Chronos – Distributed Job Scheduler System

---

1. One-Pager (High-Level Summary)

1.1 Overview

In modern computing environments, being able to efficiently schedule, manage, and monitor tasks, also known as jobs, is crucial. These could range from simple tasks like sending a weekly email to more complex operations like processing data or maintaining databases. This project will have you design and implement a robust and scalable backend for a job scheduling system. Develop a reliable and scalable distributed job scheduling system that can execute, manage, and monitor a variety of tasks. The system should support one-time jobs as well as recurring jobs, providing comprehensive job management functionality. While the system will primarily handle backend functionalities, there should be provision for interaction with a potential frontend through well-defined APIs.In the intricate landscape of modern computing, the efficient scheduling, meticulous management, and vigilant monitoring of tasks—commonly referred to as jobs—are not merely beneficial but absolutely paramount. These jobs can span a vast spectrum of complexity, from the straightforward execution of a weekly email dispatch to the intricate orchestration of data processing pipelines or the ongoing maintenance of robust database systems.

This project embarks on the ambitious endeavor of designing and implementing a resilient and scalable backend infrastructure for a sophisticated job scheduling system. The core objective is to architect a reliable and scalable distributed job scheduling system capable of executing, managing, and meticulously monitoring an eclectic array of tasks. The system must inherently support both ephemeral, one-time jobs and recurring, scheduled tasks, providing a comprehensive suite of functionalities for end-to-end job management. While the primary focus of this system will be on the robust implementation of backend functionalities, it is imperative that provisions are made for seamless interaction with a potential frontend. This will be achieved through the meticulous definition and exposition of well-structured Application Programming Interfaces (APIs), ensuring a clear and efficient communication channel between the backend and any future user interface.

1.2 Key Features

1. Job Submission

Users can submit jobs through REST APIs, specifying execution details such as target HTTP endpoint, payload, headers, and execution time. Jobs can be executed immediately or scheduled for a future timestamp.

2. Recurring & Cron-Based Scheduling

Chronos supports both simple recurring intervals (e.g., every X minutes/hours) and full cron expressions. The system automatically computes and updates the `next_execution_time` atomically for recurring jobs, ensuring correct and conflict-free scheduling across multiple application servers.

3. Distributed Job Execution

Multiple application servers can run Chronos simultaneously. Each server independently polls for due jobs and uses PostgreSQL's `SELECT … FOR UPDATE SKIP LOCKED` to safely lock and execute jobs without duplication, enabling true multi-node scalability and horizontal load distribution.

4. Robust Job Management

Users can query job status, view execution history, reschedule jobs, modify recurring patterns, or cancel upcoming executions through the exposed APIs. All job state transitions (PENDING → RUNNING → SUCCESS/FAILED/RETRY) are tracked consistently.

5. Fault Tolerance & Automatic Retries

Chronos includes a retry mechanism with exponential backoff for transient failures such as network timeouts or 5xx responses. If retries are exhausted or a job repeatedly fails, the system marks it as FAILED and (optionally) notifies the user.

6. Execution Heartbeats & Crash Recovery

While a job is executing, the worker thread updates the job's heartbeat timestamp. If the job exceeds the allowed maximum execution time or heartbeat becomes stale (e.g., server crash), Chronos marks the job as STUCK. Other servers can safely detect and re-execute these jobs, ensuring system resilience.

7. Detailed Logging & Monitoring

Every job invocation logs:

    - start time
    - end time
    - response code
    - response time
    - error details
    - execution node

1.3 Technical Highlights

1. **PostgreSQL-Based Distributed Locking** using `SELECT … FOR UPDATE SKIP LOCKED` for safe multi-node scheduling
2. **Multi-Node Job Execution** with automatic conflict prevention and horizontal scaling
3. **Atomic Recurring Job Scheduling** by updating `next_execution_time` inside the same DB transaction
4. **Fault-Tolerant Execution** via heartbeat tracking and max-execution watchdog
5. **Automatic Retry Mechanism** with exponential backoff
6. **Worker Crash Recovery** through stale-heartbeat detection and job reassignment
7. **Spring Boot Microservice Architecture** for clean separation of scheduler, executor, and APIs
8. **ExecutorService-Based Worker Pool** for concurrent job execution within each node
9. **Structured Logging & Execution History** stored in PostgreSQL for observability
10. **Future-Ready Next.js Dashboard** (with shadcn/ui) for job monitoring and lifecycle management
11. **Horizontal Scalability** without introducing Kafka/Redis in v1 (DB-driven coordination)

1.4 Project Timeline

Milestone	Target Date	Status
Project Start	10 Nov 2025	Started
One Pager Submission	15 Nov 2025	In Review
Core Features Complete	21 Nov 2025	Not Started
Testing & Debugging	23 Nov 2025	Not Started

Milestone	Target Date	Status
Final Delivery	29 Nov 2025	Not Started

1.5 Contact Info

For more details or inquiries, please contact:

- **Name:** Muhammed Hilal
- **Email:** Muhammed Hilal

2. High-Level Design (HLD)

2.1 Architecture Overview

- API Service

  A Spring Cloud Gateway receives all incoming API requests and routes them to the appropriate backend services (Scheduler Service or Worker Service). This enables clean separation between job management APIs and background execution components.

- Load Balancer

  Multiple instances of each service (Scheduler and Worker) run behind a load balancer. This allows Chronos to scale horizontally:

  - more scheduler nodes = faster scanning of due jobs
  - more worker nodes = higher throughput for job execution

  All nodes interact with the shared PostgreSQL database, relying on distributed locking to prevent conflicts.

- Scheduler Service

  The Scheduler Service provides all external APIs for job management:

  - Create a job (one-time or recurring)
  - Modify job details
  - Cancel a job
  - View job history or status

  Internally, the Scheduler Service also contains:

  Due-Job Scanner

  A periodic task that:

  1. Queries PostgreSQL for jobs where `next_execution_time <= NOW()`
  2. Uses `SELECT … FOR UPDATE SKIP LOCKED` to atomically lock each due job
  3. Updates the job's `next_execution_time` for recurring schedules inside the same transaction
  4. Marks the job as `RUNNING` and assigns it to the Worker Service by leaving it ready for worker polling

  This ensures **strict single-execution semantics** even across multiple scheduler instances.

