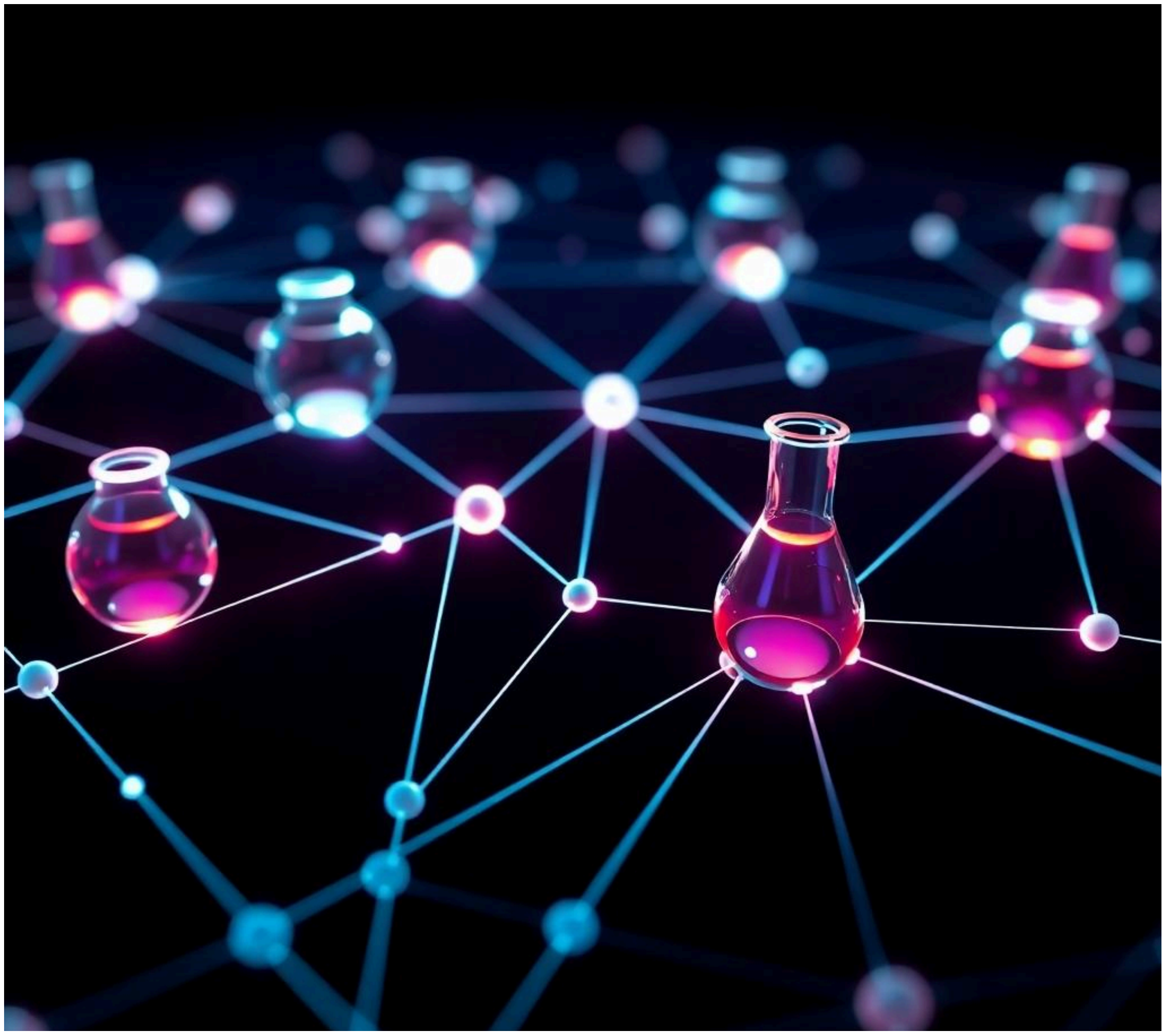
Distributed Lab Test Scheduler Design Document

For an imaginary science lab service

by Mario Linge

https://www.linkedin.com/in/mariolinge/



About this design document

This is a design document as an exercise to showcase a potential application and infrastructure set-up for a biotech or chemistry science lab service. This imaginary science lab service needs to perform and schedule measurements of lab tests and experiments. Those measurements need to be done at a precise time. There are potentially thousands of tests done each day, therefore the scale and reliability are of highest priority.

