# Phase 5 – Apex Programming (Developer)

AI-Enhanced Human–AI Collaboration Platform — Comprehensive Developer Guide  
  
This expanded document includes detailed explanations, step-by-step instructions, and example code for Apex classes, triggers, SOQL/SOSL, collections, control statements, batch/queueable/scheduled Apex, future methods, exception handling, test classes, asynchronous processing, and deployment considerations for the project.

## 1. Classes & Objects

What: Apex classes hold reusable logic (handlers, services). Use classes to encapsulate business logic, enable testability, and keep triggers thin. Create classes for services such as Task creation, AI processing, notification delivery, and utilities.

Steps:

a) In Setup → Apex Classes → New, create a class file. Prefer small, focused classes.

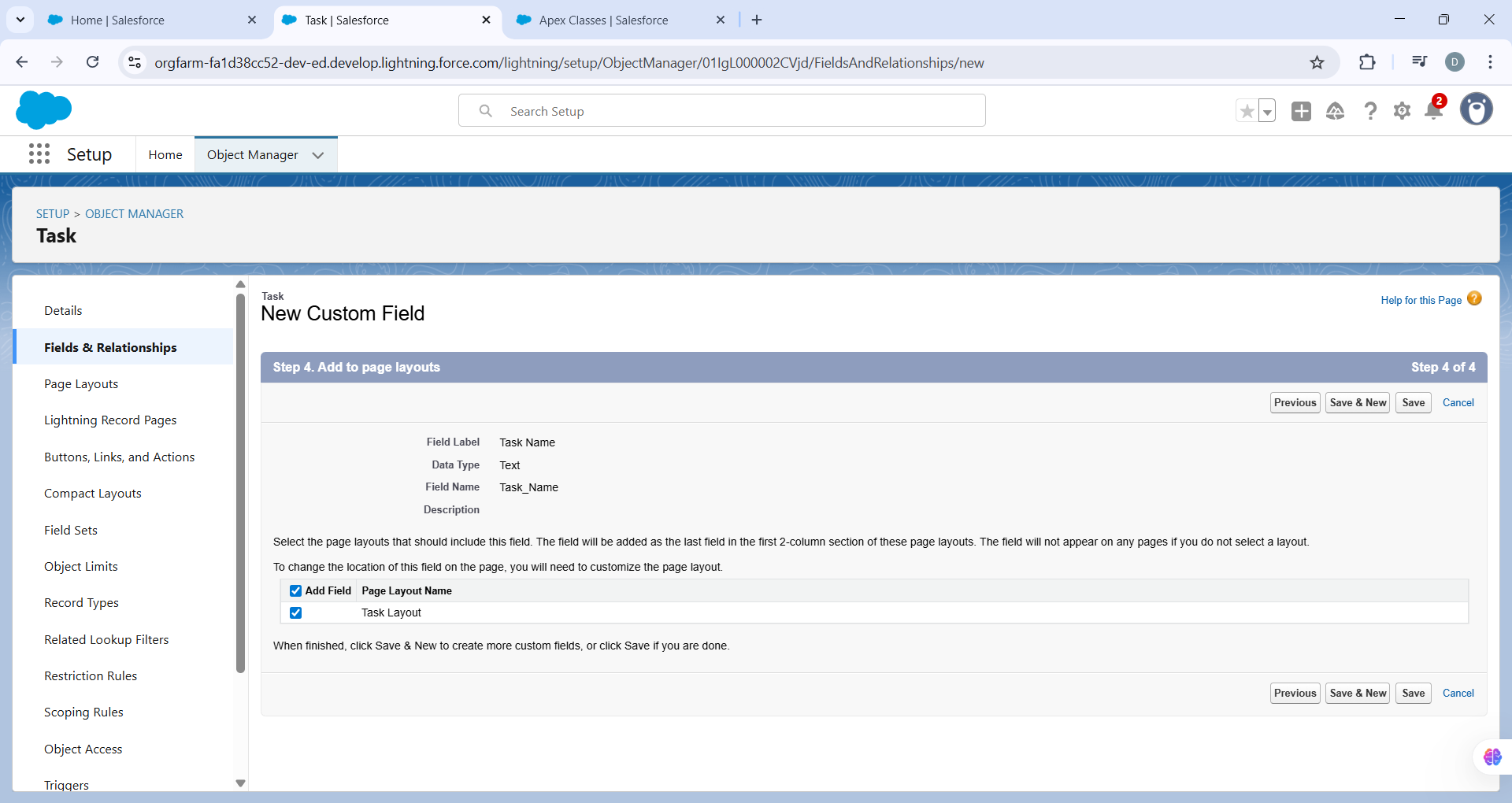
b) Keep methods public or global only when needed. Use 'with sharing' or 'without sharing' intentionally.