- Worker Executor (ExecutorService on each node)

  Each Worker Service instance:

  1. Polls the database for `RUNNING` jobs that are not assigned to a worker yet
  2. Claims them using row-level locking
  3. Submits them to its internal `ExecutorService` thread pool
  4. Executes the HTTP call specified in the job
  5. Updates the job with execution result
  6. Handles retries using exponential backoff for:
     - network timeouts
     - 5xx server errors
  7. Writes structured execution logs to the database

  Workers operate independently across nodes, making the execution layer **fully distributed**.

- Heartbeat Liveliness checker

  Each job being executed is associated with a "heartbeat" entry:

    - The worker thread updates `last_heartbeat_at` every few seconds.
    - If the worker dies or JVM crashes, the heartbeat stops.
    - This allows detecting stuck or abandoned jobs.
    - When the job finishes, its heartbeat stops and the status is updated.

  Heartbeats ensure **worker crash recovery** across multiple nodes.

- Max Execution Time Checker (Timeout Recovery)

  A separate scheduled task on every node monitors jobs in `RUNNING` state:

  1. If `NOW() - started_at > max_execution_time`, the job is considered **hung**.
  2. The system marks it as `STUCK` or `RETRY_PENDING`.
  3. Any Worker Service can later re-acquire and retry the job using the same `SKIP LOCKED` mechanism.

  This prevents jobs from hanging forever due to slow or unresponsive endpoints.

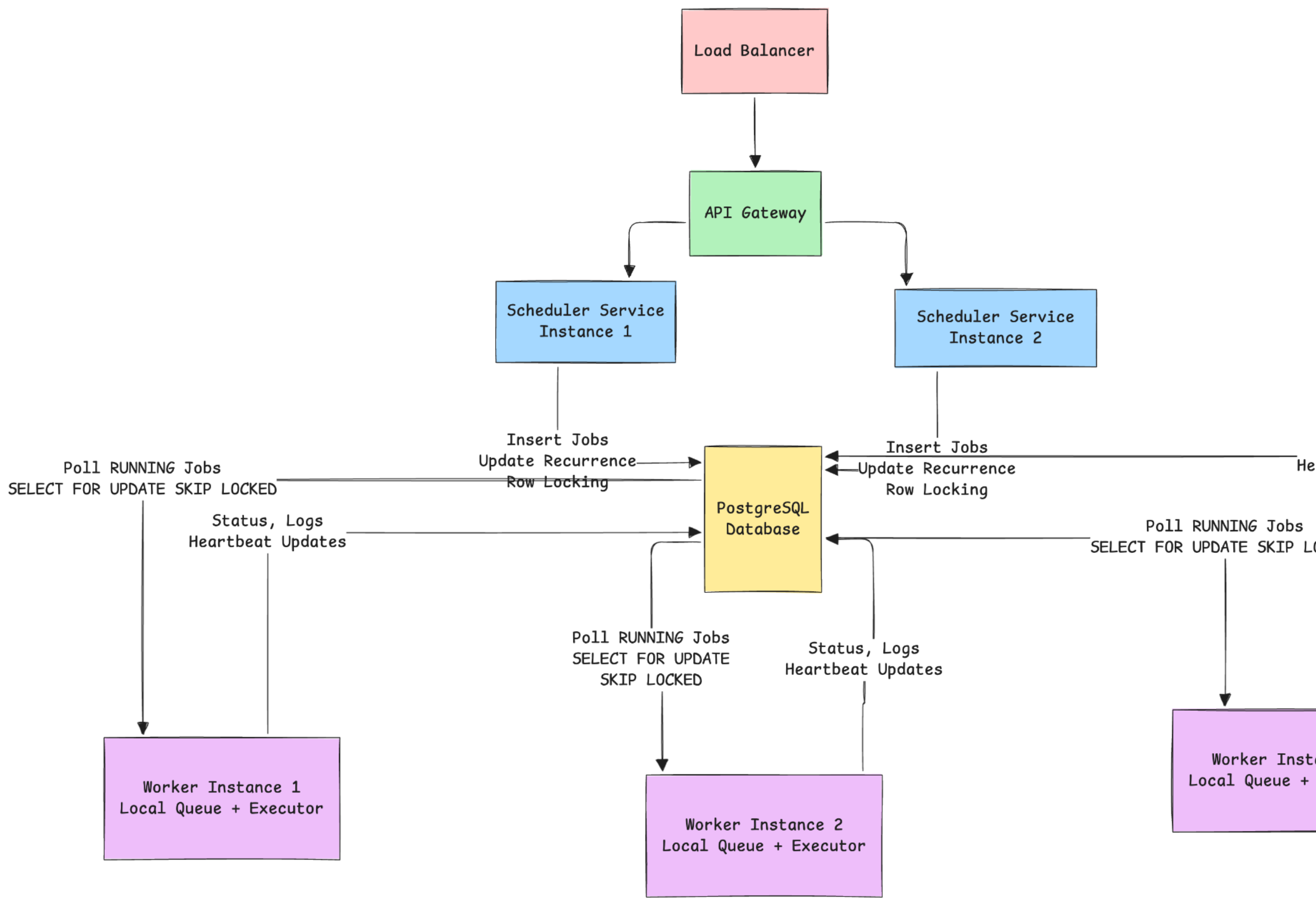
- PostgreSQL Database

  PostgreSQL acts as the central coordination mechanism for the entire distributed system. It stores:

    - job definitions
    - next execution timestamps
    - lock state
    - execution logs
    - heartbeat timestamps
    - retry counters
    - status transitions

  PostgreSQL's row-level locking ensures **safe concurrency** and **no duplicate execution** in a multi-node environment.

