Design a Distributed Job Scheduler for Millions of Tasks in Daily Operations

Unpacking Challenges, Assumptions, and the Scope of Distributed Job Scheduler

Welcome to the dynamic world of modern computing. Orchestrating millions of daily tasks is essential in our evolving digital landscapes. In this universe of distributed systems, a robust job scheduler is pivotal. Join me as we explore crafting a distributed job scheduler for seamless operations.

Sometimes, we need to *set up our job execution based on our user's configuration*. For example,

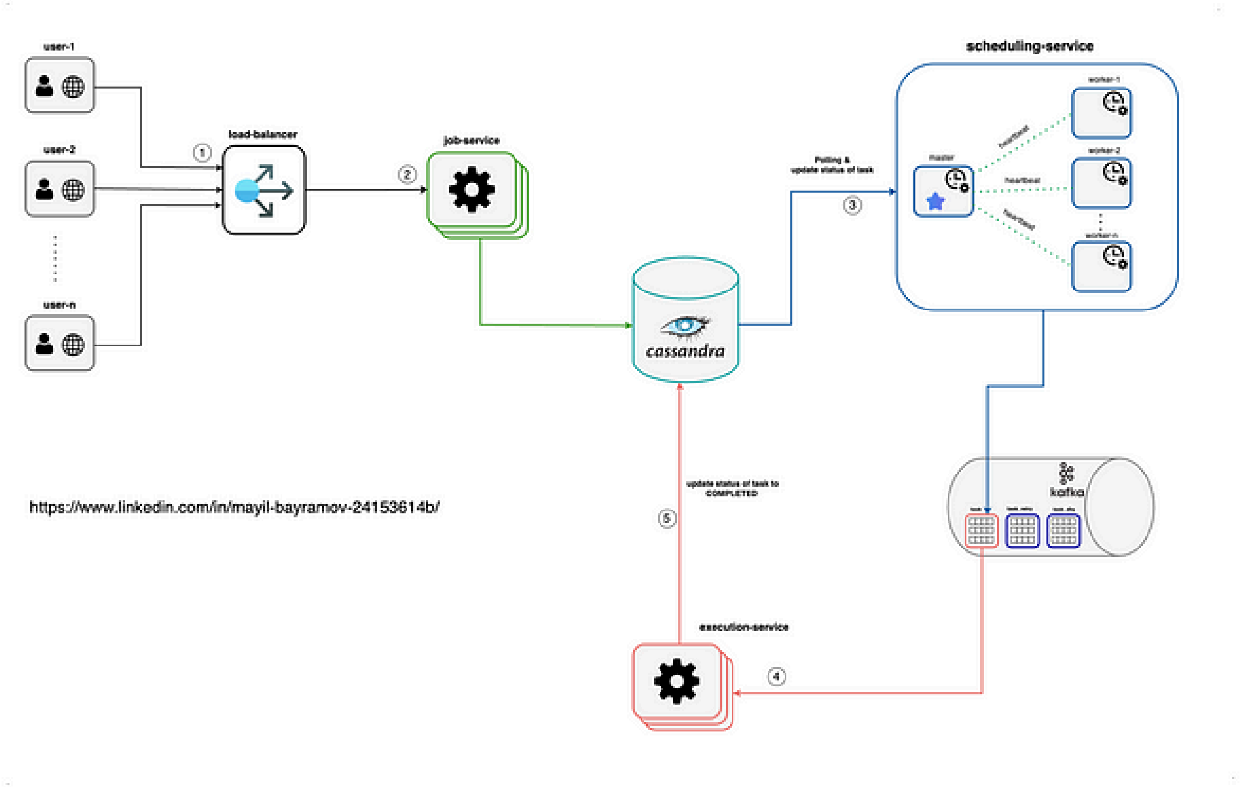
> *Imagine we have a web platform where user logs in, create new invoices, and send them to their customers. Our business team has introduced a new feature, allowing users to create invoices and schedule them for later. This process provides the option for configuration, allowing users to choose between a* ***one-time execution*** *or a* ***recurring style.***

Another example could be scheduled email, push notification, or other processes. In this post, we will design a highly scalable Job Scheduler in a Distributed environment that can execute millions of tasks each day. This architecture is versatile, addressing scheduled job execution for different types of businesses and diverse use cases.

For our system let's presume that the given requirements are like this.

- User actions: **Create and delete jobs. Retrieve a list of jobs created by the user. View the execution history for any given job.**
- Job execution style: **The system should support both one-time execution, recurring style**
- Number of average estimated daily tasks executions: **50 million tasks each day**
- Task fail strategy: **The system should have the capability to support a *retry* feature**
- Our system is configured to perform task executions **every minute**

Design a Job Scheduler in a Distributed environment



As you can see in the diagrams there are many steps(every step was numbered) and I will elucidate each step for you to make it all clear.

Optimal Database Selection

In any well-designed system, a robust database schema is a crucial element. It directly can affect the overall performance and reliability of the system. Let's overview and choose the appropriate database based on our requirements.

Retrieve Operations

- Given a `user_id`, retrieve all jobs created by that user
- Retrieve *all scheduled tasks that are scheduled to be executed right now* **(by scheduling service**, operation**(3)** in the system-design-1 diagram)
- Retrieve **all** or the **latest execution** histories and their status related to a specific `job_id`

Modification Operations

- Allow users to *create* or *delete* their job schedules
- Update the *Job's next execution timestamp* by`job_id`
- Update task status (Scheduled, Completed, Failed)

As you can see we have both **read** and **write** operations. If you pay special attention we have we don't have complex transactional operations, therefore we are not tightly coupled to use a relational database.

