCommit(false);

        String sql = """
            SELECT * FROM jobs
            WHERE state = 'PENDING'
            AND (next_retry_at IS NULL OR next_retry_at <= ?)
            ORDER BY created_at
            LIMIT 1
            FOR UPDATE
        """;

        try (PreparedStatement pstmt = connection.prepareStatement(sql)) {
            pstmt.setString(1, Instant.now().toString());
            ResultSet rs = pstmt.executeQuery();

            if (rs.next()) {
                Job job = mapResultSetToJob(rs);
                job.setState(JobState.PROCESSING);
                save(job);
                connection.commit();
                return Optional.of(job);
            }

            connection.commit();
            return Optional.empty();
        } catch (SQLException e) {
            connection.rollback();
            throw e;
        } finally {
            connection.setAutoCommit(true);
        }
    }

    public Map<JobState, Long> getStateCounts() throws SQLException {
        String sql = "SELECT state, COUNT(*) as count FROM jobs GROUP BY state";
        Map<JobState, Long> counts = new EnumMap<>(JobState.class);

        try (Statement stmt = connection.createStatement();
             ResultSet rs = stmt.executeQuery(sql)) {

            while (rs.next()) {
                JobState state = JobState.valueOf(rs.getString("state"));
                long count = rs.getLong("count");
                counts.put(state, count);
            }
        }

        return counts;
```

```
    }

    private Job mapResultSetToJob(ResultSet rs) throws SQLException {
        Job job = new Job(
            rs.getString("command"),
            rs.getInt("max_retries")
        );

        job.setId(rs.getString("id"));
        job.setState(JobState.valueOf(rs.getString("state")));
        // Set other fields...

        return job;
    }
}
```

3. RetryManager

```
package com.queuectl.core;

import com.queuectl.model.Job;
import com.queuectl.model.Job.JobState;
import java.time.Instant;

public class RetryManager {
    private final int backoffBase;
    private final boolean useJitter;

    public RetryManager(int backoffBase, boolean useJitter) {
        this.backoffBase = backoffBase;
        this.useJitter = useJitter;
    }

    /**
     * Calculate exponential backoff delay
     * delay = base ^ attempts (in seconds)
     */
    public long calculateBackoffSeconds(int attempts) {
        long delay = (long) Math.pow(backoffBase, attempts);

        if (useJitter) {
            // Add random jitter: 0 to 25% of delay
            long jitter = (long) (Math.random() * delay * 0.25);
            delay += jitter;
        }

        return delay;
    }

    /**
     * Schedule job for retry with exponential backoff
     */
    public void scheduleRetry(Job job) {
        job.incrementAttempts();
```

```
        if (job.canRetry()) {
            // Calculate next retry time
            long delaySeconds = calculateBackoffSeconds(job.getAttempts());
            Instant nextRetry = Instant.now().plusSeconds(delaySeconds);

            job.setNextRetryAt(nextRetry);
            job.setState(JobState.PENDING);
        } else {
            // Move to DLQ
            job.setState(JobState.DEAD);
        }
    }

    /**
     * Handle job failure
     */
    public void handleFailure(Job job, String errorMessage) {
        job.setErrorMessage(errorMessage);
        scheduleRetry(job);
    }
}
```

4. WorkerThread

```
package com.queuectl.worker;

import com.queuectl.core.JobManager;
import com.queuectl.core.RetryManager;
import com.queuectl.model.Job;
import com.queuectl.model.Job.JobState;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.Optional;
import java.util.concurrent.TimeUnit;

public class WorkerThread implements Runnable {
    private static final Logger logger = LoggerFactory.getLogger(WorkerThread.class);

    private final String workerId;
    private final JobManager jobManager;
    private final RetryManager retryManager;
    private final int pollIntervalMs;
    private volatile boolean running = true;

    public WorkerThread(String workerId, JobManager jobManager,
                        RetryManager retryManager, int pollIntervalMs) {
        this.workerId = workerId;
        this.jobManager = jobManager;
        this.retryManager = retryManager;
        this.pollIntervalMs = pollIntervalMs;
    }
```

