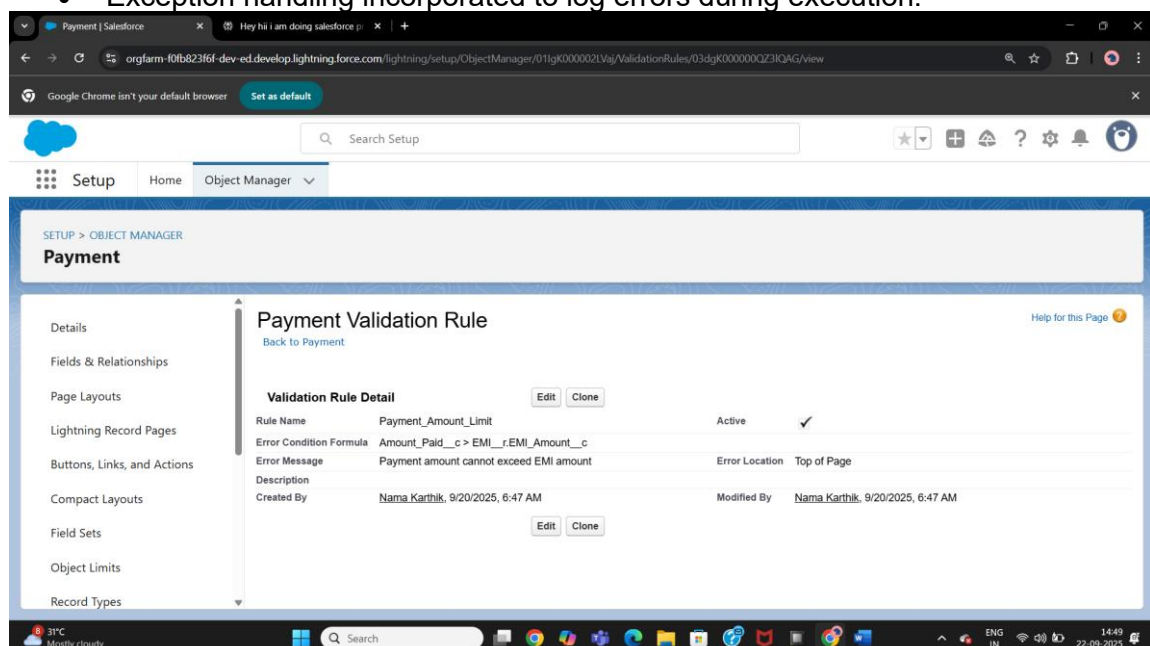


LoanEase: Salesforce Loan Management CRM

Phase 5: Apex Programming (Developer)