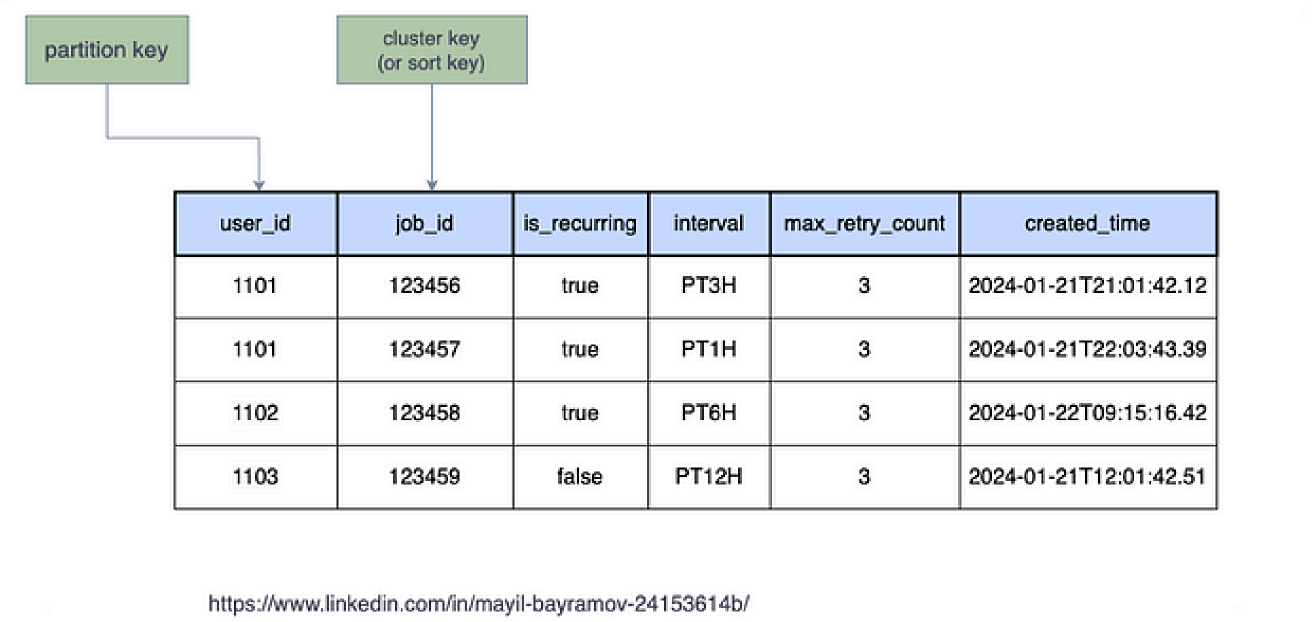
As we have mentioned above our estimated daily tasks to be executed are 50 million. If we would average system load **per minute** based on this formula (daily tasks)/(24*60) => 50 000 000/24*60 ≈ 34722 tasks/minute.

In addition to executing tasks at the scheduled time, we need to efficiently retrieve tasks that are currently scheduled to be executed. In our design we will choose **Cassandra**, as it is an open-source, distributed NoSQL database system designed to handle large amounts of data across multiple commodity servers, ensuring high availability and fault tolerance. If you are considering using cloud services also Amazon DynamoDB (AWS), and Bigtable (GCP) are recommended options for use. (p.s. I'll introduce a well-structured schema in this post, providing the flexibility to work seamlessly with also relational databases like PostgreSQL and MySQL)

Schema Design

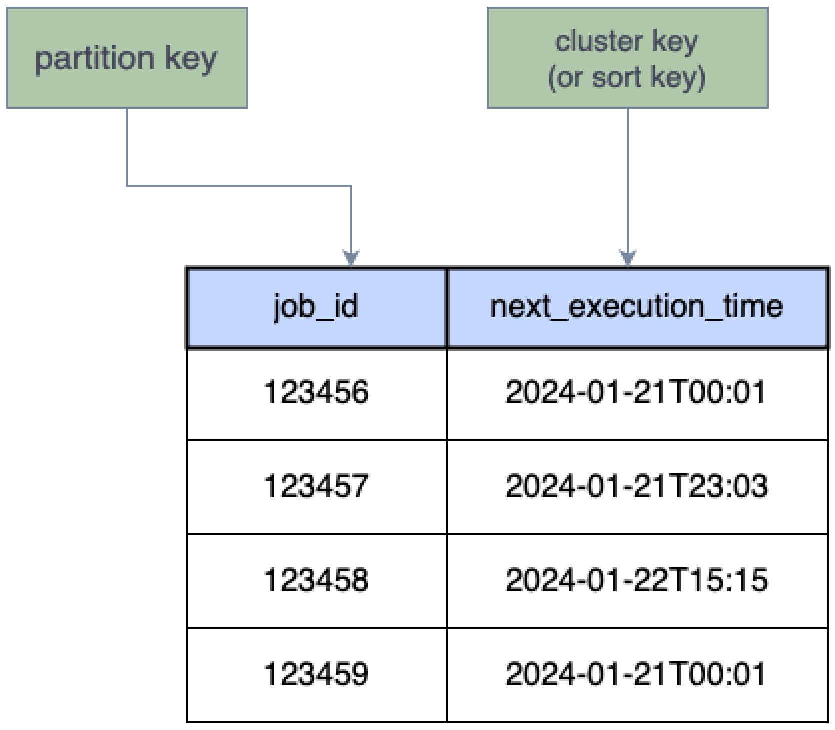
Job table design: When the user creates a job in our platform, we need a table that stores the user's job and job-related metadata, like retry-interval, recurring, max-retry count and etc. (you can extend these metadata based on your system need) we will store in that table. Remember Cassandra is a NoSQL database and does not support traditional SQL-style joins. The data in a table should in a denormalized format. For this reason, we will keep our data denormalized format in our table.

As each user will retrieve all their jobs using their `user_id`, it would be beneficial to **partition the table by** `user_id` to enhance data locality, optimize query performance, and streamline the retrieval of job information specific to individual users. In each partition, utilize the `job_id` as a **sorted key** for efficient organization and retrieval of job-related data.


partition key
cluster key (or sort key)
https://www.linkedin.com/in/mayil-bayramov-24153614b/

user_id	job_id	is_recurring	interval	max_retry_count	created_time
1101	123456	true	PT3H	3	2024-01-21T21:01:42.12
1101	123457	true	PT1H	3	2024-01-21T22:03:43.39
1102	123458	true	PT6H	3	2024-01-22T09:15:16.42
1103	123459	false	PT12H	3	2024-01-21T12:01:42.51

Task_Schedule *table design:* In the above, we designed a **job** table and we mentioned the job execution interval there. For example, *job* with id-*123456* is scheduled to execute every 3 hours (PT3H- means **P**eriod **T**ime **3 Hours**). But we need another table to store the next execution time for each job. By doing this it will be easy for us to query all scheduled tasks for the current minute(as we execute our tasks in each minute). The initial or basic design for the **task_schedule** table would typically be like the one below.