1. Overview

1.1 Functional Requirements

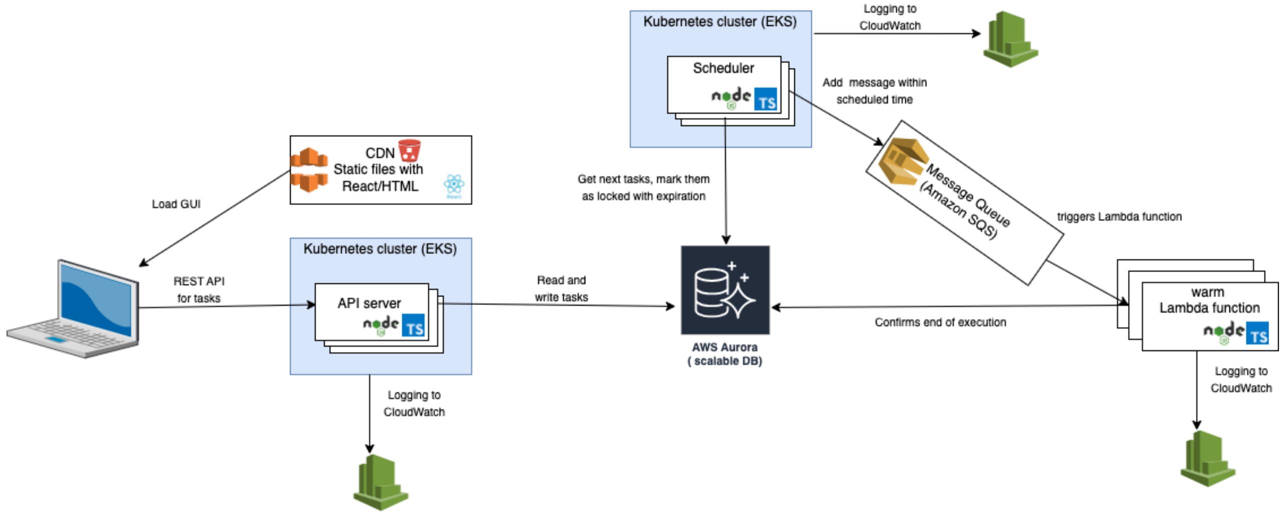
1. **Scheduling Lab Tests:**
    - The system must allow users (lab technicians, scientists) to schedule both one-time and recurring lab test measurements.
    - Each lab test is assigned a specific time or Cron-based schedule.
2. **Execution of Lab Tests:**
    - When a scheduled time arrives, the system must trigger a process (via AWS Lambda) that performs the lab measurement or retrieves/validates results.
3. **Task Management and Tracking:**
    - Users must be able to create, view, update, and delete lab test tasks.
    - Each task's status (e.g., scheduled, locked, completed, failed) should be tracked.
4. **Logging and Monitoring:**
    - The system must log each task's execution details and provide monitoring capabilities for health checks and error detection.

1.2 Non-Functional Requirements

1. **High Availability:**
    - The system should tolerate component failures without losing data or missing scheduled executions.
2. **Data Durability:**
    - All lab test tasks and results must be stored reliably in a persistent data store.
3. **Scalability:**
    - The system should handle an increasing number of scheduled tasks and lab test operations, scaling both up and down based on demand.
4. **Cost-Effectiveness:**
    - The design should use managed services (e.g., AWS Aurora, AWS Lambda) and container orchestration to minimize operational overhead and costs.