```
    @Override
    public void run() {
        logger.info("Worker {} started", workerId);

        while (running) {
            try {
                Optional<Job> jobOpt = jobManager.acquireNextJob();

                if (jobOpt.isPresent()) {
                    Job job = jobOpt.get();
                    processJob(job);
                } else {
                    // No jobs available, sleep
                    Thread.sleep(pollIntervalMs);
                }
            } catch (InterruptedException e) {
                logger.info("Worker {} interrupted", workerId);
                Thread.currentThread().interrupt();
                break;
            } catch (Exception e) {
                logger.error("Worker {} encountered error", workerId, e);
            }
        }

        logger.info("Worker {} stopped", workerId);
    }

    private void processJob(Job job) {
        logger.info("Worker {} processing job {}: {}",
            workerId, job.getId(), job.getCommand());

        try {
            // Execute command
            CommandResult result = executeCommand(job.getCommand());

            if (result.exitCode == 0) {
                // Success
                job.setState(JobState.COMPLETED);
                job.setOutput(result.output);
                logger.info("Job {} completed successfully", job.getId());
            } else {
                // Failure
                String error = String.format("Command failed with exit code %d: %s",
                    result.exitCode, result.error);
                retryManager.handleFailure(job, error);
                logger.warn("Job {} failed: {}", job.getId(), error);
            }

            jobManager.updateJob(job);

        } catch (Exception e) {
            logger.error("Error processing job {}", job.getId(), e);
            retryManager.handleFailure(job, e.getMessage());
            jobManager.updateJob(job);
        }
    }
```

```
    private CommandResult executeCommand(String command) {
        try {
            ProcessBuilder pb = new ProcessBuilder("sh", "-c", command);
            pb.redirectErrorStream(true);
            Process process = pb.start();

            // Read output
            StringBuilder output = new StringBuilder();
            try (BufferedReader reader = new BufferedReader(
                    new InputStreamReader(process.getInputStream()))) {
                String line;
                while ((line = reader.readLine()) != null) {
                    output.append(line).append("\n");
                }
            }

            // Wait for completion
            boolean finished = process.waitFor(5, TimeUnit.MINUTES);
            if (!finished) {
                process.destroyForcibly();
                throw new RuntimeException("Command timeout");
            }

            return new CommandResult(
                process.exitValue(),
                output.toString(),
                ""
            );

        } catch (Exception e) {
            return new CommandResult(-1, "", e.getMessage());
        }
    }

    public void shutdown() {
        logger.info("Shutting down worker {}", workerId);
        running = false;
    }

    private static class CommandResult {
        final int exitCode;
        final String output;
        final String error;

        CommandResult(int exitCode, String output, String error) {
            this.exitCode = exitCode;
            this.output = output;
            this.error = error;
        }
    }
}
```

5. WorkerManager

```
package com.queuectl.core;

import com.queuectl.worker.WorkerThread;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.ArrayList;
import java.util.List;
import java.util.UUID;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class WorkerManager {
    private static final Logger logger = LoggerFactory.getLogger(WorkerManager.class);

    private final JobManager jobManager;
    private final RetryManager retryManager;
    private final int pollIntervalMs;

    private ExecutorService executorService;
    private List<WorkerThread> workers;
    private boolean running = false;

    public WorkerManager(JobManager jobManager, RetryManager retryManager,
                         int pollIntervalMs) {
        this.jobManager = jobManager;
        this.retryManager = retryManager;
        this.pollIntervalMs = pollIntervalMs;
        this.workers = new ArrayList<>();
    }

    public synchronized void startWorkers(int count) {
        if (running) {
            throw new IllegalStateException("Workers already running");
        }

        logger.info("Starting {} workers", count);

        executorService = Executors.newFixedThreadPool(count);
        workers = new ArrayList<>();

        for (int i = 0; i < count; i++) {
            String workerId = "worker-" + UUID.randomUUID().toString().substring(0, 8);
            WorkerThread worker = new WorkerThread(
                workerId, jobManager, retryManager, pollIntervalMs
            );
            workers.add(worker);
            executorService.submit(worker);
        }

        running = true;

        // Register shutdown hook
```

```
        Runtime.getRuntime().addShutdownHook(new Thread(this::gracefulShutdown));

        logger.info("Workers started successfully");
    }

    public synchronized void stopWorkers() {
        if (!running) {
            return;
        }

        logger.info("Stopping workers");
        gracefulShutdown();
    }

    private void gracefulShutdown() {
        running = false;

        // Signal all workers to stop
        for (WorkerThread worker : workers) {
            worker.shutdown();
        }

        // Shutdown executor
        executorService.shutdown();

        try {
            // Wait for workers to finish current jobs
            if (!executorService.awaitTermination(30, TimeUnit.SECONDS)) {
                logger.warn("Workers did not stop gracefully, forcing shutdown");
                executorService.shutdownNow();
            } else {
                logger.info("All workers stopped gracefully");
            }
        } catch (InterruptedException e) {
            logger.error("Interrupted during shutdown", e);
            executorService.shutdownNow();
            Thread.currentThread().interrupt();
        }
    }

    public boolean isRunning() {
        return running;
    }

    public int getWorkerCount() {
        return workers.size();
    }
}
```

6. CLI Commands (Using Picocli)