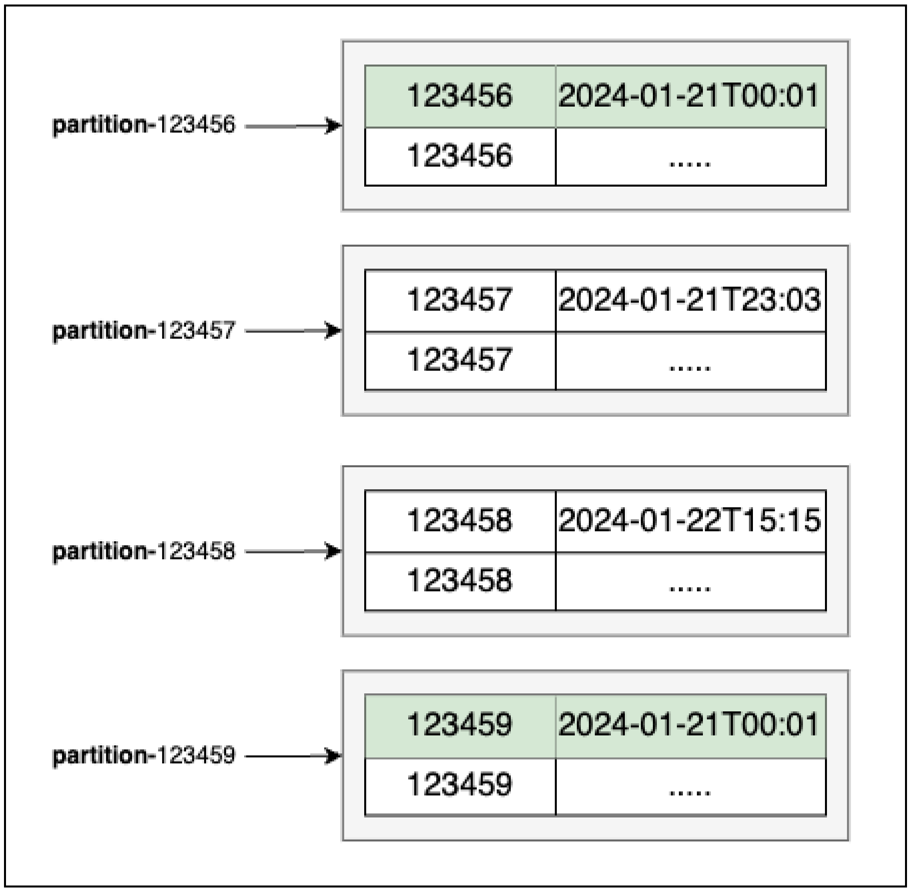


job_id	next_execution_time
123456	2024-01-21T00:01
123457	2024-01-21T23:03
123458	2024-01-22T15:15
123459	2024-01-21T00:01

https://www.linkedin.com/in/mayil-bayramov-24153614b/

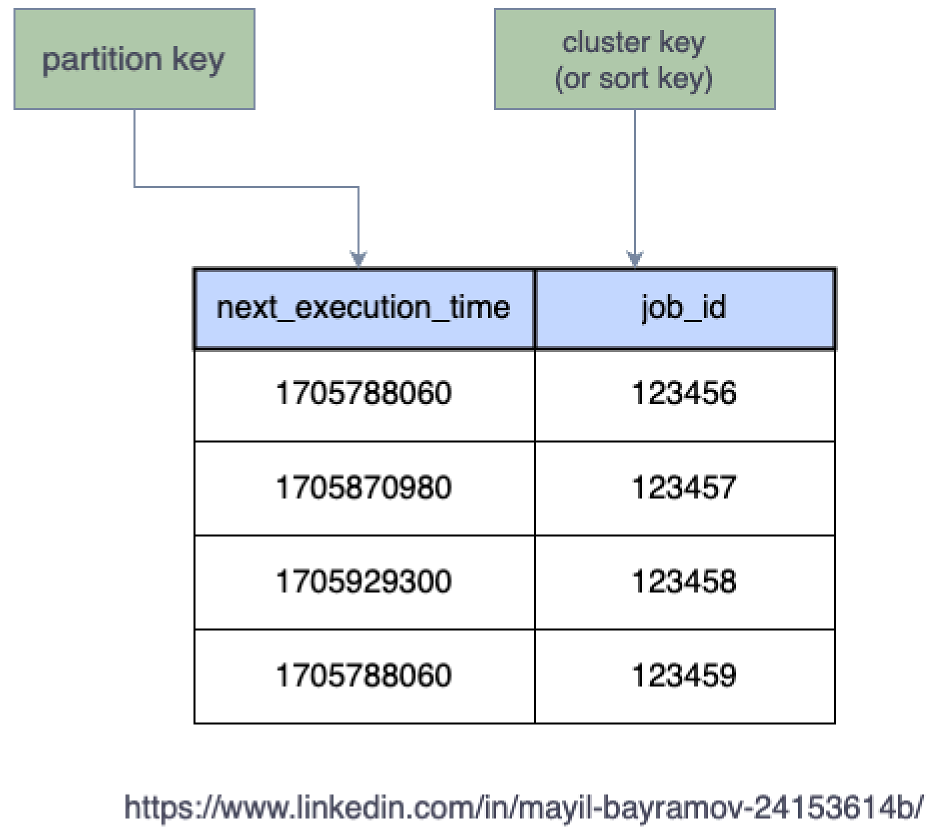
If we query this table every minute, the corresponding query would look like this:
```
select * from task_schedule where next_execution_time = '21/01/2024:00:01'
```

If we analyze this query execution plan, we will see that even though `next_execution_time` is a sorted field but query will scan every partition in *Cassandra cluster*. Because there is a possibility that the requested data, despite the sorted nature of `next_execution_time`, spans multiple partitions. In such cases, the partition key alone may not fully optimize the search, leading to the need for a cluster-wide scan during each query execution. I'll follow this statement with an accompanying diagram to provide a visual representation of the described process.



In the diagram above, you can observe that our query has identified two matching rows. However, since these rows are situated in different partitions, the query necessitates scanning all partitions to retrieve the matched rows. As we are talking big data we can have hundreds of thousands of partitions in the production database, to scanning all these these partitions can lead to a substantial decrease in performance.