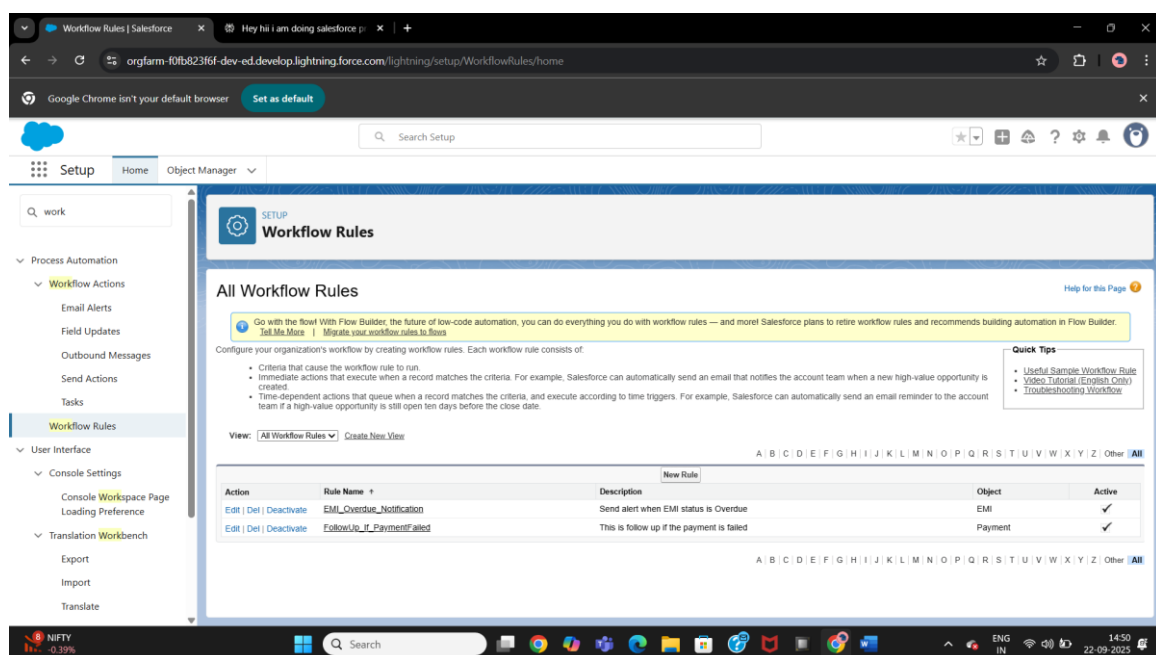
Classes & Objects

- **LoanManager (Service Class)**
 - Encapsulates business logic related to loan creation, update, and management.
 - Provides reusable methods for loan processing, ensuring separation of concerns.
 - Supports maintainability and scalability by isolating loan-related operations in a dedicated class.
- **EMIHandler (Trigger Handler Class)**
 - Manages trigger logic for the EMI object following best practices to keep triggers slim.
 - Handles bulk data efficiently and applies business rules related to EMI lifecycle events.
 - Acts as an abstraction layer between the database trigger and business logic.
- **EMIConstants**
 - A utility Apex class holding constant values for EMI status such as "Paid" and "Overdue".
 - Centralizes status literals to avoid hardcoding and ease maintenance across all Apex classes.
- **OverdueEMIBatch (Batch Apex Class)**
 - Implements Database.Batchable and Database.Stateful interfaces to asynchronously process large data sets.
 - Queries unpaid EMI records with due dates passed.
 - Marks these EMI records as "Overdue".
 - Includes try-catch blocks for exception handling to ensure robustness.
 - Contains start, execute, and finish methods fulfilling Batch Apex contract.
 - Provides debug logging on completion for traceability.
- **OverdueEMIQueueable (Queueable Apex Class)**
 - Implements Queueable interface to perform asynchronous job execution.
 - Updates EMI records overdue, similarly to the batch job but suitable for smaller jobs or chaining.
 - Exception handling incorporated to log errors during execution.