2. Architecture

Below is an overview of the architecture:



2.1 Components

1. **CDN (React/HTML Frontend):**
    - Hosts static assets for the user interface.
    - Provides quick load times worldwide
2. **API Server (Node.js + TypeScript in Kubernetes):**
    - Exposes REST endpoints for creating, viewing, updating, and deleting lab test tasks and potentially more like adding new lab tests
    - Interacts with the AWS Aurora database for storing and retrieving task information.
    - Sends logs to CloudWatch (or another logging mechanism).
3. **AWS Aurora (Scalable DB):**
    - Stores task metadata (e.g., type of lab test, schedule, status) and execution logs.
    - Provides high durability and automatic scaling.
4. **Scheduler (Node.js + TypeScript in Kubernetes):**
    - Periodically polls the database to find tasks that are due.
    - Performs an atomic "SELECT … FOR UPDATE" to lock tasks with an expiration, ensuring no duplicate pickups in a multi-scheduler environment.
    - Publishes a message to AWS SQS once a task is locked by marking it and adding expiration date.
5. **AWS SQS (Message Queue):**
    - Decouples the scheduling from the execution phase.
    - Is very scalable and and reliable an cost effective
    - Buffers task messages and delivers them reliably to the Lambda function.

    - has dead letter queue
6. **Warm Lambda Function (Node.js):**
    - Triggered by new messages in the SQS queue.
    - A pool of Lambda functions are already “warmed up” to minimize the loading time
    - Executes the lab test logic (e.g., runs a measurement, retrieves data, processes results).
    - Updates the database to mark the task as completed or failed, and logs execution details.
7. **AWS CloudWatch (Logging & Monitoring):**
    - Aggregates logs from the API server, the scheduler, and the Lambda function.
    - Provides monitoring metrics (CPU, memory, function invocation counts, etc.).

2.2 Rationale and High-Level Requirements

1. **High Availability:**
    - The API server and scheduler run in Kubernetes with multiple replicas, ensuring no single point of failure.
    - AWS Aurora offers multi-AZ replication for continuous data availability.
2. **Data Durability:**
    - Task details are stored in Aurora, which uses distributed storage and automatic backups.
    - Locking logic ensures that tasks are not lost or executed twice.
3. **Scalability:**
    - Kubernetes can horizontally scale the API server and the scheduler pods.
    - AWS SQS and Lambda automatically handle spikes in the number of scheduled tasks, with Lambda scaling up to meet demand.
4. **Cost-Effectiveness:**
    - By leveraging AWS Lambda, the system pays only for the compute time needed to run lab test logic, reducing idle costs.
    - Aurora serverless or provisioned modes can scale storage and compute on demand, avoiding over-provisioning.

---

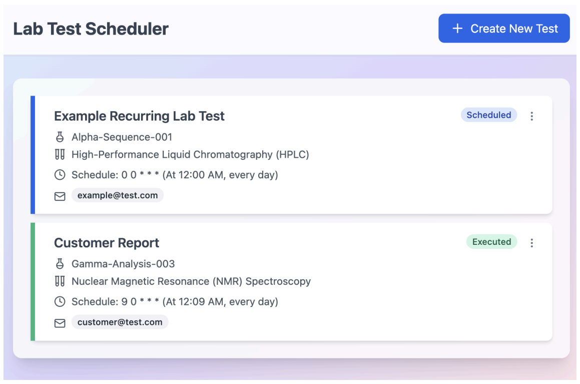
3. Implementation

3.1 Task Creation Workflow

1. **User Action:** A lab technician uses the React-based GUI (served by the CDN) to create a new lab test task.
2. **API Request:** The request is sent to the API server, which validates the data and stores the new task in Aurora.
3. **Acknowledgment:** The API returns a success response, and the task appears in the user’s list of scheduled tasks.

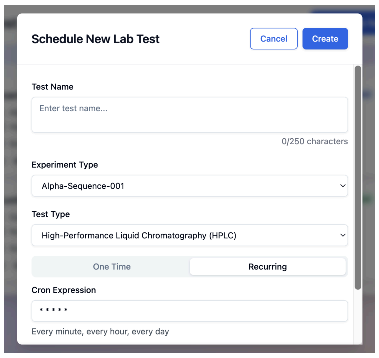
3.1 Graphical User Interface (GUI)

- Landing page displays a scrollable, paginated list of upcoming lab tests (title, type, experiment, schedule, recipients of report).
- A “Create Test” button in the header opens a modal to enter task details, including title, type, experiment, and time-based or Cron-based scheduling.
- Each list item can invoke an action menu to edit, show logs, or delete the task.
- Fully responsive layout ensures usability on both desktop and mobile devices.