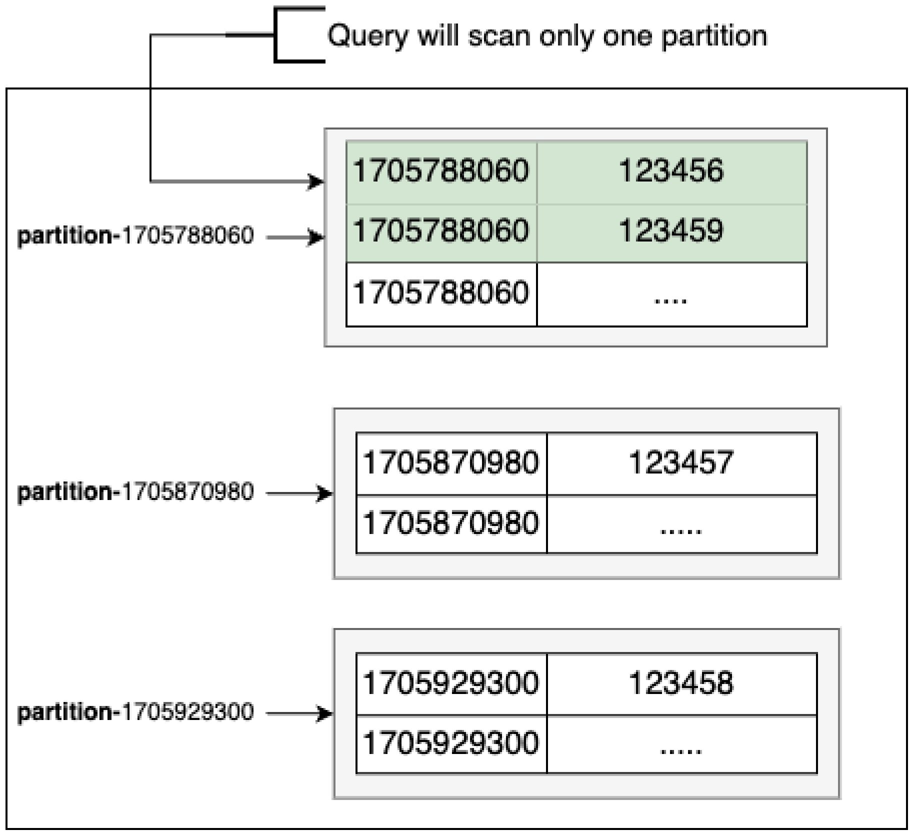
Let's enhance the structure of the **task_schedule** table to optimize its performance.



next_execution_time	job_id
1705788060	123456
1705870980	123457
1705929300	123458
1705788060	123459

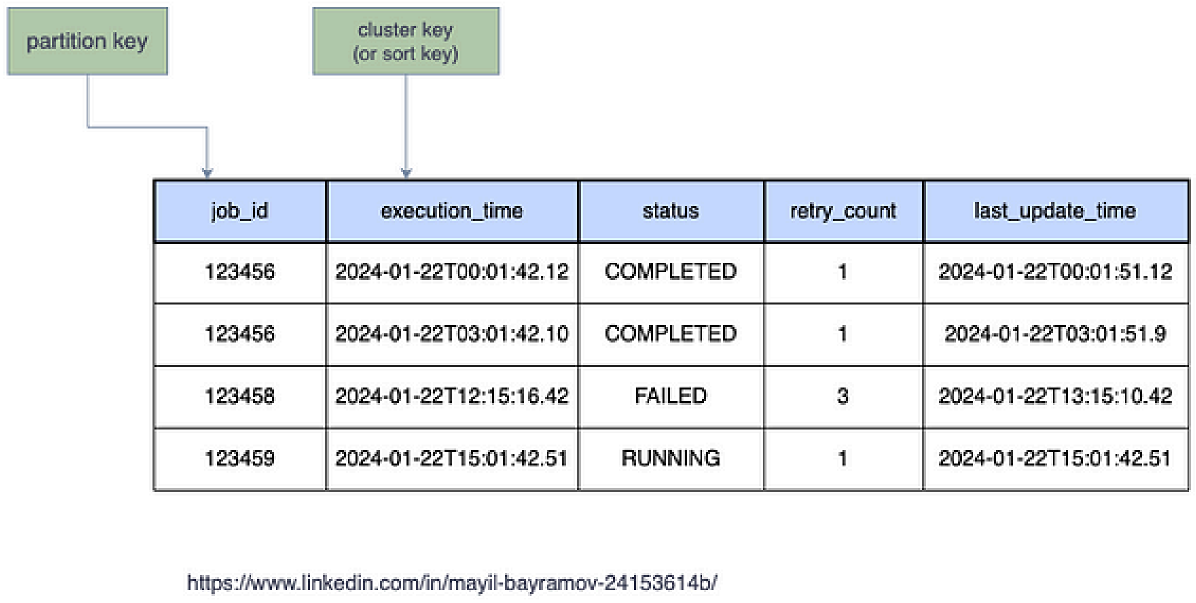
We converted `next_execution_time` to a UNIX timestamp with minute-level granularity and marked it as a *partition key* .`job_id` field also marked as s*ort key.* These changes will significantly enhance performance. With this modification, tasks scheduled to execute in the current minute will be located in the same partition, allowing the query to scan only one partition to retrieve the list of tasks to be executed

The diagram below provides a visual representation of how this table will be organized on the disk after implementing these changes.



https://www.linkedin.com/in/mayil-bayramov-24153614b/

Task_Execution_History table design: We need one more table to store task's execution whenever a scheduled task is executed. That table will also store the task's status, retry count and etc. As a result, users will have the capability to access the execution history of any job. We will access the job's execution history by `job_id` , therefore it is better to partition this table by `job_id` to be able to access all history for the same job in the same partition. Also, we will define `execution_time` as a sort key, to keep job history as sorted by execution time.


partition key
cluster key (or sort key)
https://www.linkedin.com/in/mayil-bayramov-24153614b/

job_id	execution_time	status	retry_count	last_update_time
123456	2024-01-22T00:01:42.12	COMPLETED	1	2024-01-22T00:01:51.12
123456	2024-01-22T03:01:42.10	COMPLETED	1	2024-01-22T03:01:51.9
123458	2024-01-22T12:15:16.42	FAILED	3	2024-01-22T13:15:10.42
123459	2024-01-22T15:01:42.51	RUNNING	1	2024-01-22T15:01:42.51

As we have chosen the optimal database and designed the database schema, now it is time to design services and interactions between them.