2.2 Architecture Diagram



2.3 Job Execution Flow

Explain:

- **Job creation → DB**
  - A user creates a job through the API, specifying:
    - whether it is one-time or recurring
    - execution time
    - payload and configuration
  - The API validates the request
  - The job is saved in the PostgreSQL database with:
    - `status = PENDING`
    - `next_execution_time` set to the user-defined schedule
    - `retry_count = 0`
    - `max_execution_time` (if configured)

  At this point the job is dormant, waiting for the scheduler.

- **Scheduler → picks due jobs**

  All application servers run the scheduler loop in parallel.

To safely pick jobs without conflict, each scheduler node executes:

```sql
SELECT *
FROM jobs
WHERE next_execution_time <= NOW()
  AND status = 'PENDING'
FOR UPDATE SKIP LOCKED
LIMIT X;
```

This ensures:

- Only one scheduler instance locks a job at a time
- Locked rows are skipped by other scheduler nodes
- Multiple schedulers can run concurrently without race conditions

Once a scheduler successfully locks a job, it:

- sets `status = RUNNING`
- computes and updates `next_execution_time` (for recurring jobs)
- sets `assigned_worker_id` (for tracing)
- sets `started_at = NOW()`

All of these updates occur atomically within the same transaction, ensuring strict single-execution guarantees across distributed nodes.

- **Worker executor runs job**

  Worker Service instances continuously poll the database for jobs that have:

  - `status = RUNNING`, and
  - have **not** yet been picked by a worker

  Workers claim these jobs using another `SELECT FOR UPDATE SKIP LOCKED`, ensuring exclusive ownership.

  Once claimed, the job is placed into the worker's internal `ExecutorService` thread pool.
  During execution:

  - the worker thread periodically updates `last_heartbeat_at`
  - a hang checker monitors for unresponsive or stalled execution
  - any exception is captured and logged

  On completion, the worker updates the job in the database with:

  - final `status` (SUCCESS / FAILED / RETRY_PENDING)
  - execution logs

        - timestamps

- **Updates job state**

  The final job state is always written back to the database, ensuring accurate tracking for monitoring, retries, and history queries.

- **Execution Completion Handler (Post-Execution Processing)**

  Every scheduler node runs a periodic task to process finished executions:

    - Scan the `job_executions` table for rows where `status IN ('SUCCESS','FAILED','TIMED_OUT')` and not yet processed.
    - For each finished execution:
        - If `SUCCESS`:
            - For recurring jobs → compute & update `next_execution_time`
            - Set job `status = PENDING`
        - If `FAILED` or `TIMED_OUT`:
            - If `retry_count < max_retry`:
                - increment retry_count
                - compute backoff delay
                - update next_execution_time
                - set job `status = PENDING`
            - Else:
                - set job `status = FAILED`
                - next_execution_time = NULL (For one time executions else schedule the next run)

    - Mark the execution entry as processed (or rely on timestamps to skip old ones)

  This ensures correct rescheduling and safe retries across nodes.

2.4 Recurring Job Flow

For recurring jobs, the scheduler computes the next occurrence **before** execution starts.
This is done inside the same transaction that sets the job to `RUNNING`, guaranteeing:

- no missed intervals
- no double scheduling
- deterministic recurrence behavior

The formula for calculating the next execution is based on the user's defined interval (e.g., hourly, daily, custom cron-like values).

2.5 Distributed Coordination

Chronos achieves distributed behavior through PostgreSQL's row-level locking and stateless service design.

Multiple application servers reading from the DB

Both Scheduler and Worker Services run as multiple instances behind a load balancer. All nodes access the same shared PostgreSQL database.

How locking prevents duplicates

`SELECT … FOR UPDATE SKIP LOCKED` ensures:

- only one scheduler can pick a pending job
- only one worker can claim a running job
- no two nodes can ever execute the same job simultaneously

How crash recovery works

Workers update a `last_heartbeat_at` field while executing jobs.
If a node crashes:

- heartbeat stops
- timeout checker identifies stalled jobs
- their status is reset to allow retry
- any healthy worker can safely re-acquire the job using SKIP LOCKED

This provides fault tolerance across distributed worker nodes.

2.6 Scalability Strategy

Horizontal Scaling of Application Servers

Both Scheduler and Worker services can scale independently by simply adding more instances.
More schedulers = faster detection of due jobs
More workers = higher execution throughput

PostgreSQL Locking Behavior

Row-level locks and SKIP LOCKED provide natural distributed coordination without external systems like Redis or Zookeeper.

Batching + Indexing

Indexes on `next_execution_time`, `status`, and `assigned_worker_id` ensure efficient scanning even with large job volumes.
Schedulers query jobs in small batches to reduce lock contention.

Fault Tolerance

Heartbeat-based liveliness checks and timeout monitors ensure:

- detection of stuck jobs
- safe re-execution after crashes
- no job left in indefinite-running state

Retry Mechanism with Exponential Backoff

Failed jobs transition to `RETRY_PENDING`.
The scheduler recalculates the next retry time using exponential backoff to reduce load during repeated failures.

---

3. Low-Level Design (LLD)

3.1 DB Schema

Tables:

- Jobs

```
id (PK)
status ENUM('PENDING','SCHEDULED','DISABLED')
payload JSONB
is_recurring BOOLEAN
interval_seconds INTEGER
next_execution_time TIMESTAMPTZ
retry_count INTEGER
max_retry INTEGER
max_execution_time INTEGER
enabled BOOLEAN
Picked_by_worker BOOLEAN

created_by VARCHAR
created_at TIMESTAMPTZ
updated_at TIMESTAMPTZ
```

- job_executions

```
id (PK)
job_id (FK → jobs.id)

status ENUM('RUNNING','SUCCESS','FAILED','TIMED_OUT','RETRYING')
worker_id VARCHAR

started_at TIMESTAMPTZ
finished_at TIMESTAMPTZ
last_heartbeat_at TIMESTAMPTZ

retry_number INTEGER
log TEXT
```

- user_authentication

```
id (PK)
username VARCHAR UNIQUE
password_hash VARCHAR
roles TEXT[]
created_at
updated_at

```

3.2 Scheduler Logic

- **Cron to Scan Due Jobs**
  Each Scheduler Service instance runs a lightweight periodic task (e.g., every 1–5 seconds).
  This cron task is responsible for identifying jobs whose next_execution_time has passed and preparing them for execution.
  The cron does:

  - Scan the jobs table for due jobs
  - Lock and claim each job safely
  - Update scheduling metadata
  - Mark the job as ready for workers
  - All operations happen inside a small, fast transaction

  Multiple scheduler nodes can run this cron simultaneously without conflicts.