Best practice: Put DML and SOQL in service classes (not in triggers) and write unit tests for each class.

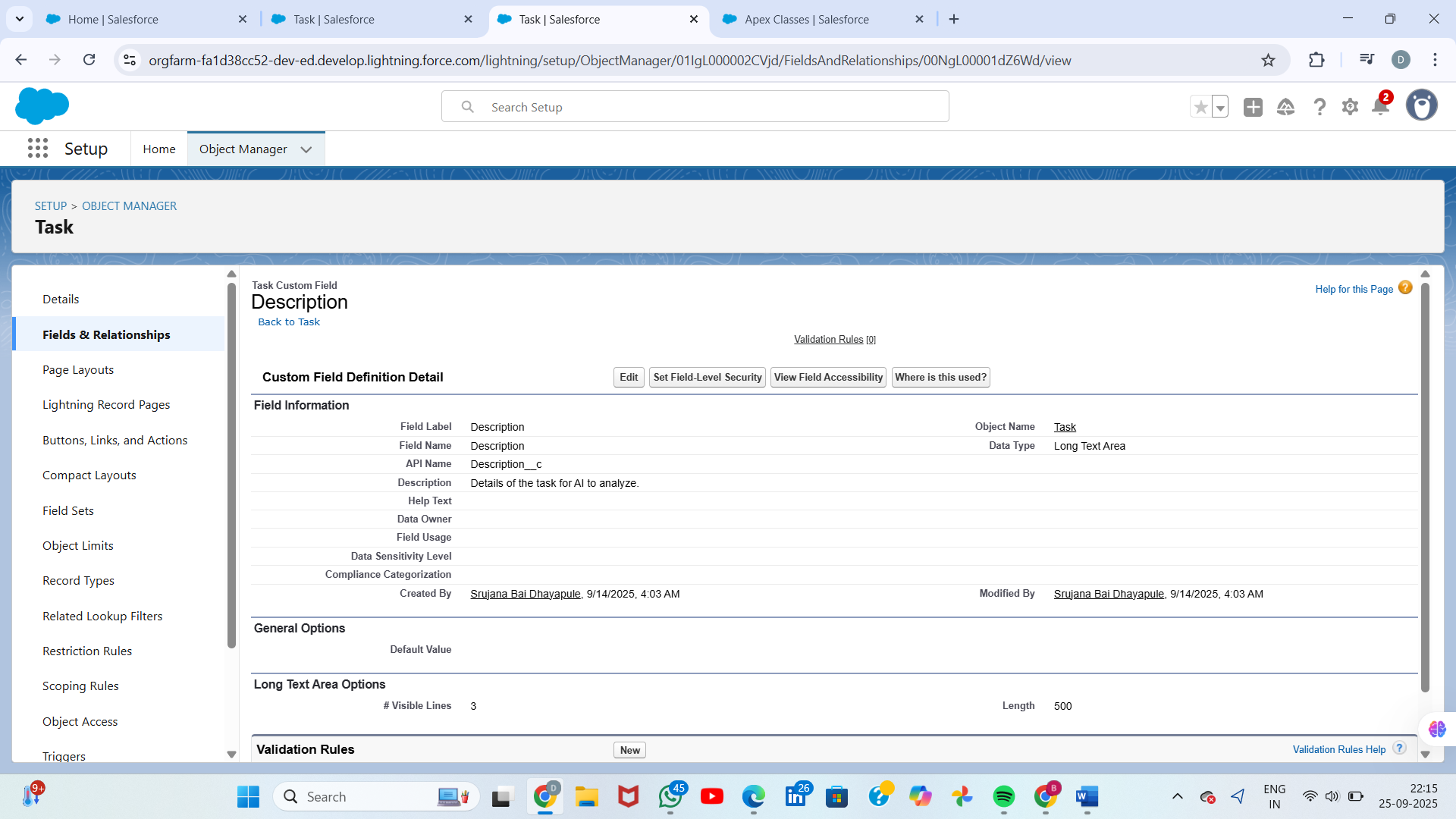
Example: TaskCreatorService (builds Task\_\_c records)

