DESIGN DOCUMENT

Event-Driven Job Queue System

Problem Statement

Background job processing is deceptively hard.

Jobs must continue executing correctly even when processes crash, servers restart, or shutdowns occur mid-execution. In these conditions, losing jobs is unacceptable — but preventing duplicate execution entirely is often impractical.

The core problem this system addresses is:

How do we reliably execute background jobs under crashes and shutdowns, without losing work, while keeping the system simple, debuggable, and correct?

This system explicitly targets **at-least-once execution semantics**.
Exactly-once execution is intentionally avoided, as it requires distributed transactions, pervasive idempotency, or external coordination mechanisms that significantly increase complexity with limited real-world benefit. In practice, bounded retries and recovery provide a more robust and understandable solution.

Goals

This system is designed to:

- Persist job state so work is not lost across crashes or restarts
- Recover jobs that are interrupted during execution
- Retry failures in a controlled, observable manner
- Shut down predictably without corrupting state or abandoning work
- Keep concurrency and failure handling explicit and easy to reason about

The design prioritizes **clarity, correctness, and failure-mode reasoning** over horizontal scalability or feature completeness.

Non-Goals

This system intentionally does **not** attempt to solve:

- Exactly-once execution semantics

- Distributed scheduling across multiple nodes
- High-throughput message streaming
- Horizontal database scalability
- Real-time job guarantees

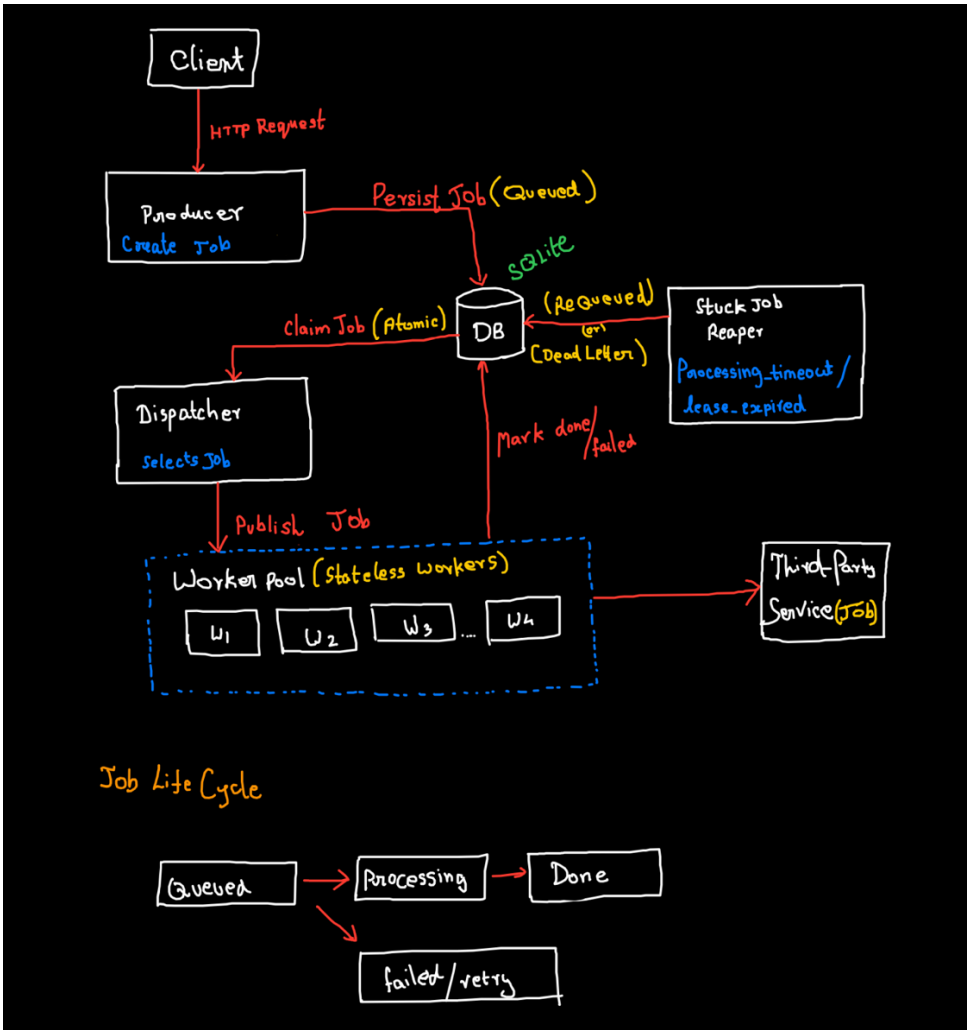
These trade-offs are deliberate to keep the system focused and correct.