  - SQL Queries Used

    To safely pick due jobs in a distributed environment, the scheduler uses:

```
SELECT *
FROM jobs
WHERE next_execution_time <= NOW()
  AND status = 'PENDING'
FOR UPDATE SKIP LOCKED
LIMIT X;
```

Once a row is locked, the scheduler performs an update:

```
UPDATE jobs
SET status = 'RUNNING',
    started_at = NOW(),
    next_execution_time = :computed_next_time,
    updated_at = NOW()
WHERE id = :jobId;
```

These statements run inside a **single transaction** to enforce atomicity.

- **How Locking Works**

  PostgreSQL ensures strong consistency using row-level locks:

  - When a scheduler instance calls `SELECT … FOR UPDATE SKIP LOCKED`, the DB immediately locks the rows it returns.
    Other scheduler nodes attempting the same query:
    - **skip locked rows**
    - only pick currently unlocked rows
  - This allows **multiple schedulers to run in parallel** without conflicts or duplicates.

  Locks are released **as soon as the transaction ends**, so workers can pick jobs immediately.

  This eliminates the need for external locking systems such as Redis, Zookeeper, or leader election.

- **How Next Execution Is Computed**

The scheduler computes the next execution time *before* the worker executes the job.

One-Time Jobs

- `next_execution_time = NULL`
- `status` will later be set by completion handler

Recurring Jobs

Computed as:

`next_execution_time = previous_next_execution_time + interval_seconds`

This happens **inside the same transaction** that marks the job as RUNNING, ensuring:

- No missed intervals
- No duplicate schedules
- Same behavior across all distributed scheduler nodes

When previous execution failed

If the previous run ended with FAILURE or TIMEOUT:

- If retry_count < max_retry → scheduler computes **backoff-based** next_execution_time
- If retry_count >= max_retry → scheduler computes **next normal interval run** and resets retry_count for recurring jobs

This ensures retries are bounded and that recurring jobs continue indefinitely.

- **Scan for Completed or Failed Executions (Post-Execution Processing)**

In addition to scanning for due jobs, every Scheduler Service instance runs a second periodic task that processes completed job executions.

The scheduler scans the `job_executions` table for executions where:

    status IN ('SUCCESS', 'FAILED', 'TIMED_OUT') AND not yet processed

For each finished execution:

If SUCCESS:

- **Recurring jobs:**
    - Reset `retry_count = 0`
    - Set job `status = PENDING`
- **One-time jobs:**
    - Set job `status = SUCCESS`
- **If FAILED or TIMED_OUT:**
- **Recurring jobs:**
    - If `retry_count < max_retry`:
        - increment `retry_count`

            - compute retry backoff
              `next_execution_time = NOW() + backoff(retry_count)`
            - set job `status = PENDING`
        - Else (`retry_count >= max_retry`):
            - reset `retry_count = 0`
            - compute next regular interval run
              `next_execution_time = previous_next_execution_time + interval_seconds`
            - set job `status = PENDING`
- **One-time jobs:**
    - If `retry_count < max_retry`:
        - increment retry_count
        - compute retry backoff
        - set job `status = PENDING`
    - Else:
        - set job `status = FAILED`
        - set `next_execution_time = NULL`

3.3 Worker Execution Lifecycle

- ExecutorService

  Each Worker Service instance contains an internal `ExecutorService` thread pool that executes jobs concurrently.
  The job lifecycle inside the worker is:

  1. Worker polls the `jobs` table for entries with:
      - `status = 'RUNNING`
      - and **not yet picked by any worker**
        (using `FOR UPDATE SKIP LOCKED` to safely claim the job)
  2. Once a job is claimed, the worker:
      - marks it as picked (e.g., `picked_by_worker = true` or similar)
      - creates a new entry in `job_executions` with:
          - `status = 'RUNNING'`
          - `started_at = NOW()`
          - `retry_number = current retry_count`
          - `worker_id = this instance's ID`
  3. The job payload is submitted to the `ExecutorService` thread pool for execution.
     The worker does **not** hold any database lock during execution.

- Heartbeats

  To detect crashes or stuck tasks, each executing job emits periodic heartbeats:

  - A scheduled heartbeat thread updates: `job_executions.last_heartbeat_at = NOW()` every few seconds.
  - If the worker process or thread crashes, the heartbeat naturally stops. The scheduler later detects stale heartbeats and treats the execution as a hung job.

  This mechanism enables **distributed crash recovery** without a worker registry.

- Max execution timeout

  Every worker runs a background timeout checker that monitors all RUNNING executions.

  A job is considered "hung" when:

  ```
  NOW() - started_at > max_execution_time
  ```

  On timeout:

  - The worker updates `job_executions.status = 'TIMED_OUT'`
  - `finished_at` is set
  - A failure log is stored
  - The worker stops tracking the job

  The scheduler will later reschedule or retry the job according to retry logic.

  This ensures no job runs indefinitely due to unresponsive external systems.

- Status transitions

  Jobs transition through the following states:

  1. PENDING

     The job is scheduled and waiting for the scheduler to pick it.

  2. RUNNING

     Scheduler sets:

     - `status = 'RUNNING'`
     - `started_at = NOW()`
     - `next_execution_time` (precomputed for recurring jobs)Worker picks the job and creates a `job_executions` record.

3. SUCCESS

   The job logic completes without errors.