Example Recurring Lab Test
Scheduled
Alpha-Sequence-001
High-Performance Liquid Chromatography (HPLC)
Schedule: 0 0 * * * (At 12:00 AM, every day)
example@test.com

Customer Report
Executed
Gamma-Analysis-003
Nuclear Magnetic Resonance (NMR) Spectroscopy
Schedule: 9 0 * * * (At 12:09 AM, every day)
customer@test.com



Test Name
Enter test name...
0/250 characters
Experiment Type
Alpha-Sequence-001
Test Type
High-Performance Liquid Chromatography (HPLC)
One Time [✓ Recurring]
Cron Expression
* * * * *
Every minute, every hour, every day

Landing page and creation form

3.2 Scheduling and Locking

1. **Scheduler Polling:** A scheduler pod in Kubernetes periodically queries Aurora for tasks that are due or nearly due.
2. **Row Locking:** The scheduler uses a `SELECT ... FOR UPDATE` statement to atomically lock a task, setting `locked_at` and `lock_expires_at`.
3. **SQS Publish:** The scheduler publishes a message (including the task ID and any relevant metadata) to AWS SQS.

3.3 Task Execution

1. **Lambda Invocation:** AWS SQS triggers the Lambda function when a new message arrives in the queue.
2. **Lab Test Logic:** The Lambda function performs the necessary steps (e.g., calling external measurement APIs, running data analysis, or performing the test logic).
3. **Result Logging:** The Lambda function updates the task status in Aurora (e.g., `completed` or `failed`) and writes execution logs (e.g., timestamps, results, errors).

3.4 Monitoring and Logging

- **AWS CloudWatch:** All logs from the API server, scheduler, and Lambda function are streamed to CloudWatch, enabling a unified view of system behavior, performance metrics, and error handling.

---

4. Choke Points, Limits, and Mitigations

1. **Database Lock Contention:**
   - **Choke Point:** High concurrency in locking tasks can cause contention on Aurora.
   - **Mitigation:** Optimize polling frequency, shard tasks if needed, or implement a distributed approach (e.g., a leader scheduler for high-volume workloads).
2. **SQS Throughput & Lambda Concurrency:**
   - **Choke Point:** Large spikes in scheduled tasks can create a high volume of messages.
   - **Mitigation:** AWS SQS and Lambda scale automatically, but concurrency limits may need to be raised. Dead-letter queues can capture failed or delayed tasks for later reprocessing.
3. **Kubernetes Resource Constraints:**
   - **Choke Point:** If the API server or scheduler pods are undersized, performance may degrade.
   - **Mitigation:** Use Horizontal Pod Autoscaling based on CPU/memory usage or custom metrics (e.g., tasks in queue).
4. **Lambda Cold Starts:**
   - **Choke Point:** Infrequent function invocations may trigger cold starts.
   - **Mitigation:** Configure provisioned concurrency (i.e., warm Lambda) for time-critical tasks.
5. **Cost vs. Performance Balance:**
   - **Choke Point:** Over-scaling can lead to high costs, while under-scaling can hurt performance.
   - **Mitigation:** Monitor metrics (API usage, SQS queue length, DB capacity) to balance cost and performance.

Conclusion

This design meets the high-level requirements of availability, durability, scalability, and cost-effectiveness by combining managed AWS services (Aurora, SQS, Lambda) with Kubernetes for container orchestration. The system can handle lab test scheduling and execution in a robust, flexible manner, ensuring that each measurement is processed on time, and logs are captured for audit and analysis. By leveraging proven patterns (row locking, message queues, autoscaling), this architecture remains adaptable to future growth and complexity.