public with sharing class TaskCreatorService {  
 public static Task\_\_c buildReviewTask(Id aiSuggestionId, Id feedbackId, Decimal confidence, Id assignedUserId) {  
 Task\_\_c t = new Task\_\_c();  
 t.Task\_Name\_\_c = 'Review Approved Feedback';  
 t.Description\_\_c = 'Feedback ' + (feedbackId==null? '' : feedbackId) + ' approved. Suggestion: ' + String.valueOf(aiSuggestionId) + '. Confidence: ' + String.valueOf(confidence) + '%';  
 t.Status\_\_c = 'New';  
 t.Assigned\_User\_\_c = assignedUserId;  
 t.Related\_AI\_Suggestion\_\_c = aiSuggestionId;  
 return t;  
 }  
  
 public static void insertTasks(List<Task\_\_c> tasks) {  
 if (tasks == null || tasks.isEmpty()) return;  
 try {  
 insert tasks;  
 } catch (DmlException e) {  
 // handle or log error  
 System.debug('Error inserting tasks: ' + e.getMessage());  
 throw e; // rethrow or implement custom handling  
 }  
 }  
}  
**Task\_Name\_\_c:**

* **Data Type:** This should be a **Text** field.
* **Field Label:** Task Name
* **Field Name:** Task\_Name (the system will automatically add \_\_c)