```
package com.queuectl.cli;

import picocli.CommandLine;
import picocli.CommandLine.Command;
import picocli.CommandLine.Option;
import picocli.CommandLine.Parameters;

import com.queuectl.service.JobService;
import com.queuectl.service.WorkerService;
import com.queuectl.model.Job;

import java.util.concurrent.Callable;

@Command(
    name = "queuectl",
    mixinStandardHelpOptions = true,
    version = "queuectl 1.0",
    description = "CLI-based background job queue system",
    subcommands = {
        EnqueueCommand.class,
        WorkerCommand.class,
        StatusCommand.class,
        ListCommand.class,
        DLQCommand.class,
        ConfigCommand.class
    }
)
public class QueueCtlCommand implements Callable<Integer> {

    @Override
    public Integer call() {
        CommandLine.usage(this, System.out);
        return 0;
    }
}

@Command(name = "enqueue", description = "Add a new job to the queue")
class EnqueueCommand implements Callable<Integer> {

    @Parameters(index = "0", description = "Job specification in JSON format")
    private String jobJson;

    @Override
    public Integer call() {
        try {
            JobService jobService = new JobService();
            Job job = jobService.enqueueFromJson(jobJson);
            System.out.println("Job enqueued successfully: " + job.getId());
            return 0;
        } catch (Exception e) {
            System.err.println("Error: " + e.getMessage());
            return 1;
        }
    }
```

```
}

@Command(name = "worker", description = "Manage worker processes")
class WorkerCommand implements Callable<Integer> {

    @Command(name = "start", description = "Start workers")
    int start(
        @Option(names = {"--count", "-c"}, defaultValue = "3",
                description = "Number of workers to start")
        int count
    ) {
        try {
            WorkerService workerService = new WorkerService();
            workerService.startWorkers(count);
            System.out.println("Started " + count + " workers");

            // Keep running until interrupted
            Thread.currentThread().join();
            return 0;
        } catch (InterruptedException e) {
            System.out.println("Workers stopping...");
            return 0;
        } catch (Exception e) {
            System.err.println("Error: " + e.getMessage());
            return 1;
        }
    }

    @Command(name = "stop", description = "Stop workers")
    int stop() {
        try {
            WorkerService workerService = new WorkerService();
            workerService.stopWorkers();
            System.out.println("Workers stopped");
            return 0;
        } catch (Exception e) {
            System.err.println("Error: " + e.getMessage());
            return 1;
        }
    }

    @Override
    public Integer call() {
        CommandLine.usage(this, System.out);
        return 0;
    }
}

@Command(name = "status", description = "Show system status")
class StatusCommand implements Callable<Integer> {

    @Override
    public Integer call() {
        try {
            JobService jobService = new JobService();
            Map<Job.JobState, Long> counts = jobService.getStateCounts();
```

```
            System.out.println("=== QueueCTL Status ===");
            System.out.println("Pending:    " + counts.getOrDefault(Job.JobState.PENDING,
            System.out.println("Processing: " + counts.getOrDefault(Job.JobState.PROCESSI
            System.out.println("Completed:  " + counts.getOrDefault(Job.JobState.COMPLETE
            System.out.println("Failed:     " + counts.getOrDefault(Job.JobState.FAILED,
            System.out.println("Dead (DLQ): " + counts.getOrDefault(Job.JobState.DEAD, 0L

            WorkerService workerService = new WorkerService();
            System.out.println("\nActive Workers: " + workerService.getActiveWorkerCount(

            return 0;
        } catch (Exception e) {
            System.err.println("Error: " + e.getMessage());
            return 1;
        }
    }
}

@Command(name = "list", description = "List jobs by state")
class ListCommand implements Callable<Integer> {

    @Option(names = {"--state", "-s"}, required = true,
            description = "Job state: ${COMPLETION-CANDIDATES}")
    private Job.JobState state;

    @Override
    public Integer call() {
        try {
            JobService jobService = new JobService();
            List<Job> jobs = jobService.getJobsByState(state);

            System.out.printf("=== %s Jobs (%d) ===\n", state, jobs.size());
            for (Job job : jobs) {
                System.out.printf("ID: %s | Command: %s | Attempts: %d/%d\n",
                    job.getId(), job.getCommand(),
                    job.getAttempts(), job.getMaxRetries());
            }

            return 0;
        } catch (Exception e) {
            System.err.println("Error: " + e.getMessage());
            return 1;
        }
    }
}
```

Database Schema