Interaction between user and job service (step 1,2)

Api-Design**:** When users create jobs on our web platform, the platform will invoke the **job_service** REST endpoint through a *load balancer*. In this context, the *load balancer* serves as a distribution mechanism, efficiently routing and balancing the incoming requests among multiple instances of the **job_service** for optimal performance and reliability. If we were to structure the design of that endpoint, the resulting snippet might look like the one provided below:

```
request-method: POST
endpoint-path: https://hostname/users/{user_id}/jobs
request-body:
{
   "executionInterval": "PT3H",
   "maxRetryCount": 3,
   "is_reccuring": true
}


response-status: HTTP/1.1 201 Created
response-body:
{
   "job_id": "123456"
}
```

Job-Service-Design: Our **job_service** accepts request body (diagram-1) will

- Generate a unique `id` for the job and proceed to store its details in the '`jobs`' table.
- Calculate `next_execution_time = (currentDateTime()+ executionInterval duration)` with UNIX timestamp *minute-level granularity*

```
//Java implementation would look like this
long nextExecutionTime = LocalDateTime.now()
                              .plus(Duration.parse(executionInterval))
                              .atZone(ZoneOffset.systemDefault())
                              .toInstant()
                              .getEpochSecond() / 60;
```

store `next_execution_time` and `job_id` in the **task_schedule** table.

Furthermore, deploy the **job_service** across multiple instances. This ensures distributed handling of user-related operations, contributing to parallel processing and scalability across the system.

Scheduling service(step3)

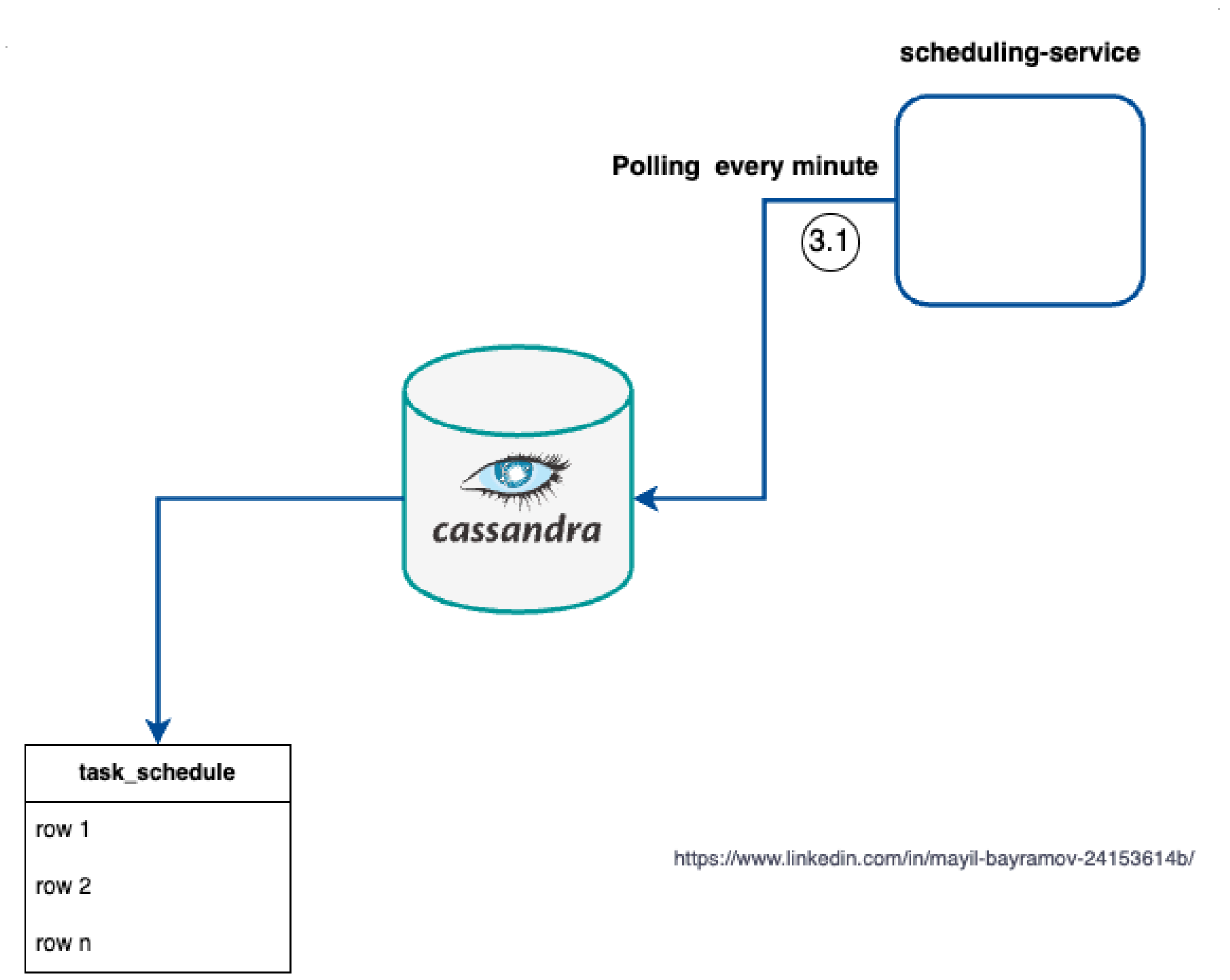
The s**cheduling_service (**diagram-1**)** is responsible for

3.1 - *polling* the `task_schedule` table every minute for *pending tasks*

3.2 - producing pending tasks to `task`topic in a queue for execution

3.3 - After the task is scheduled add one row to `task_execution_history` table with `status = SCHEDULED` . Check current job's recurring status in `job` table, if it `is_recurring` job, update current job's `next_execution_time` in `task_schedule` table.