   Worker updates:

   - `job_executions.status = 'SUCCESS'`
   - `finished_at = NOW()`
   - `log` (optional)

   Scheduler later sets:

   - `job.status = PENDING` (for recurring)
   - OR `job.status = SUCCESS` and disables further execution (for one-time)

4. FAILED

   If the job throws an exception, returns an error, or explicitly fails:

   Worker updates:

   - `job_executions.status = 'FAILED'`
   - `finished_at = NOW()`
   - `log` with error details

   Scheduler later decides:

   - retry (if retry_count < max_retry)
   - or mark permanent failure (one-time jobs)

5. RETRY_PENDING

   Not an explicit worker state — but a scheduler state.

   Worker reports failure → scheduler computes backoff and sets status back to `PENDING` for retry.

3.4 Retry & Backoff Design

- Configurable retry count

  Chronos allows users to specify the maximum number of retries for any job:

  - `max_retry` is stored in the `jobs` table
  - `retry_count` tracks how many times the job has retried so far

    - Scheduler increments `retry_count` only after a FAILED or TIMED_OUT execution
    - Workers never modify retry_count

  Once retry_count reaches `max_retry`:

    - **One-time jobs** → permanently marked as `FAILED`
    - **Recurring jobs** → skip retrying this run and continue with the next scheduled interval

  This ensures predictable retry behavior and prevents infinite retry loops.

- Backoff algorithm

  retry_delay = base_delay * (2 ^ retry_count)

  Chronos uses an exponential backoff.

3.5 Error Handling

- Transient failures

  Transient failures are temporary issues that are likely to succeed if tried again.

  Worker behavior:

    - Worker marks the job execution as `FAILED`
    - Records error details into `job_executions.log`
    - Sets `job_executions.status = FAILED`

  Scheduler behavior:

    - If `retry_count < max_retry`:
        - Increment `retry_count`
        - Apply exponential backoff to compute retry delay
        - Set job `status = PENDING` for retry
    - No need to disable or stop the job
    - Recurring jobs continue their next scheduled interval regardless of failure

  This ensures that transient issues do not permanently disrupt the system.

- Permanent failures

  Permanent failures occur when retrying will **not** fix the issue.

Worker behavior:

- Marks `job_executions.status = FAILED`
- Logs full error traceback or message

Scheduler behavior:

- If `max_retry` is reached:
  - **One-time jobs:**
    - Mark job `status = FAILED`
    - Set `next_execution_time = NULL`
  - **Recurring jobs:**
    - Reset `retry_count = 0`
    - Continue with the next scheduled recurring run
    - (Recurring jobs must NOT be canceled permanently)

This ensures fatal errors stop one-time jobs, but recurring jobs remain healthy.

- Dead letter behavior

Chronos provides a simple dead-letter mechanism through historical storage:

- All executions (success or failure) are recorded in `job_execution.`
- When a job permanently fails:
  - Its last execution entry functions as the "dead-letter record"
  - All past retries and failure logs remain available for debugging

Dead letter queue is NOT a separate table in v1.
Instead, Chronos treats:

```
job_executions where status = FAILED AND job.status =
FAILED
```

as your effective dead-letter entries.

---

4. REST API Endpoints

Endpoint	Method	Description
`/jobs`	POST	Create a one-time job

`/jobs/recurring`	POST	Create a recurring job
`/jobs/{id}`	GET	Retrieve job details and current status
`/jobs/{id}`	DELETE	Cancel/disable a job
`/jobs/{id}/executions`	GET	Fetch execution history for a job
`/jobs`	GET	List all jobs (optional filtering)
`/jobs/{id}/retry`	POST	Force a manual retry of a job (optional)
`/monitoring/jobs`	GET	System-wide job statistics
`/monitoring/workers`	GET	Worker health metrics (optional)
`/auth/login`	POST	Authenticate user and return JWT token
`/auth/signup`	POST	Create a new user

---

5. Implementation Details

5.1 Technologies Used

- **Spring Boot**
  Used to build both the Scheduler Service and Worker Service.
  Provides REST APIs, background tasks, and dependency injection.

- **ExecutorService**
  The Worker uses Java's ExecutorService for parallel job execution through a configurable thread pool.
- **PostgreSQL**
  Stores job definitions, execution history, and coordinates distributed scheduling using
  `SELECT … FOR UPDATE SKIP LOCKED`.

5.2 Design Decisions

Brief justification for:

- Why Postgres locking

  PostgreSQL row-level locks with `SKIP LOCKED` provide a reliable and simple distributed coordination mechanism:

  - Prevents multiple scheduler nodes from picking the same job
  - Prevents multiple worker nodes from executing the same job
  - No need for external locking systems like Redis or Zookeeper
  - Strong transactional guarantees (ACID)
  - Allows full horizontal scaling of both schedulers and workers

  This keeps the system lightweight yet fault-tolerant.

- Why no queue (for now)

  A message queue (Kafka, RabbitMQ, etc.) is not required in v1 because:

  - PostgreSQL already guarantees safe single-worker job assignment
  - Workers can directly poll the DB for RUNNING jobs
  - Adding a queue increases operational complexity
  - No need for decoupling scheduler → worker communication at current scale

  Queues may be considered in future versions if throughput reaches extremely high levels.

- Why multi-node architecture

  Both the Scheduler and Worker Services are built to run as multiple instances:

  - **Multiple Scheduler nodes** process due jobs faster
  - **Multiple Worker nodes** increase execution throughput
  - Stateless services allow load balancers to distribute requests evenly
  - The shared database ensures consistency across all node.

This architecture enables fault tolerance and linear scalability.

5.3 Observability

- Log structure

  Chronos logs essential events for debugging and auditability:

  - Job creation
  - Job picked by scheduler
  - Worker claiming a job
  - Heartbeat updates
  - Job execution start/end
  - Failures and exceptions
  - Retry attempts
  - Timeout events

  Execution-level logs are stored persistently in the `job_executions.log` field.

- Monitoring metrics

  The system can expose metrics such as:

  - Total jobs created
  - Pending jobs
  - Running job
  - Successful / Failed / Timed-out executions
  - Average execution time
  - Retry counts
  - Stuck/hung job count
  - Worker throughput
  - Scheduler scan latency

  These metrics help operators track health and performance.