Apex Triggers

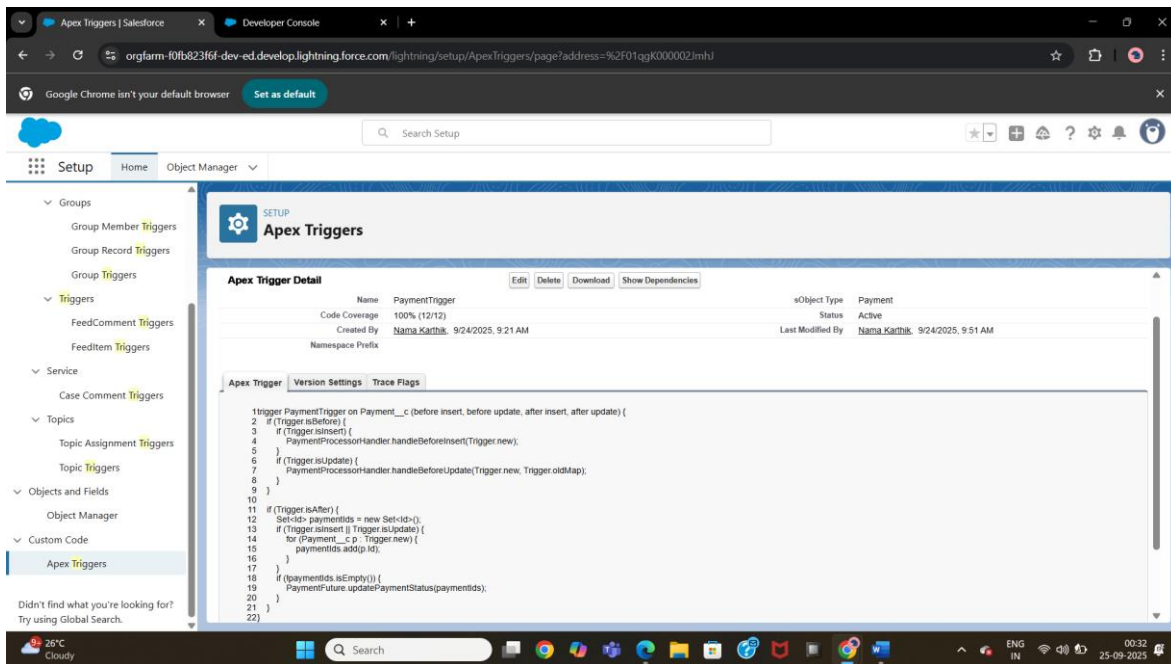
- **EMI Trigger(on EMI Object)**
 - Automates business logic for EMI records on events such as before insert, after insert, before update, and after update.
 - Delegates core logic to the EMHandler Apex class for maintainability and clean separation.
 - Bulk-safe: processes collections of EMI records efficiently.
 - Ensures validation and correct status updates for EMI lifecycle changes.
- **Payment Trigger(on Payment Object)**
 - Handles automated processes whenever Payment records are inserted, updated, or deleted.
 - Invokes business logic and validations for payments, such as enforcing payment limits and updating payment status.
 - Calls handler or service classes for main logic, following best practices.



Trigger Design Pattern

In this project, the Trigger Design Pattern was implemented to keep Apex triggers clean, maintainable, and efficient. The key characteristics of the pattern as applied are:

- **Single Trigger Per Object:** Each object has one trigger that handles all trigger events (insert, update, delete, etc.).
- **Delegation to Handler Classes:** Triggers serve only as routers that delegate processing to specialized handler Apex classes. Example:
 - EMITrigger delegates logic to EMHandler class.
 - PaymentTrigger calls into respective handler methods.
- **Context-Specific Handling:** The handler classes contain distinct methods for different trigger contexts like beforeInsert, afterUpdate, etc., enabling clear separation of logic.
- **Bulk Processing:** All logic in handlers supports bulk operations, processing lists of records to avoid governor limit exceptions.
- **Maintainability & Reusability:** By keeping logic away from triggers and inside classes, the codebase is modular, easier to test, debug, and extend.



SOQL & SOSL

- Salesforce Object Query Language (SOQL) was used extensively in Apex classes to retrieve records from Salesforce objects based on specific conditions.
- Typical SOQL queries were used to filter EMI records where status is not 'Paid' and due date has passed, aiming to batch update their status to 'Overdue'.
- SOQL was also used in queueable Apex to fetch the necessary records asynchronously.
- Sample SOQL query used in project Apex classes:

```
List<EMI__c> overdueEmis = [
    SELECT Id, EMI_Amount__c, EMI_Due_Date__c, Status__c
    FROM EMI__c
    WHERE Status__c != 'Paid' AND EMI_Due_Date__c < TODAY
];
```
- In this project, SOSL (Salesforce Object Search Language) was not required or used because the project focused on querying specific object records, not broad text search across multiple objects.