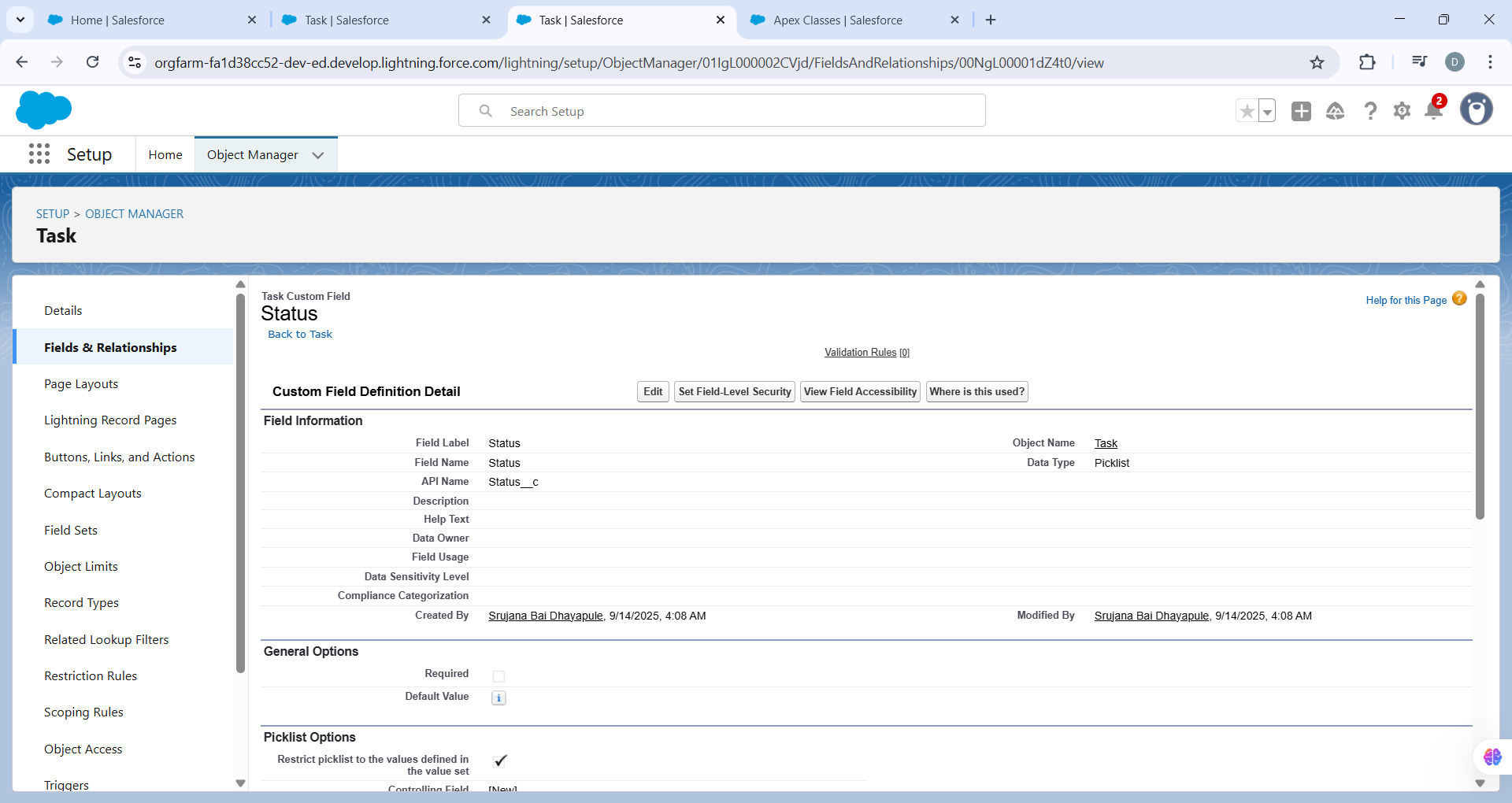
  
**Description\_\_c:**

* **Data Type:** This should be a **Text Area (Long)** field to hold more text.
* **Field Label:** Description
* **Field Name:** Description