- Tracking failures

  Failures are handled at two layers:

  1. **job_executions table**
     i. Stores statuses per execution
     ii. Captures logs, timestamps, and retry number

  2. **jobs table**
     i. Tracks retry_count, max_retry
     ii. Stores final job status (SUCCESS, FAILED, DISABLED)
     iii. Records next_execution_time for future scheduling

This structure ensures that both real-time job state and historical executions are fully traceable.

---

6. Frontend Plan (Next.js + Shadcn)

6.1 Dashboard Pages

- **Jobs List Page**
  Displays all jobs with their current status (PENDING, RUNNING, SUCCESS, FAILED).
  Includes filters for recurring jobs, failed jobs, and next execution time.
- **Create Job Page**
  Form to create one-time or recurring jobs.
  Inputs include payload, interval, max retries, and execution time.
- **Job Details & Execution Logs Page**
  Shows job metadata and a table of past executions with timestamps, status, and logs.
- **System Health Monitor Page**
  Shows simple metrics like:
  - total jobs
  - running jobs
  - failed jobs
  - worker node count
  - scheduler activity

Minimal charts or counters only.

6.2 UI Components

Shadcn Forms

Used for job creation and editing:

- text inputs
- number pickers
- JSON editors
- date/time pickers

Tables

shadcn DataTable components for:

        - job list
        - execution logs
        - worker status list

Charts (Light Usage)

Simple line or bar charts (using Recharts) for:

        - job success vs failure
        - execution duration trends

---

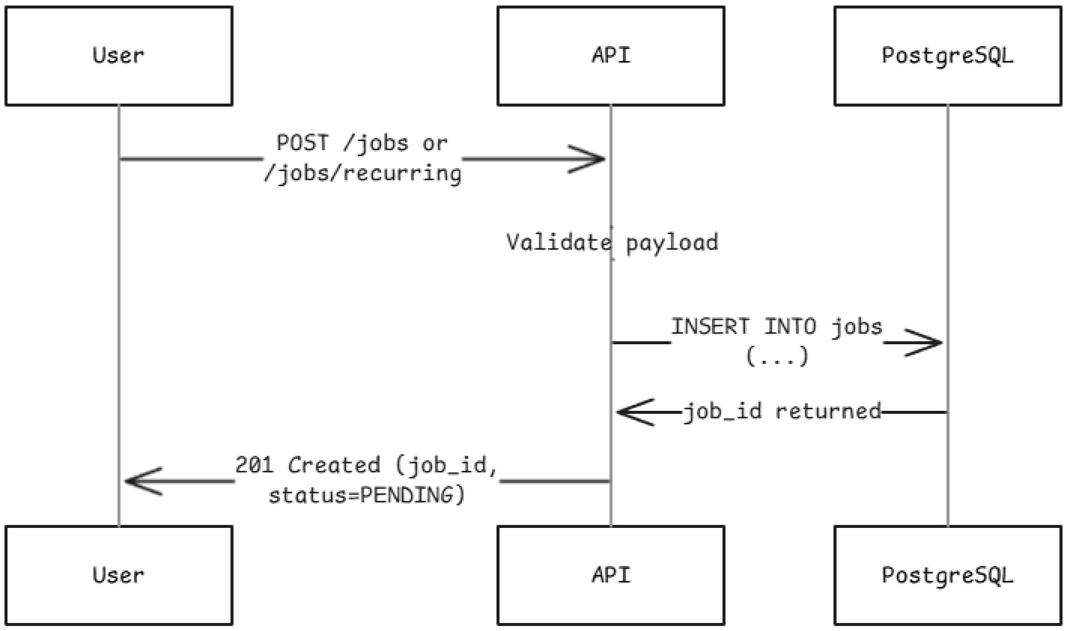
7. Future Enhancements

- Queue-based execution (Kafka/Redis)
- Multi-tenant support
- Worker autoscaling
- Priority scheduling
- Cron expression support