A straightforward design for **process 3.1** would look like this:

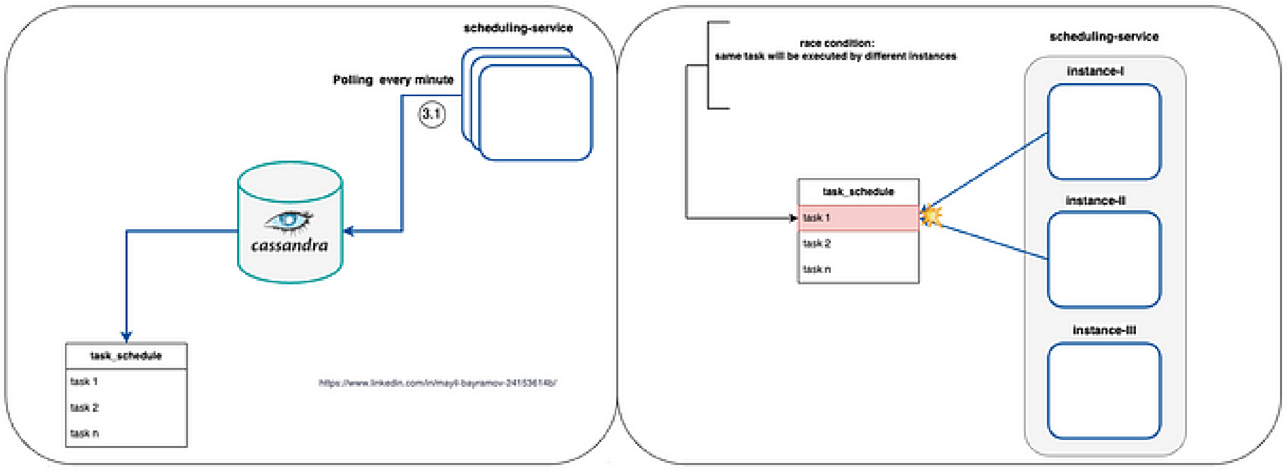


Query to `task_schedule` table would look like a snippet in the below:
```
select * from task_schedule where next_execution_time = 1705788060
```

At first glance in *diagram 8*, everything appears satisfactory. However, two nuances need addressing with this approach:

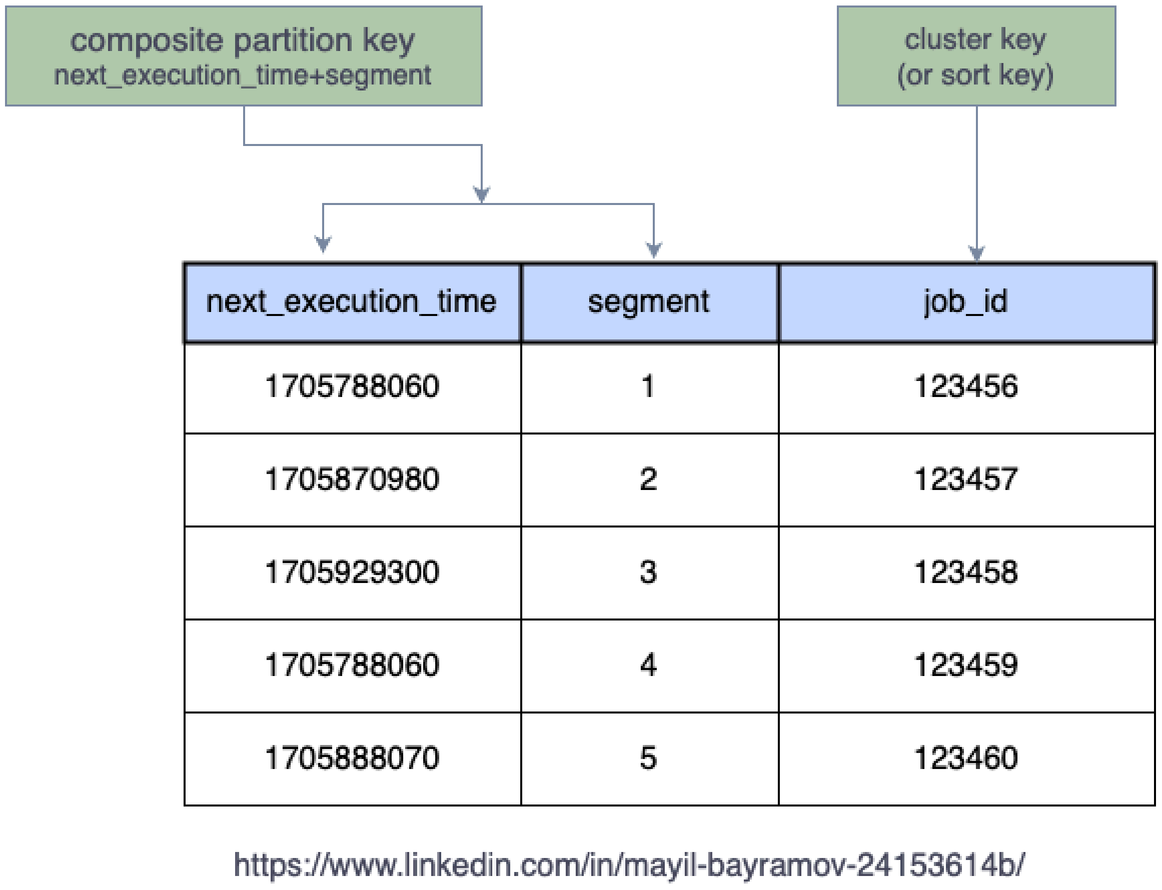
- As mentioned earlier, we are designing a system expected to handle an *average of 34 722 tasks per minute*. With only one ***scheduling-service*** instance, there is a concern about its ability to handle this load efficiently, especially during **peak hours** when the load may surge to *100 000 tasks per minute*. This could lead to potential issues such as long latency in task executions.
- The second nuance involves the **single point of failure** in *diagram 8*, where the scheduling service is susceptible. If the scheduling service experiences a failure, the entire system becomes unavailable.

To enhance this architecture, a proposed solution is to i*ncrease the number of instances for the scheduling service* . Let's implement this adjustment in **diagram 9** and conduct a thorough analysis of its impact.



In the above diagram (diagram 9), you can observe the increased number of instances for the scheduling service. However, this augmentation introduces the potential for a race condition. For instance, if two different instances attempt to poll and execute the same task simultaneously, the operation *may be executed twice*. This scenario poses a serious problem, especially for critical tasks such as sending notifications to customers, generating invoices or performing payment operations. The repercussions could include sending duplicate notifications, generating identical invoices for payment, or executing financial transactions twice. Such issues could have significant implications for a company, especially in the context of financial tasks.