High-Level Architecture

The system consists of five main components:

1. **HTTP API** – accepts job creation requests
2. **Dispatcher** – selects runnable jobs from persistent storage
3. **Worker Pool** – executes jobs concurrently
4. **Visibility Reaper** – recovers stuck jobs
5. **Persistent Store (SQLite)** – source of truth for job state

See Architecture Diagram for component interactions



Data Flow

```
HTTP Request
     ↓
Persist Job (queued)
     ↓
Dispatcher selects job
     ↓
Worker executes job
     ↓
Job marked done / retried / failed
```

Job Data Model

Each job is represented as a persistent row with explicit state.

Core Fields

- `id` – unique job identifier
- `type` – job type (email, webhook, etc.)
- `status` – queued | processing | done | failed
- `payload` – job-specific data
- `attempts` – number of execution attempts
- `max_retries` – retry limit
- `run_at` – scheduled execution time
- `Idempotency_key` - The idempotency key is used only to deduplicate job creation requests, not to guarantee idempotent execution.
- `started_at` – timestamp when processing began
- `finished_at` – timestamp when job completed
- `created_at` / `updated_at` – lifecycle tracking

The database is treated as the **single source of truth**.

Job Lifecycle & State Machine

States

- **queued** – ready to be picked up
- **processing** – currently being executed by a worker
- **done** – completed successfully
- **failed** – exceeded retry limit

Transitions

```
queued → processing → done
queued → processing → failed
```

`processing → queued (visibility timeout)`

A job may legally move from `processing` back to `queued`.
This enables recovery when a worker crashes after claiming a job but before persisting completion.

Execution Semantics

At-Least-Once Execution

- Jobs may run more than once
- Duplicate execution is **bounded by retry limits**
- Job loss is unacceptable

This system explicitly prefers bounded duplication over loss.

Why Not Exactly-Once?

Exactly-once execution cannot be guaranteed under crashes without heavy coordination or external systems. Attempting it adds complexity without providing real correctness guarantees.

Guarantees

This system guarantees:

- At-least-once execution
- No job loss after persistence
- Eventual recovery of stuck jobs
- Bounded concurrency
- Safe shutdown without partial writes

This system does NOT guarantee:

- Exactly-once execution
- Ordering across job types
- Real-time execution
- Distributed fault tolerance

Dispatcher (Single Dispatcher)

The dispatcher is a coordination component that selects runnable jobs from the database and assigns them to workers.

The dispatcher continuously queries for runnable jobs:

- `status = queued`
- `run_at <= now`

Claiming a job is performed atomically using a conditional update to ensure that **only one component** can transition a job from `queued` to `processing`.

Once claimed:

- Status transitions to `processing`
- `started_at` is recorded
- Attempt count is incremented

Why is a Dispatcher needed?

If each worker independently polls the database for runnable jobs, the system experiences:

- High write lock contention
- Many failed transactions that do no useful work
- Increased tail latency under load

The dispatcher exists to **centralize job claiming**, ensuring that only one component competes for database locks at a time. This dramatically reduces lock contention and stabilizes throughput.

Importantly, the dispatcher **does not provide correctness**.
Correctness is guaranteed solely by **atomic database state transitions inside transactions**.
The dispatcher is an **optimization and coordination mechanism**, not a correctness mechanism.

What happens without the Dispatcher?

Without a dispatcher, every worker continuously queries the database for runnable jobs.

For example, if there is **1 runnable job and 10 workers**, all 10 workers attempt to claim it concurrently:

- 1 succeeds
- 9 fail after acquiring and releasing database locks

This wastes CPU cycles, increases database load, and distributes concurrency control across all workers, making system behavior harder to reason about.

A dispatcher avoids this by **serializing job selection**, while still allowing workers to execute jobs concurrently.

Isn’t the dispatcher a bottleneck?

The dispatcher **trades peak throughput for efficiency and stability**.

It bounds job-claiming throughput to reduce lock contention and wasted work. Execution remains parallel, and in SQLite this improves effective throughput and tail latency.

Worker Pool Design

Concurrency Model

- Fixed-size worker pool to bound concurrent job execution
- Workers are long-lived goroutines to avoid churn
- Jobs are delivered via a buffered channel to decouple job admission from execution

Worker Invariants

- A worker processes **exactly one job at a time**
- A worker **never exits due to job failure**
- Job success or failure **does not affect worker lifecycle**
- Workers exit **only when the job channel is closed during shutdown**

Workers are **stateless and disposable** with respect to job outcomes.

Responsibility Boundary

Workers do **not** enforce correctness or retries.
They execute jobs and report outcomes.

All correctness guarantees (retries, at-least-once delivery, recovery) are enforced by **persistent database state**, not in-memory worker logic.