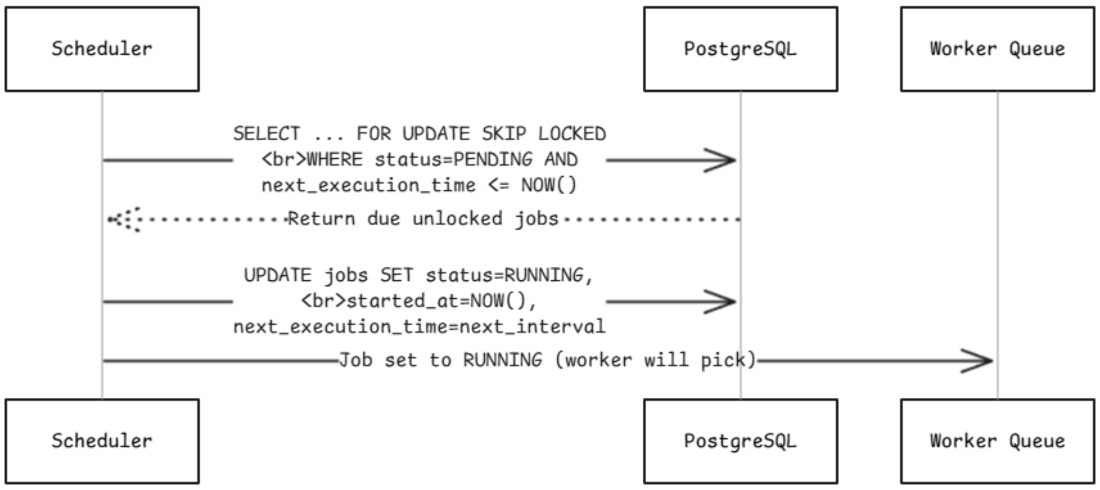
---

8. Appendix

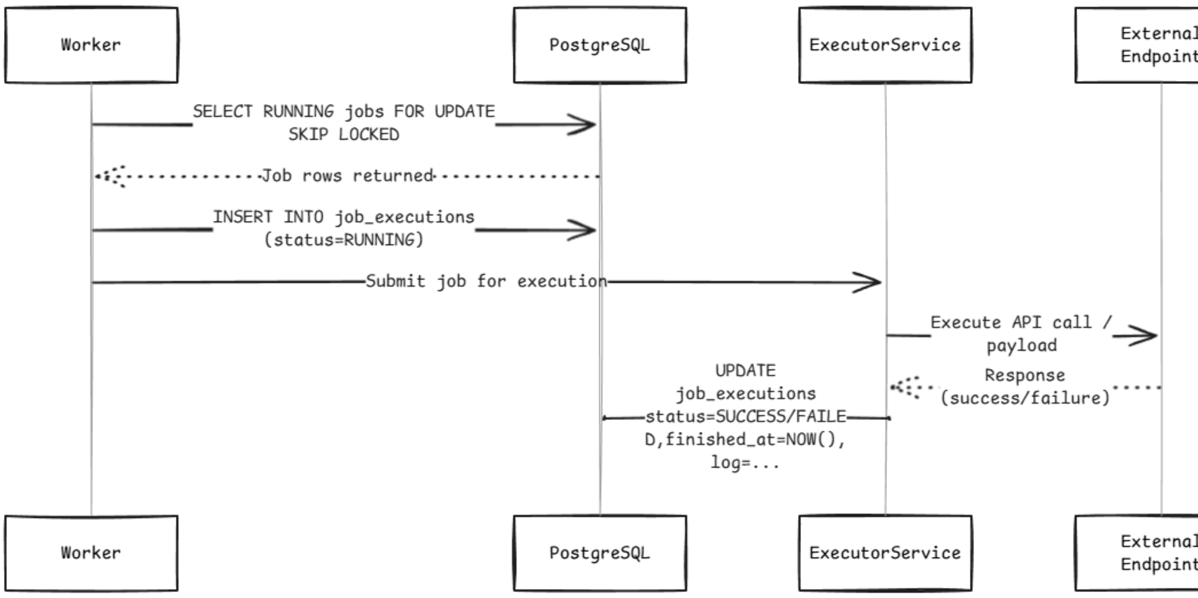
- Sequence diagrams
    1. Job Creation



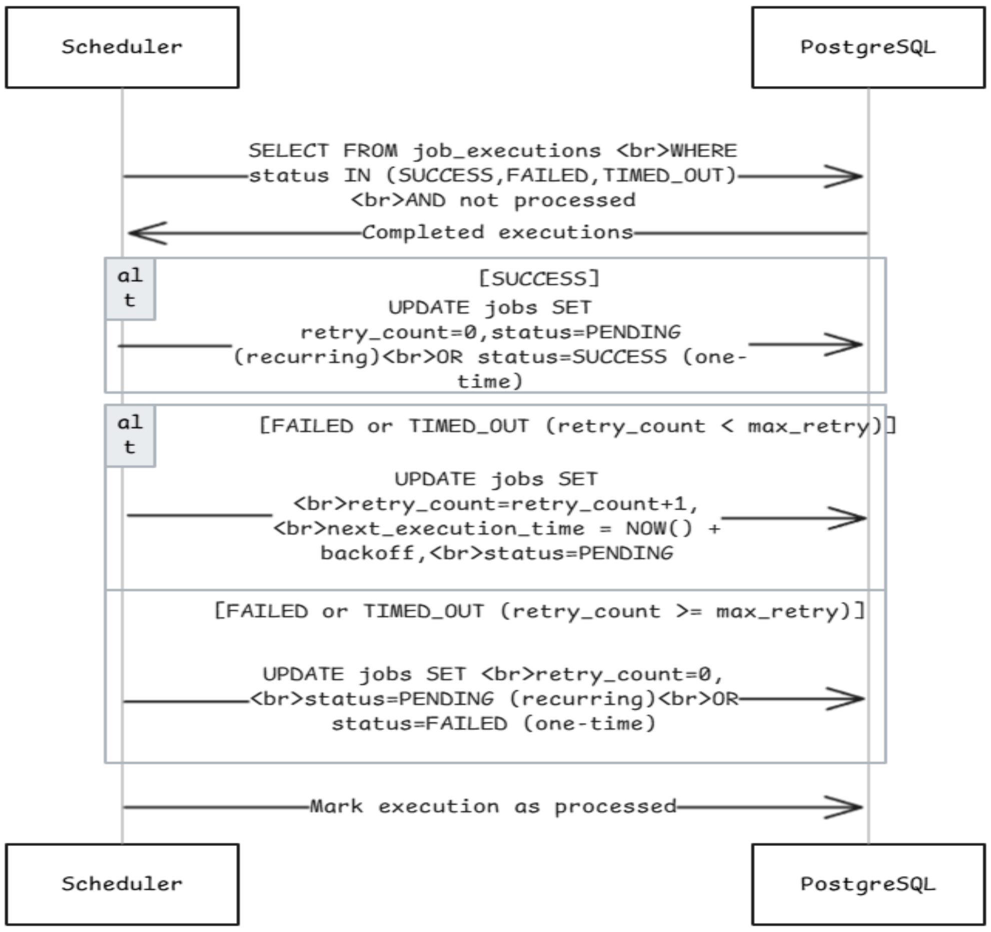
2. Scheduler Scans and Picks Due Jobs


Scheduler
PostgreSQL
Worker Queue
SELECT ... FOR UPDATE SKIP LOCKED
WHERE status=PENDING AND next_execution_time <= NOW()
Return due unlocked jobs
UPDATE jobs SET status=RUNNING,
started_at=NOW(), next_execution_time=next_interval
Job set to RUNNING (worker will pick)
Scheduler
PostgreSQL
Worker Queue