```
-- Jobs table
CREATE TABLE jobs (
    id TEXT PRIMARY KEY,
    command TEXT NOT NULL,
    state TEXT NOT NULL CHECK(state IN ('PENDING', 'PROCESSING', 'COMPLETED', 'FAILED', '
    attempts INTEGER DEFAULT 0,
```

```
    max_retries INTEGER DEFAULT 3,
    created_at TEXT NOT NULL,
    updated_at TEXT NOT NULL,
    next_retry_at TEXT,
    error_message TEXT,
    output TEXT
);

CREATE INDEX idx_jobs_state ON jobs(state);
CREATE INDEX idx_jobs_next_retry ON jobs(next_retry_at) WHERE state = 'PENDING';
CREATE INDEX idx_jobs_created ON jobs(created_at);

-- Configuration table
CREATE TABLE config (
    key TEXT PRIMARY KEY,
    value TEXT NOT NULL,
    updated_at TEXT NOT NULL
);

-- Insert default configuration
INSERT INTO config (key, value, updated_at) VALUES
    ('max_retries', '3', datetime('now')),
    ('backoff_base', '2', datetime('now')),
    ('worker_poll_interval_ms', '1000', datetime('now')),
    ('graceful_shutdown_timeout_sec', '30', datetime('now'));

-- Workers table (for tracking active workers)
CREATE TABLE workers (
    id TEXT PRIMARY KEY,
    started_at TEXT NOT NULL,
    last_heartbeat TEXT NOT NULL,
    status TEXT NOT NULL CHECK(status IN ('ACTIVE', 'STOPPED'))
);

-- DLQ view (alternative: separate table)
CREATE VIEW dlq_jobs AS
SELECT * FROM jobs WHERE state = 'DEAD';
```

Project Structure

```
queuectl/
├── src/
│   ├── main/
│   │   ├── java/
│   │   │   └── com/
│   │   │       └── queuectl/
│   │   │           ├── QueueCtlApp.java          # Main entry point
│   │   │           ├── cli/                       # CLI commands
│   │   │           │   ├── QueueCtlCommand.java
│   │   │           │   ├── EnqueueCommand.java
│   │   │           │   ├── WorkerCommand.java
│   │   │           │   ├── StatusCommand.java
│   │   │           │   ├── ListCommand.java
│   │   │           │   ├── DLQCommand.java
│   │   │           │   └── ConfigCommand.java
```

```
│   │   │           ├── service/                  # Service layer
│   │   │           │   ├── JobService.java
│   │   │           │   ├── WorkerService.java
│   │   │           │   ├── DLQService.java
│   │   │           │   └── ConfigService.java
│   │   │           ├── core/                     # Core business logic
│   │   │           │   ├── JobManager.java
│   │   │           │   ├── WorkerManager.java
│   │   │           │   ├── RetryManager.java
│   │   │           │   └── LockManager.java
│   │   │           ├── worker/                   # Worker implementation
│   │   │           │   └── WorkerThread.java
│   │   │           ├── model/                    # Domain models
│   │   │           │   └── Job.java
│   │   │           ├── repository/               # Data access
│   │   │           │   ├── JobRepository.java
│   │   │           │   ├── ConfigRepository.java
│   │   │           │   └── WorkerRepository.java
│   │   │           ├── persistence/              # Database connection
│   │   │           │   └── DatabaseManager.java
│   │   │           └── util/                     # Utilities
│   │   │               ├── JsonUtil.java
│   │   │               └── CommandExecutor.java
│   │   └── resources/
│   │       ├── logback.xml                       # Logging config
│   │       └── application.properties            # App config
│   └── test/
│       └── java/
│           └── com/
│               └── queuectl/
│                   ├── core/
│                   │   ├── RetryManagerTest.java
│                   │   └── JobManagerTest.java
│                   ├── worker/
│                   │   └── WorkerThreadTest.java
│                   └── integration/
│                       └── QueueCtlIntegrationTest.java
├── pom.xml                                       # Maven config
├── README.md                                     # Documentation
├── .gitignore
└── scripts/
    ├── build.sh                                  # Build script
    ├── run.sh                                    # Run script
    └── test-scenarios.sh                         # Testing script
```

Maven Dependencies (pom.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.queuectl</groupId>
```