Collections: List, Set, Map

- **List**
 - Used extensively to hold collections of records retrieved by SOQL queries.
 - Example: List<EMI__c> to store EMI records fetched for processing in batch and queueable Apex classes.
 - Enables iterating over records for bulk-safe processing and DML operations.
- **Set**
 - Used to store unique identifiers or values to avoid duplicates.
 - For example, Set<Id> holding EMI record IDs to filter records in SOQL queries or to check
- **Map**
 - Used to associate record IDs with sObjects for fast lookups.
 - Example: Map<Id, EMI__c> built from querying EMI records or extracted from Trigger.newMap and Trigger.oldMap in triggers.
 - Supports comparing new and old state of records and implementing update logic efficiently.

The screenshot shows the Salesforce Developer Console with the 'OverdueEMIQueueable.apex' file open. The code defines a class that implements the Queueable interface. It contains a single 'execute' method that performs a SOQL query to find EMI records that are not paid and are due. It then iterates through the results and updates their status to 'OVERDUE'.

```

1 public class OverdueEMIQueueable implements Queueable {
2     public void execute(QueueableContext context) {
3         try {
4             List<EMI__c> overdueEmis = [
5                 SELECT Id, EMI_Amount__c, EMI_Due_Date__c, Status__c
6                 FROM EMI__c
7                 WHERE Status__c != :EMIConstants.STATUS_PAID
8                 AND EMI_Due_Date__c < TODAY
9             ];
10
11             for (EMI__c emi : overdueEmis) {
12                 emi.Status__c = EMIConstants.STATUS_OVERDUE;
13             }
14             if (!overdueEmis.isEmpty()) {

```

Control Statements

- If-Else Statements: Used extensively to enforce business rules and validations, such as checking status values before updating records or processing logic conditionally.
- For Loops: Iteration over collections (Lists of EMI or Payment records) for bulk processing, status updates, and aggregations.
- In batch Apex execute method, for loop iterates over EMI records to update status.
- If-else conditions verify payment statuses and amounts before performing updates.

The screenshot shows the Salesforce Developer Console with the 'PaymentProcessorHandler.apex' file open. The code defines a class with two static methods: 'handleBeforeInsert' and 'handleBeforeUpdate'. 'handleBeforeInsert' checks if the 'Amount_Paid__c' is null and sets it to 0. 'handleBeforeUpdate' checks if the 'Amount_Paid__c' is less than the previous value and adds an error message.

```

3 public static void handleBeforeInsert(List<Payment__c> newPayments) {
4     for(Payment__c payment : newPayments) {
5         if (payment.Amount_Paid__c == null) {
6             payment.Amount_Paid__c = 0;
7         }
8     }
9 }
10
11 public static void handleBeforeUpdate(List<Payment__c> newPayments, Map<Id, Payment__c> oldPaymentsMap) {
12     for(Payment__c payment : newPayments) {
13         Payment__c oldPayment = oldPaymentsMap.get(payment.Id);
14         if (payment.Amount_Paid__c < oldPayment.Amount_Paid__c) {
15             payment.addError('Amount cannot be decreased.');

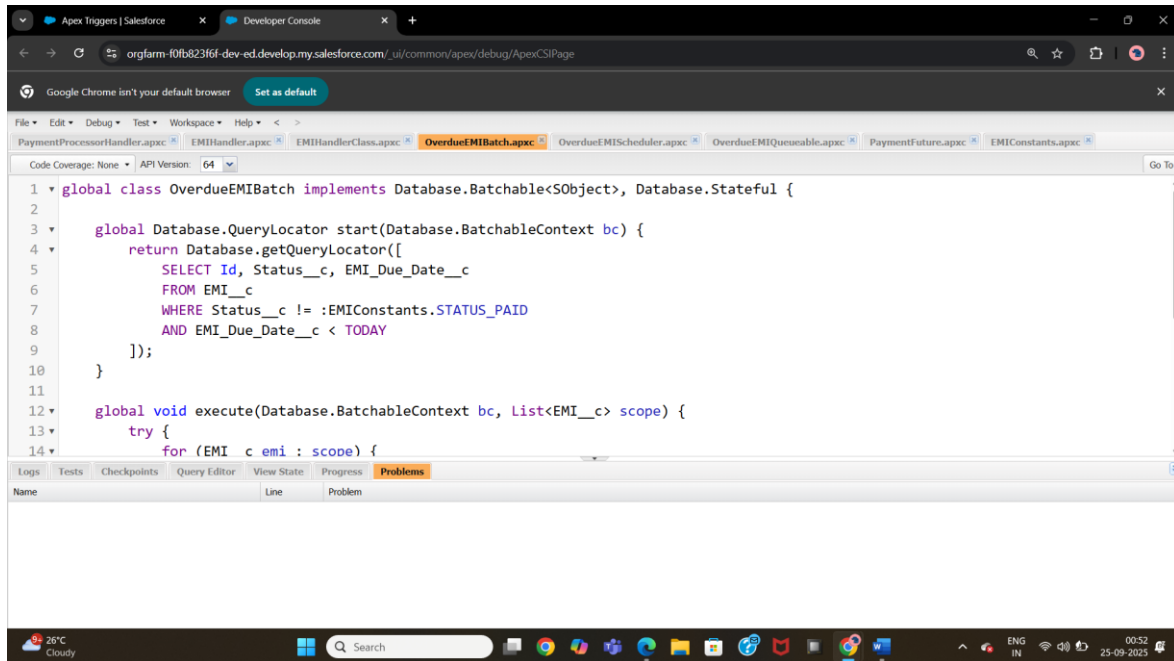
```