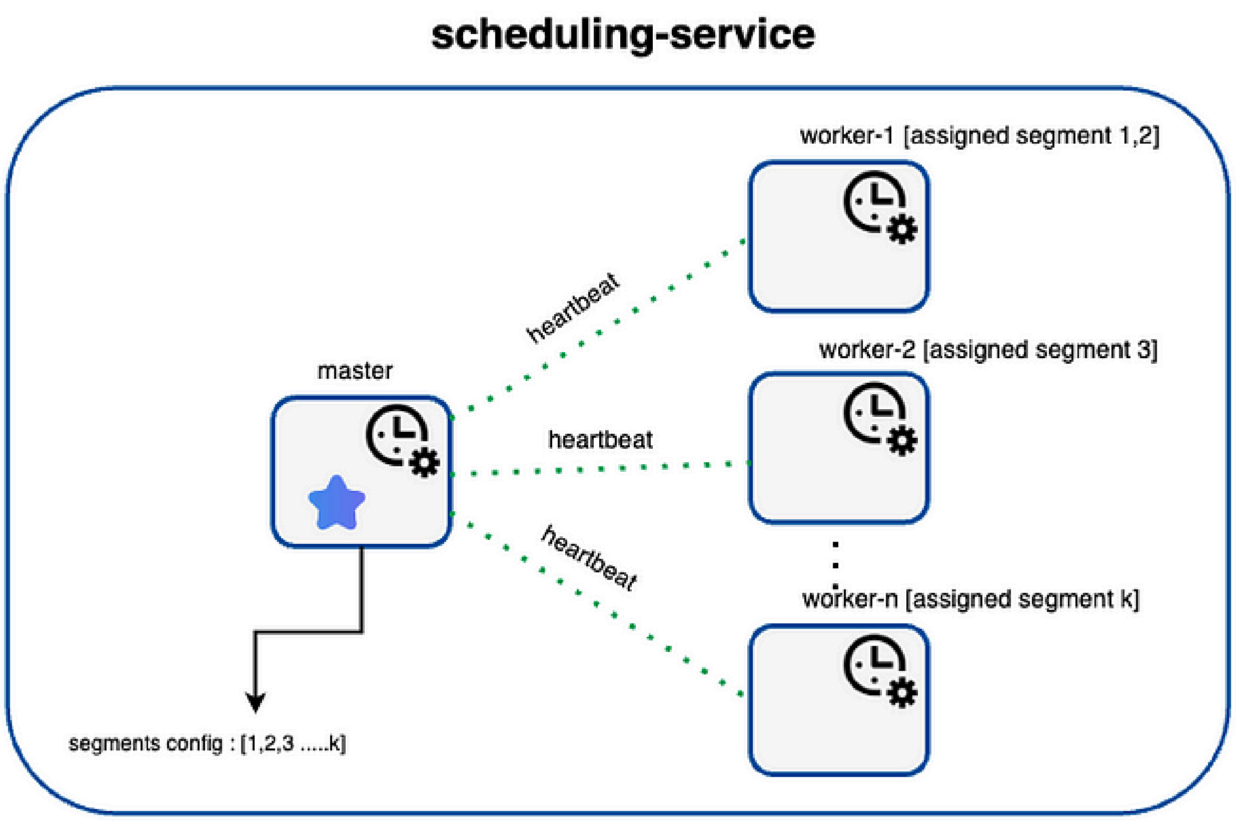
Schema changes in `task_schedule` table: Let's think about how we can distribute loads between these scheduling service instances. If we *have 8 instances of scheduling service* and 40 000 tasks for a given minute, how can we assign tasks to scheduling service instances based on this formula **tasks/number of instances => 5 000 tasks for each instance without leading racing condition?** I suggest altering `task_schedule` the table and adding one column named `segment` . When the application initiates, establish a *segment interval* [1,2,3….k] within the application configuration. For instance, let's define [1,2,3,4,5,6,7,8] for our design. Upon inserting a *new row* into the `job_schedule` table, randomly assign one of the segments from the specified interval (in our example, the interval is [1,2,3,4,5,6,7,8]) to that row.


composite partition key
next_execution_time+segment
cluster key
(or sort key)
https://www.linkedin.com/in/mayil-bayramov-24153614b/

next_execution_time	segment	job_id
1705788060	1	123456
1705870980	2	123457
1705929300	3	123458
1705788060	4	123459
1705888070	5	123460

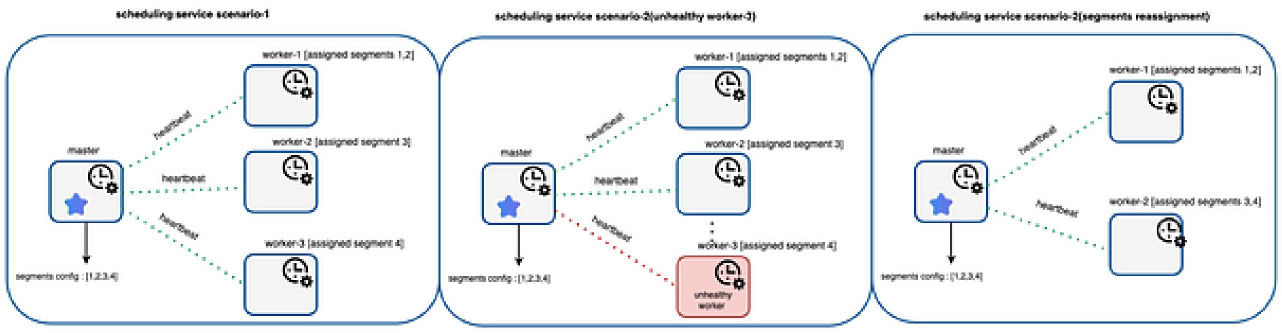
In the diagram above (**diagram 10**), note that we have also modified the partition key to a **composite partition key**, which is formed by the combination of `next_execution_time` and `segment`. This partitioning strategy guarantees that tasks scheduled for the *same minute and the same segmen*t will be stored in the *same partition* in the database.