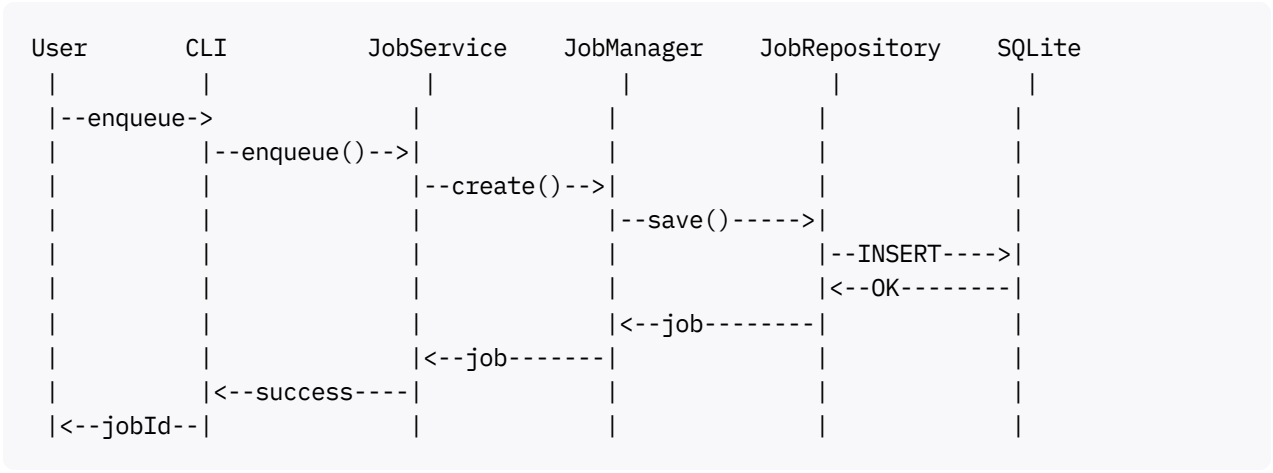
```
    <artifactId>queuectl</artifactId>
    <version>1.0.0</version>
    <packaging>jar</packaging>

    <properties>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <picocli.version>4.7.5</picocli.version>
        <sqlite.version>3.44.1.0</sqlite.version>
        <slf4j.version>2.0.9</slf4j.version>
        <logback.version>1.4.14</logback.version>
        <gson.version>2.10.1</gson.version>
        <junit.version>5.10.1</junit.version>
    </properties>

    <dependencies>
        <!-- Picocli for CLI -->
        <dependency>
            <groupId>info.picocli</groupId>
            <artifactId>picocli</artifactId>
            <version>${picocli.version}</version>
        </dependency>

        <!-- SQLite JDBC -->
        <dependency>
            <groupId>org.xerial</groupId>
            <artifactId>sqlite-jdbc</artifactId>
            <version>${sqlite.version}</version>
        </dependency>

        <!-- Logging -->
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>${slf4j.version}</version>
        </dependency>
        <dependency>
            <groupId>ch.qos.logback</groupId>
            <artifactId>logback-classic</artifactId>
            <version>${logback.version}</version>
        </dependency>

        <!-- JSON processing -->
        <dependency>
            <groupId>com.google.code.gson</groupId>
            <artifactId>gson</artifactId>
            <version>${gson.version}</version>
        </dependency>

        <!-- Testing -->
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter</artifactId>
            <version>${junit.version}</version>
            <scope>test</scope>
```

```
        </dependency>
        <dependency>
            <groupId>org.mockito</groupId>
            <artifactId>mockito-core</artifactId>
            <version>5.8.0</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <!-- Maven Compiler Plugin -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.11.0</version>
                <configuration>
                    <annotationProcessorPaths>
                        <path>
                            <groupId>info.picocli</groupId>
                            <artifactId>picocli-codegen</artifactId>
                            <version>${picocli.version}</version>
                        </path>
                    </annotationProcessorPaths>
                </configuration>
            </plugin>

            <!-- Maven Assembly Plugin for uber JAR -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-assembly-plugin</artifactId>
                <version>3.6.0</version>
                <configuration>
                    <descriptorRefs>
                        <descriptorRef>jar-with-dependencies</descriptorRef>
                    </descriptorRefs>
                    <archive>
                        <manifest>
                            <mainClass>com.queuectl.QueueCtlApp</mainClass>
                        </manifest>
                    </archive>
                </configuration>
                <executions>
                    <execution>
                        <id>make-assembly</id>
                        <phase>package</phase>
                        <goals>
                            <goal>single</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>

            <!-- Maven Surefire Plugin for tests -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
```