Batch Apex

- Developed the OverdueEMIBatch class implementing Database.Batchable<SObject> and Database.Stateful interfaces.
- The batch job queries EMI records where the status is not paid and the due date has passed.
- It processes these records in manageable chunks (batch size), updating their status to

"Overdue".

- Exception handling within the execute method ensures that any processing errors are caught and logged without aborting the entire batch.
- The finish method is used for post-processing or logging once the batch completes.
- Batch Apex facilitates asynchronous processing of large data sets efficiently without hitting platform limits.
- The batch can be scheduled or invoked programmatically through the Apex scheduler or executeBatch method.



The screenshot shows the Salesforce Developer Console with the 'OverdueEMIBatch.apex' file open. The code defines a global class 'OverdueEMIBatch' that implements 'Database.Batchable<SObject>' and 'Database.Stateful'. It includes a 'start' method that returns a 'Database.QueryLocator' with a SOQL query selecting 'Id, Status__c, EMI_Due_Date__c' from 'EMI__c' where 'Status__c' is not 'STATUS_PAID' and 'EMI_Due_Date__c' is less than 'TODAY'. It also includes an 'execute' method that iterates over a list of 'EMI__c' objects and processes them. The interface at the bottom shows tabs for 'Logs', 'Tests', 'Checkpoints', 'Query Editor', 'View State', 'Progress', and 'Problems'.

```
1 global class OverdueEMIBatch implements Database.Batchable<SObject>, Database.Stateful {
2
3     global Database.QueryLocator start(Database.BatchableContext bc) {
4         return Database.getQueryLocator([
5             SELECT Id, Status__c, EMI_Due_Date__c
6             FROM EMI__c
7             WHERE Status__c != :EMIConstants.STATUS_PAID
8             AND EMI_Due_Date__c < TODAY
9         ]);
10    }
11
12    global void execute(Database.BatchableContext bc, List<EMI__c> scope) {
13        try {
14            for (EMI__c emi : scope) {
```

Queueable Apex

- Developed the OverdueEMIQueueable class implementing the Queueable interface.
- This class performs asynchronous processing of overdue EMI records by querying and updating their status.
- Queueable Apex allows running jobs asynchronously with more flexibility than future methods.
- Supports passing complex objects and allows monitoring jobs via a job ID returned by System.enqueueJob().
- The execute method contains the job's logic and Salesforce runs it when system resources are available.
- Exception handling is implemented to catch and log errors gracefully during async execution.
- Queueable jobs can be chained for sequential asynchronous processing when needed.