This design ensures stable throughput and predictable behavior even under repeated job failures.

Visibility Timeout & Recovery (Stuck Job Reaper)

Problem

If a worker crashes or the process terminates while processing a job, the job may remain indefinitely in the `processing` state and never be retried.

Solution: Visibility Reaper

A periodic background process scans for jobs that:

- Are in `processing`
- Have a `started_at` timestamp older than a configured visibility timeout

For such jobs, an atomic update is performed:

- If retry attempts remain, the job is requeued
- If retries are exhausted, the job is marked as failed (dead-lettered)

Guarantees

This mechanism ensures **eventual recovery assuming the system continues to make progress**.

Stuck jobs are retried up to a **bounded limit and then permanently failed.**

It provides **at-least-once execution semantics**, allowing jobs to be retried after crashes while avoiding permanent loss.

The reaper does not guarantee exactly-once execution; **bounded duplicates** are possible and explicitly accepted.

Graceful Shutdown

Shutdown Requirements

- Stop accepting new HTTP requests
- Allow in-flight HTTP requests to complete
- Prevent new jobs from entering the system
- Allow workers to finish in-flight jobs
- Exit without data corruption or lost state

Shutdown Sequence

1. OS signal received (`SIGINT` / `SIGTERM`)
2. HTTP server stops accepting new requests
3. Dispatcher stops polling and producing new jobs
4. Job channel is closed
5. Workers drain and complete in-flight jobs
6. Process exits cleanly

A hard timeout can be layered on top of graceful shutdown to force exit if jobs exceed a maximum shutdown duration.(Need to implement)

Concurrency & Synchronization

- Channels are used for work distribution
- `sync.WaitGroup` tracks worker lifetimes
- Only producers close channels
- Workers never close shared channels

These rules prevent deadlocks and panics.

Failure Scenarios

Handled explicitly:

- Worker crash mid-job
- Process termination
- SQLite lock contention
- Partial shutdown

Not handled (by design):

- Machine-level disk corruption
- Byzantine failures

Trade-offs & Design Rationale

Why SQLite?

SQLite was chosen to support a **single-node, crash-resilient job queue** with strong transactional guarantees and minimal operational complexity.

Running in WAL mode allows concurrent readers with serialized writes, making transactional job claiming reliable while keeping the system easy to operate.

The primary tradeoff is **limited write concurrency and lack of horizontal scalability**, which directly motivates the use of a centralized dispatcher to reduce lock contention.

Why In-Memory Channels?

In-memory channels are used to deliver jobs from the dispatcher to workers to provide clear ownership, natural backpressure, and deterministic shutdown semantics.

Channels act as an execution pipeline, while the database remains the source of truth for job state and recovery.

The tradeoff is that jobs in transit are not durable; on process crashes, recovery relies on persisted job state and the visibility reaper.

Why Bounded Retries?

Retries are bounded to prevent infinite retry loops and to ensure that repeated failures become explicit and observable.

After retries are exhausted, jobs are marked as failed and moved to a dead-letter state for inspection.

The tradeoff is accepting permanent failure for some jobs in exchange for overall system stability and debuggability.

Future Improvements (for Version 2)

The system is intentionally scoped as a **single-node, correctness-first job queue**.
The following enhancements are deferred to preserve clarity and avoid premature complexity.

PostgreSQL Backend

Migrating to PostgreSQL would enable row-level locking and higher write concurrency, potentially eliminating the need for a centralized dispatcher.
This is deferred because it fundamentally changes the contention model and shifts the focus from correctness to scalability.

Idempotency Helpers

Idempotency keys and deduplication helpers would reduce the impact of at-least-once execution semantics.
This is deferred to keep the core system focused on **job lifecycle correctness**, leaving side-effect safety to application-level logic.

Metrics & Observability

Metrics such as queue depth, processing latency, and retry counts would improve operability.
This is deferred to avoid coupling the core execution model to a specific observability stack.

Priority Queues

Job prioritization would allow higher-importance jobs to bypass FIFO ordering.
This is deferred to keep scheduling semantics simple and avoid starvation edge cases.

Summary

This system demonstrates how a **single-node, persistent job queue** can be designed to behave correctly under real-world failure conditions.

It explicitly addresses:

- Crash recovery via persistent state and visibility timeouts
- Bounded concurrency through a fixed worker pool
- Predictable shutdown using admission control and drain semantics
- Clear separation between coordination, execution, and correctness

The system provides **at-least-once execution semantics** and accepts bounded duplicate execution in exchange for simpler recovery and stronger correctness guarantees.

The design intentionally favors **clarity, explicit trade-offs, and failure-mode reasoning** over horizontal scalability or exactly-once execution.