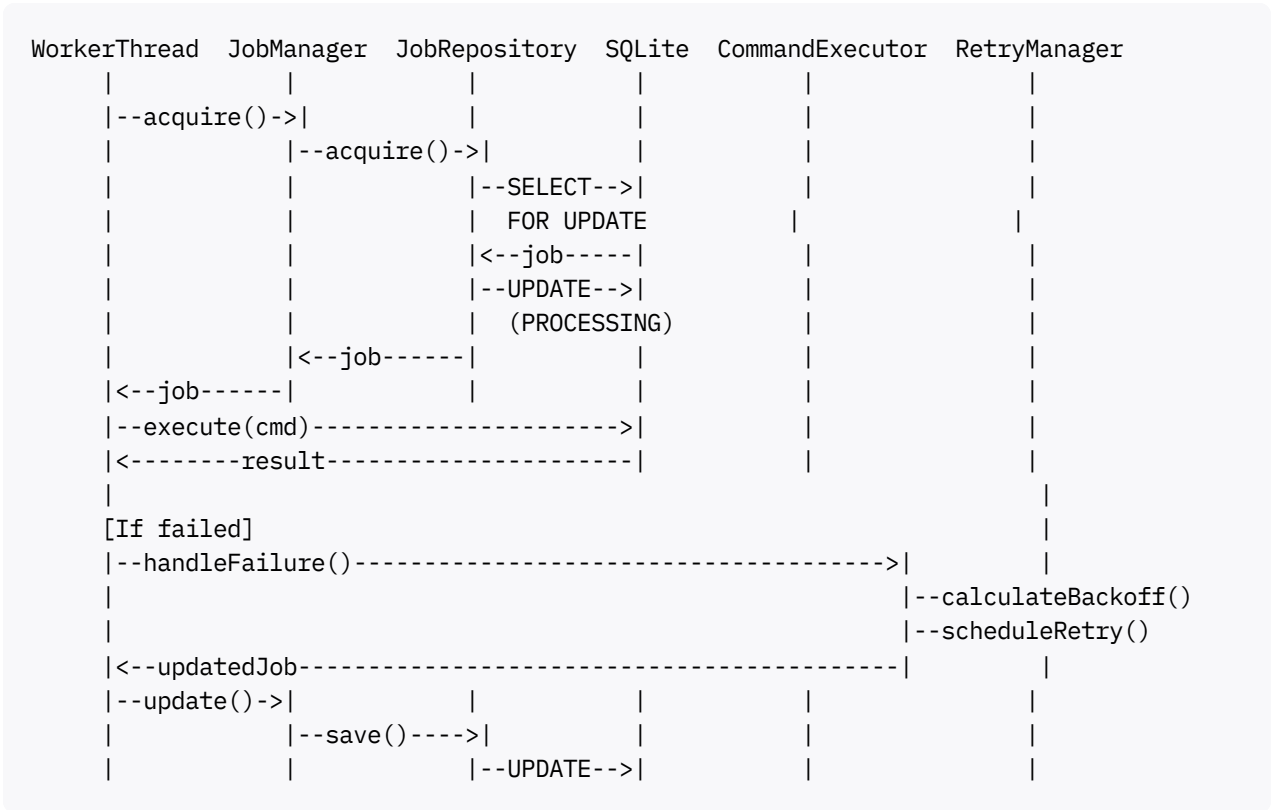
```
                <artifactId>maven-surefire-plugin</artifactId>
                <version>3.2.3</version>
            </plugin>
        </plugins>
    </build>
</project>
```

Sequence Diagrams

Job Enqueue Sequence:



Job Processing Sequence:



Key Algorithms

1. Exponential Backoff Calculation:

```
public long calculateBackoffSeconds(int attempts) {
    // Formula: delay = base^attempts
    long baseDelay = (long) Math.pow(backoffBase, attempts);

    // Apply jitter (optional)
    if (useJitter) {
        long jitter = (long) (Math.random() * baseDelay * 0.25);
        baseDelay += jitter;
    }

    // Apply maximum delay cap (optional)
    return Math.min(baseDelay, maxDelaySeconds);
}
```

2. Job Acquisition with Locking:

```
public Optional<Job> acquireNextPendingJob() throws SQLException {
    connection.setAutoCommit(false);
    try {
        // SELECT FOR UPDATE provides pessimistic locking
        String sql = """
            SELECT * FROM jobs
            WHERE state = 'PENDING'
            AND (next_retry_at IS NULL OR next_retry_at <= ?)
            ORDER BY created_at
            LIMIT 1
            FOR UPDATE
        """;

        Job job = // ... execute query and fetch job

        if (job != null) {
            job.setState(JobState.PROCESSING);
            save(job);
            connection.commit();
            return Optional.of(job);
        }

        connection.commit();
        return Optional.empty();
    } catch (SQLException e) {
        connection.rollback();
        throw e;
    }
}
```