Scheduled Apex

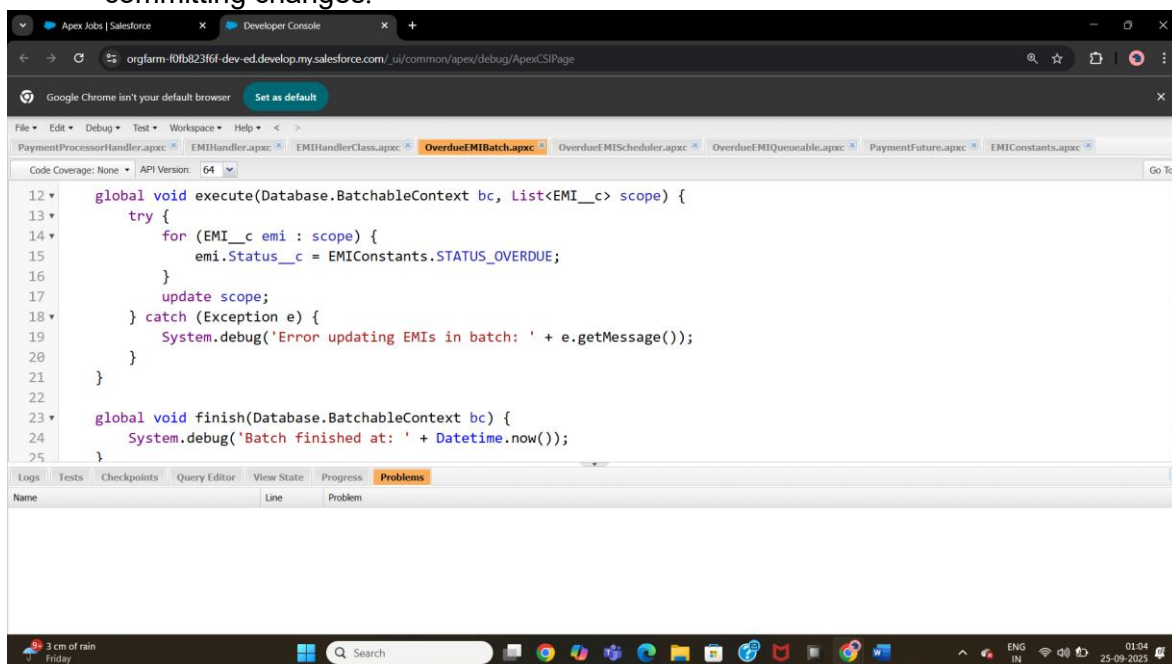
- Developed a scheduler class to run OverdueEMIBatch batch Apex job automatically on a defined schedule.
- The scheduler class implements the Schedulable interface and contains the execute() method where the batch job is invoked.
- The batch job updates EMI records overdue by setting their status to "Overdue".
- Schedule expression used runs the batch daily at midnight.
- Scheduling ensures regular maintenance of records without manual intervention.
- You can schedule the job in the Developer Console or Salesforce Setup using System.schedule().

Future Method

- Future methods were used to handle asynchronous processing needed for scenarios such as:
 - Performing resource-intensive calculations related to Payments or EMI processing without delaying the user transaction.
 - Calling external web services or APIs asynchronously after record inserts or updates.
 - Offloading non-critical processing logic from synchronous triggers or controller code, enhancing system responsiveness and user experience.
- Specifically, when payment or EMI records are processed, some calculations or notifications were deferred using future methods to ensure that primary operations complete quickly.
- This use of future methods guarantees better performance, throttle management, and adherence to Salesforce governor limits while maintaining data integrity and business logic correctness.