Scheduling- service side we will have a *master(coordinator)* node which will be the *coordinator* for scheduling service instances and assigning segments [1,2,3 … k] to scheduling service worker instances when they bootstrapped. At regular intervals, the *master(coordinator)* sends a heartbeat request to the scheduling service worker instances to assess their **health**. If any scheduling service worker instance is detected as unhealthy (due to failure or termination), the master becomes aware of the situation. In response, the master dynamically reassigns segments [1,2,3 … k] to other available scheduling service instances, ensuring continuous operation even in the face of worker instance failures. **diagram 11** depicts a visual representation of this process.


scheduling-service
worker-1 [assigned segment 1,2]
heartbeat
worker-2 [assigned segment 3]
master
heartbeat
heartbeat
worker-n [assigned segment k]
segments config : [1,2,3 .....k]

In the diagram above (diagram 11), different segments are assigned to each worker. This assignment enables them to independently scan the `task_schedule` table efficiently without concerns about race conditions. For example for *worker-1* given query to `task_schedule` table would look like this.
`select * from task_schedule where next_execution_time = 1705788060 and segment in (1,2)`

Let's illustrate this architecture with a specific example. Consider a scenario where segments are defined as [1, 2, 3, 4], and the number of scheduling service worker instances is 3 (diagram-12 below)


scheduling service scenario-1
worker-1 [assigned segments 1,2]
worker-2 [assigned segment 3]
worker-3 [assigned segment 4]
master
heartbeat
heartbeat
heartbeat
segments config : [1,2,3,4]
scheduling service scenario-2(unhealthy worker-3)
worker-1 [assigned segments 1,2]
worker-2 [assigned segment 3]
worker-3 [assigned segment 4]
master
heartbeat
heartbeat
heartbeat
unhealthy worker
segments config : [1,2,3,4]
scheduling service scenario-2(segments reassignment)
worker-1 [assigned segments 1,2]
worker-2 [assigned segments 3,4]
master
heartbeat
heartbeat
segments config : [1,2,3,4]

(ps. If you're interested in delving deeper into the intricacies of the master->worker scenario, feel free to follow my channel. I'll be sharing another article soon, providing an expert-level explanation of the master->worker architecture.)

When defining the segment interval, consider both your anticipated workload and the number of running scheduling service instances. Ensure that the length of segments is greater than or equal to the number of worker instances using the formula: Length of segments ≥ Number of worker instances.

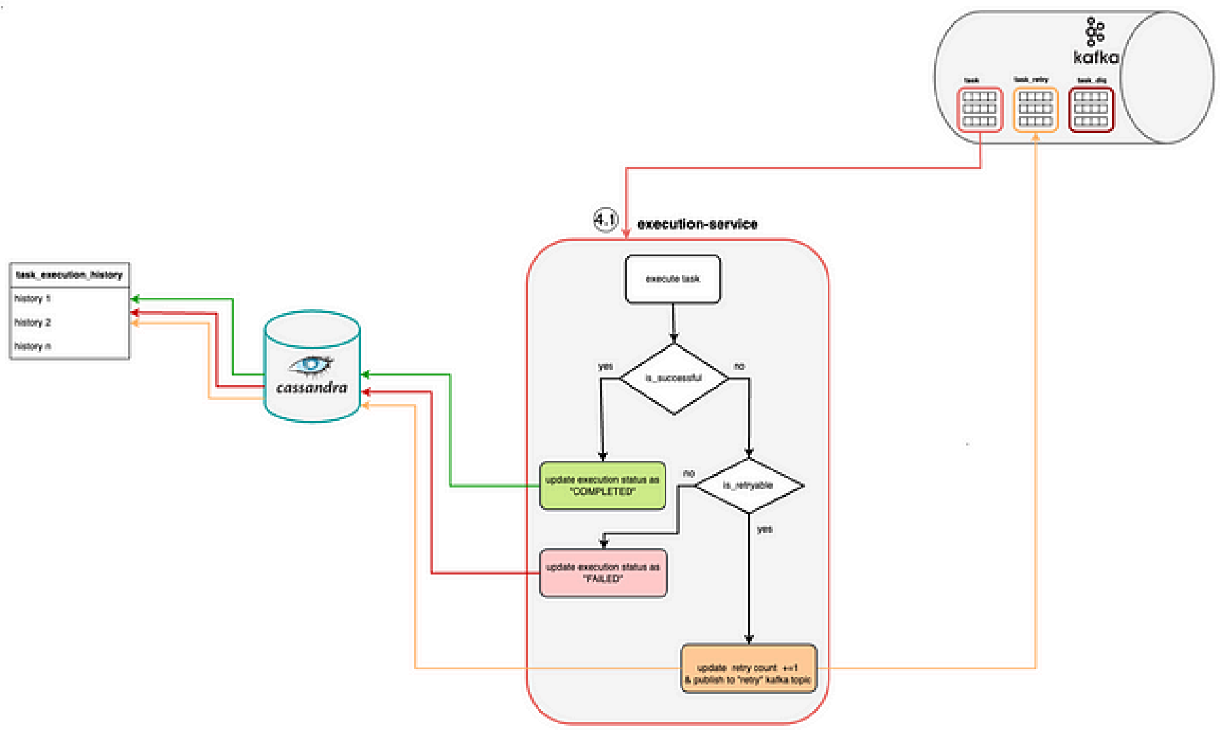
Proceed to follow **steps 3.2** and **3.3** for the remaining process as mentioned above, ensuring a comprehensive implementation of the described steps.

Execution service (step 4)

Execution service takes on the responsibility of subscribing to a Kafka topic named `task`. Its role involves consuming tasks from this topic and executing them as part of its functionality. As previously mentioned, *tasks in this context can encompass various executable actions*, such as payment processing, notification sending, invoice generation, and more. The nature of these tasks is adaptable based on the specific requirements of your business.

4.1 - Upon successful completion of task execution, the task execution service will update the execution status of the task as `COMPLETED` in the `task_execution_history` table.

In the event of a failure during the initial attempt at task execution, the execution service will examine the task's retry count in the `job` table. If the *retry count is greater than 0*, the execution service *will increment the task's retry count in the* `task_execution_history` *table*. Subsequently, the task will be added to the `task_retry` topic, initiating additional retry attempts. The process is visually represented in the diagram-11 below.



4.2-consumer of task retry topic: The consumer of the `task_retry` topic will retrieve data from the topic will delay in the specified interval (it can be fixed of incremental delay) then proceed to execute the task.

Upon successful completion of task execution, the task execution service will update the execution status of the task as `COMPLETED` in the `task_execution_history` table.

If the execution of the task fails again, its retryable status will be assessed. In the case of `is_retryable` the task's retry count will be incremented and the task will be produced to another retry topic for execution (for ex. `task_retry-2`). When encountering prolonged delays, it is advisable to utilize *multiple retry topics*. This practice helps prevent the retrying process in a single topic from blocking the execution of other tasks(If you need to also store retry attemps for each task execution , you can create extra table and store each attempt history in that table).

If the retry count has surpassed its limit, the task's execution status will be updated `FAILED` and the task will be produced to `task_dlq`topic. DLQ stands for **D**ead **L**etter **Q**ueue. Messages are produced to the DLQ when a message cannot be successfully processed or delivered after exhausting all retry attempts.

Thank you for taking the time to read my article! Follow & Subscribe for more articles