3. Graceful Shutdown:

```
private void gracefulShutdown() {
    // 1. Signal all workers to stop accepting new jobs
    for (WorkerThread worker : workers) {
        worker.shutdown();
    }

    // 2. Shutdown executor (no new tasks accepted)
    executorService.shutdown();

    try {
        // 3. Wait for current jobs to complete
        if (!executorService.awaitTermination(30, TimeUnit.SECONDS)) {
            // 4. Force shutdown if timeout
            executorService.shutdownNow();
        }
    } catch (InterruptedException e) {
        executorService.shutdownNow();
        Thread.currentThread().interrupt();
    }
}
```

Testing Strategy

Unit Tests:

- RetryManagerTest: Test backoff calculations
- JobManagerTest: Test job state transitions
- WorkerThreadTest: Test job processing logic
- CommandExecutorTest: Test command execution

Integration Tests:

- End-to-end job flow: enqueue → process → complete
- Retry flow: enqueue → fail → retry → complete
- DLQ flow: enqueue → fail (max retries) → DLQ
- Worker management: start → process jobs → stop
- Concurrency: multiple workers processing jobs

Test Scenarios Script:

```
#!/bin/bash
# test-scenarios.sh

# Scenario 1: Basic job completes successfully
queuectl enqueue '{"id":"job1","command":"echo hello"}'
queuectl worker start --count 1 &
sleep 5
queuectl status
queuectl list --state completed
killall queuectl
```

```
# Scenario 2: Failed job retries
queuectl enqueue '{"id":"job2","command":"exit 1","max_retries":3}'
queuectl worker start --count 1 &
sleep 15  # Wait for retries
queuectl status
queuectl dlq list

# Scenario 3: Multiple workers
queuectl enqueue '{"id":"job3","command":"sleep 2"}'
queuectl enqueue '{"id":"job4","command":"sleep 2"}'
queuectl worker start --count 3 &
queuectl status
sleep 5
queuectl status

# Scenario 4: Persistence across restarts
queuectl enqueue '{"id":"job5","command":"echo persist"}'
# Kill and restart
queuectl worker start --count 1 &
queuectl status
```

Performance Considerations

Concurrency: [16] [26]

- Use connection pooling for database access
- Worker threads use ThreadPoolExecutor for efficiency
- Implement job-level pessimistic locking to prevent race conditions[7]

Scalability:

- Index on job state and next_retry_at for fast queries
- Partition jobs by state in database for better performance
- Consider Redis for high-throughput scenarios

Reliability:

- WAL mode for SQLite for better concurrency[10]
- Transaction management for atomic operations
- Worker heartbeat for failure detection

Extension Points

Bonus Features Implementation:

1. **Job Timeout Handling:**

```
Future<?> future = executor.submit(() -> executeCommand(job));
try {
    future.get(job.getTimeoutSeconds(), TimeUnit.SECONDS);
```

```
} catch (TimeoutException e) {
    future.cancel(true);
    retryManager.handleFailure(job, "Job timeout");
}
```

2. **Priority Queues:**

```
// Add priority field to Job
ALTER TABLE jobs ADD COLUMN priority INTEGER DEFAULT 0;

// Modify acquisition query
SELECT * FROM jobs
WHERE state = 'PENDING'
ORDER BY priority DESC, created_at
LIMIT 1 FOR UPDATE;
```

3. **Scheduled Jobs:**

```
// Add run_at field
private Instant runAt;

// Worker checks run_at before processing
if (job.getRunAt() != null &&
    Instant.now().isBefore(job.getRunAt())) {
    continue; // Skip this job
}
```

Summary

This design provides a robust, production-ready job queue system with:

- **Modular architecture** with clear separation of concerns[11]
- **Reliable persistence** using SQLite[9] [10]
- **Concurrent processing** with multiple worker threads[15] [16]
- **Exponential backoff retry** mechanism[21] [6] [5]
- **Dead Letter Queue** for failed jobs[24] [23]
- **Graceful shutdown** handling[19] [20] [18]
- **CLI interface** using Picocli[3] [4]
- **Pessimistic locking** to prevent duplicate processing[8] [7]
- **Comprehensive testing** strategy

The system is designed to handle production workloads while remaining simple to understand, extend, and maintain.