Exception Handling

- Exception handling was implemented extensively in asynchronous Apex classes including batch Apex (OverdueEMIBatch) and queueable Apex (OverdueEMIQueueable).
- Try-catch blocks surround critical DML operations and business logic to gracefully catch runtime exceptions.
- Caught exceptions are logged using System.debug for troubleshooting and monitoring.
- This prevents unhandled exceptions from aborting entire batch or queueable jobs, enabling partial completion even in error scenarios.
- Exception handling enhances the reliability, maintainability, and user trust of the system by ensuring issues are flagged without total failure.
- In triggers, validation exceptions are used to prevent invalid data updates before committing changes.



```
12 global void execute(Database.BatchableContext bc, List<EMI__c> scope) {
13     try {
14         for (EMI__c emi : scope) {
15             emi.Status__c = EMIConstants.STATUS_OVERDUE;
16         }
17         update scope;
18     } catch (Exception e) {
19         System.debug('Error updating EMIs in batch: ' + e.getMessage());
20     }
21 }
22
23 global void finish(Database.BatchableContext bc) {
24     System.debug('Batch finished at: ' + Datetime.now());
25 }
```


Asynchronous Processing

- Implemented multiple asynchronous Apex techniques including Batch Apex, Queueable Apex, Future Methods, and Scheduled Apex.
- Batch Apex: Used for processing large volumes of EMI records to update overdue statuses in manageable chunks.
- Queueable Apex: Applied for fine-grained asynchronous jobs that handle EMI updates, allowing chaining and complex job control.
- Future Methods: Utilized for offloading non-critical, time-consuming operations like notifications or external API calls to asynchronous execution.
- Scheduled Apex: Scheduled the batch job to run daily, ensuring periodic update of overdue EMI statuses without manual trigger.
- This multi-tier asynchronous strategy guarantees scalability, avoids hitting platform limits, and improves user responsiveness

The screenshot displays the Salesforce 'Apex Jobs' setup page. A sidebar on the left contains navigation links for 'Email', 'Custom Code', 'Apex Classes', 'Apex Settings', 'Apex Test Execution', 'Apex Test History', 'Apex Triggers', 'Environments', 'Jobs', 'Apex Flex Queue', and 'Apex Jobs'. The main content area is titled 'Apex Jobs' and includes a status bar indicating 'Percent of Asynchronous Apex Used: 0%'. Below this, a table lists the execution details of various Apex jobs.

Action	Submitted Date	Job Type	Status	Status Detail	Total Batches	Batches Processed	Failures	Submitted By	Completion Date	Apex Class	Apex Method	Apex Job ID
	9/24/2025, 11:10 AM	Scheduled Apex	Queued		0	0	0	Karthik_Nama		OverdueEMIScheduler		707gK00000Dw07q
	9/24/2025, 11:08 AM	Scheduled Apex	Aborted		0	0	0	Karthik_Nama		OverdueEMIScheduler		707gK00000DvjLN
	9/24/2025, 10:53 AM	Batch Apex	Completed		1	1	0	Karthik_Nama	9/24/2025, 10:53 AM	OverdueEMIScheduler		707gK00000DwSEW
	9/24/2025, 10:52 AM	Scheduled Apex	Aborted		0	0	0	Karthik_Nama		OverdueEMIScheduler		707gK00000Dw4N
	9/24/2025, 10:45 AM	Batch Apex	Completed		1	1	0	Karthik_Nama	9/24/2025, 10:45 AM	OverdueEMIScheduler		707gK00000DwvG
	9/24/2025, 9:40 AM	Queueable	Completed		0	0	0	Karthik_Nama	9/24/2025, 9:40 AM	OverdueEMIScheduler		707gK00000DvrvG
	9/24/2025, 9:33 AM	Scheduled Apex	Aborted		0	0	0	Karthik_Nama		OverdueEMIScheduler		707gK00000DvqEB