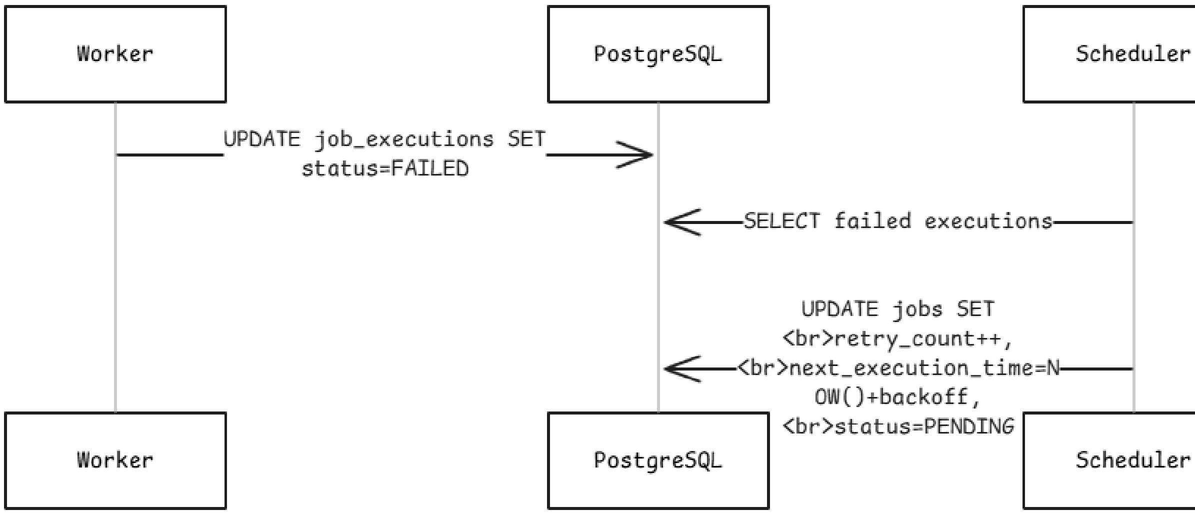
3. Worker Execution Lifecycle


Worker
PostgreSQL
ExecutorService
External Endpoint
SELECT RUNNING jobs FOR UPDATE SKIP LOCKED
Job rows returned
INSERT INTO job_executions (status=RUNNING)
Submit job for execution
Execute API call / payload
Response (success/failure)
UPDATE job_executions status=SUCCESS/FAILED,finished_at=NOW(), log=...
Worker
PostgreSQL
ExecutorService
External Endpoint

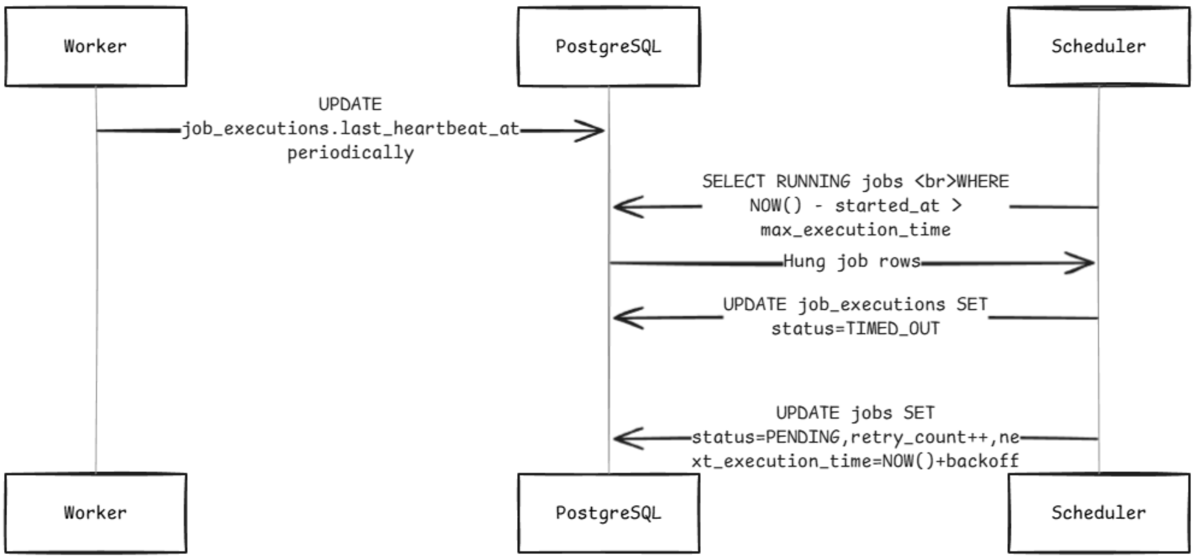
4. Execution Completion Handler


Scheduler
PostgreSQL
SELECT FROM job_executions
WHERE status IN (SUCCESS,FAILED,TIMED_OUT)
AND not processed
Completed executions
alt
[SUCCESS]
UPDATE jobs SET retry_count=0,status=PENDING (recurring)
OR status=SUCCESS (one-time)
alt
[FAILED or TIMED_OUT (retry_count < max_retry)]
UPDATE jobs SET
retry_count=retry_count+1,
next_execution_time = NOW() + backoff,
status=PENDING
[FAILED or TIMED_OUT (retry_count >= max_retry)]
UPDATE jobs SET
retry_count=0,
status=PENDING (recurring)
OR status=FAILED (one-time)
Mark execution as processed
Scheduler
PostgreSQL

5. Retry Flow (Transient Failure)


Worker
PostgreSQL
Scheduler
UPDATE job_executions SET status=FAILED
SELECT failed executions
UPDATE jobs SET
retry_count++,
next_execution_time=NOW()+backoff,
status=PENDING
Worker
PostgreSQL
Scheduler

6. Timeout / Heartbeat Failure Recovery


Worker
PostgreSQL
Scheduler
UPDATE job_executions.last_heartbeat_at periodically
SELECT RUNNING jobs
WHERE NOW() - started_at > max_execution_time
Hung job rows
UPDATE job_executions SET status=TIMED_OUT
UPDATE jobs SET status=PENDING,retry_count++,next_execution_time=NOW()+backoff
Worker
PostgreSQL
Scheduler