⁂

1. https://www.geeksforgeeks.org/system-design/task-queues-system-design/

2. https://dev.to/sgchris/designing-a-job-queue-system-sidekiq-and-background-processing-2oln
3. https://picocli.info
4. https://www.baeldung.com/java-picocli-create-command-line-program
5. https://betterstack.com/community/guides/monitoring/exponential-backoff/
6. https://dzone.com/articles/understanding-retry-pattern-with-exponential-back
7. https://www.geeksforgeeks.org/java/file-locking-in-java-to-prevent-concurrent-access/
8. https://stackoverflow.com/questions/11543967/how-to-lock-a-file
9. https://www.youtube.com/watch?v=871Jj_7gYx4
10. https://developer.android.com/training/data-storage/sqlite
11. https://www.digitalocean.com/community/tutorials/java-design-patterns-example-tutorial
12. https://refactoring.guru/design-patterns/command/java/example
13. https://refactoring.guru/design-patterns/command
14. https://www.geeksforgeeks.org/system-design/command-pattern/
15. https://java-design-patterns.com/patterns/master-worker/
16. https://scand.com/company/blog/java-concurrency-multithreading-practical-guide/
17. https://docs.oracle.com/javase/8/docs/api/java/nio/channels/FileLock.html
18. https://gist.github.com/007ebb70cd46e8f1031aac5f181b9d30
19. https://www.javacodegeeks.com/2013/10/tomcats-graceful-shutdown-with-daemons-and-shutdown-hooks.html
20. https://dzone.com/articles/tomcats-graceful-shutdown
21. https://www.hackerone.com/blog/retrying-and-exponential-backoff-smart-strategies-robust-software
22. https://en.wikipedia.org/wiki/Exponential_backoff
23. https://www.geeksforgeeks.org/system-design/dead-letter-queue-system-design/
24. https://andrew-jones.com/blog/the-dead-letter-queue-pattern/
25. https://ibm-cloud-architecture.github.io/refarch-kc/implementation/dead-letter-queue/
26. https://wpwebinfotech.com/blog/java-concurrency/
27. https://slack.engineering/scaling-slacks-job-queue/
28. https://code-along.hashnode.dev/why-job-queues-in-your-system-architecture-cm2836rt7000008k3135a46z5
29. https://softwareontheroad.com/job-queues-cqrs-nodejs-mongodb-agenda
30. https://www.designgurus.io/answers/detail/how-to-design-a-message-queue-for-system-design-interviews
31. https://blog.shalvah.me/posts/building-a-task-queue-part-1
32. https://learn.microsoft.com/en-us/azure/architecture/patterns/queue-based-load-leveling
33. https://www.devitpl.com/application-development/beginners-guide-to-retry-pattern-implementation-with-exponential-backoff/
34. https://newsletter.systemdesignclassroom.com/p/work-queues-the-simplest-form-of
35. https://www.reddit.com/r/commandline/comments/m62cjq/recommended_architecture_for_cli_applications/
36. https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/retry-backoff.html

37. https://www.youtube.com/watch?v=mkT0TZ0RSiw
38. https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-dead-letter-queues
39. https://www.elastic.co/docs/reference/logstash/dead-letter-queues
40. https://stackoverflow.com/questions/6524842/sqlite-persistent-storage-of-object-instances
41. https://www.confluent.io/learn/kafka-dead-letter-queue/
42. https://stackoverflow.com/questions/7903503/java-performance-processes-vs-threads
43. https://codingwithmitch.com/blog/sqlite-and-the-room-persistence-library/
44. https://www.enterpriseintegrationpatterns.com/patterns/messaging/DeadLetterChannel.html
45. https://docs.oracle.com/javase/tutorial/uiswing/concurrency/worker.html
46. https://www.freecodecamp.org/news/room-sqlite-beginner-tutorial-2e725e47bfab/
47. https://aws.amazon.com/what-is/dead-letter-queue/
48. https://github.com/remkop/picocli
49. https://www.infoq.com/articles/java-native-cli-graalvm-picocli/
50. https://quarkus.io/guides/picocli
51. https://picocli.info/quick-guide.html
52. https://mail.openjdk.org/pipermail/loom-dev/2022-December/005119.html
53. https://www.baeldung.com/java-lock-files
54. https://www.youtube.com/watch?v=PaxBXABJIzY
55. https://stackoverflow.com/questions/3194545/how-to-stop-a-java-thread-gracefully
56. https://www.baeldung.com/java-concurrent-locks
57. https://devm.io/java/java-picocli-framework