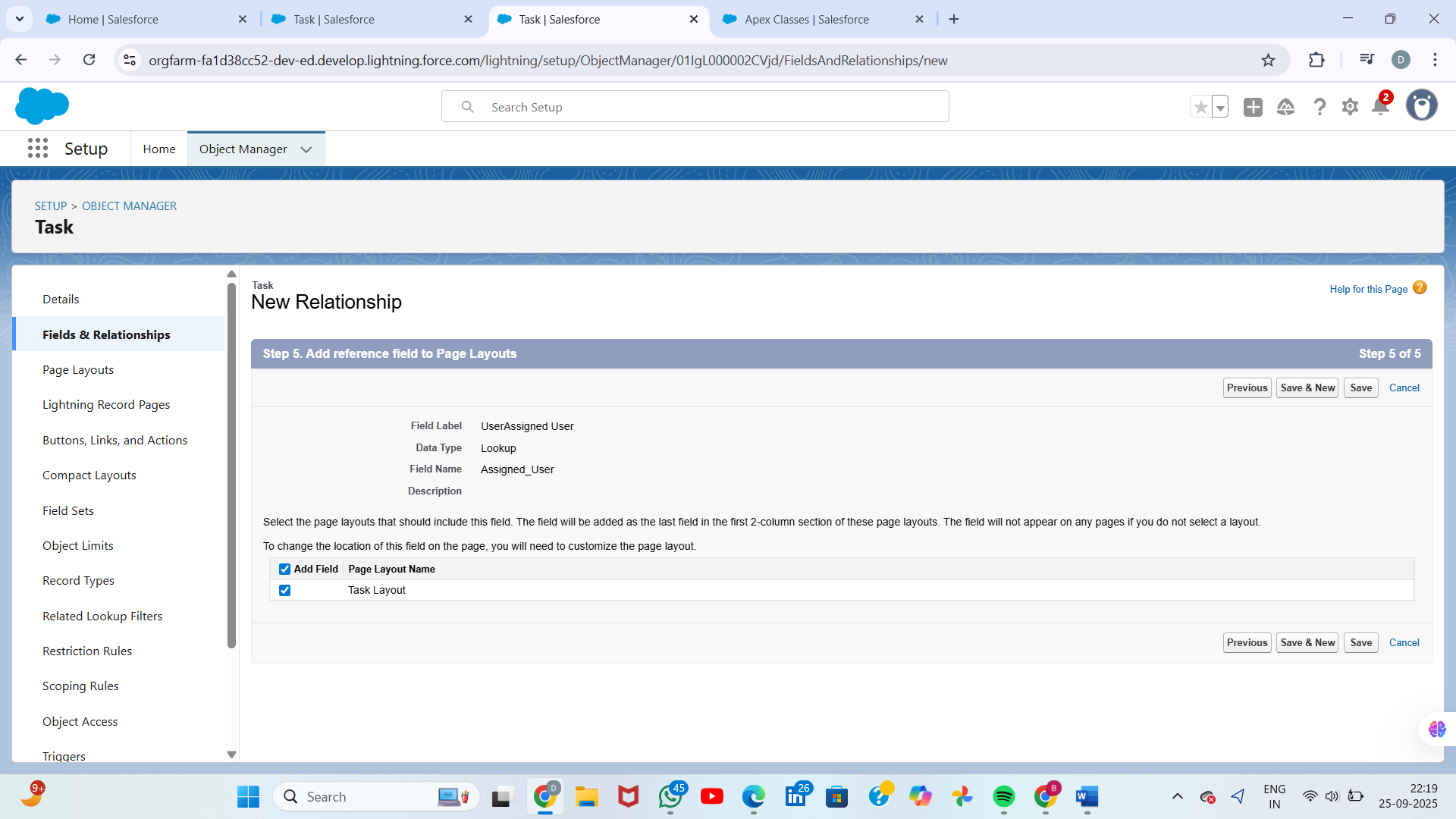
**Status\_\_c:**

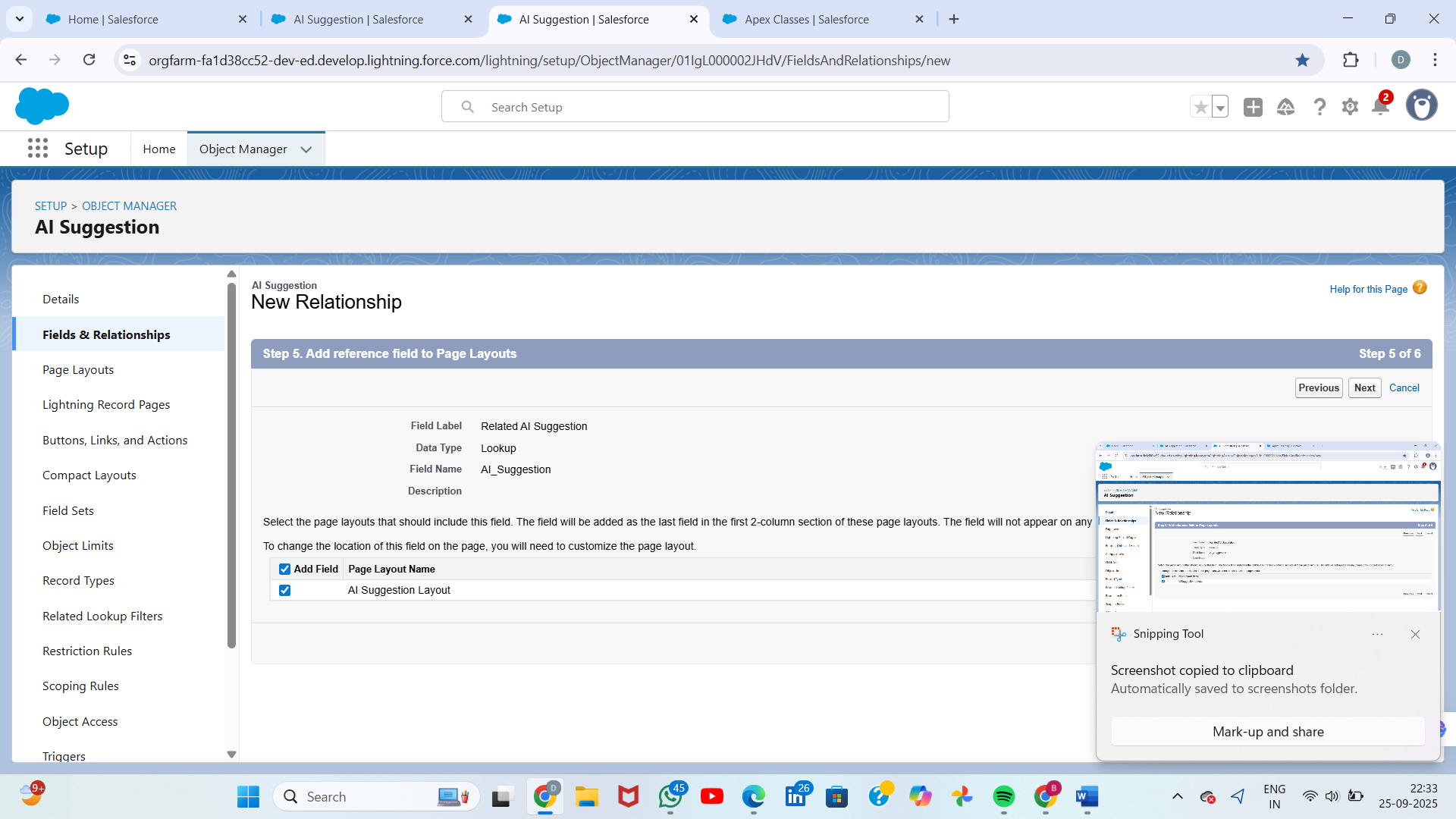
* **Data Type:** This is likely a **Picklist** field with values like 'New', 'In Progress', 'Completed', etc.
* **Field Label:** Status
* **Field Name:** Status



**Assigned\_User\_\_c:**

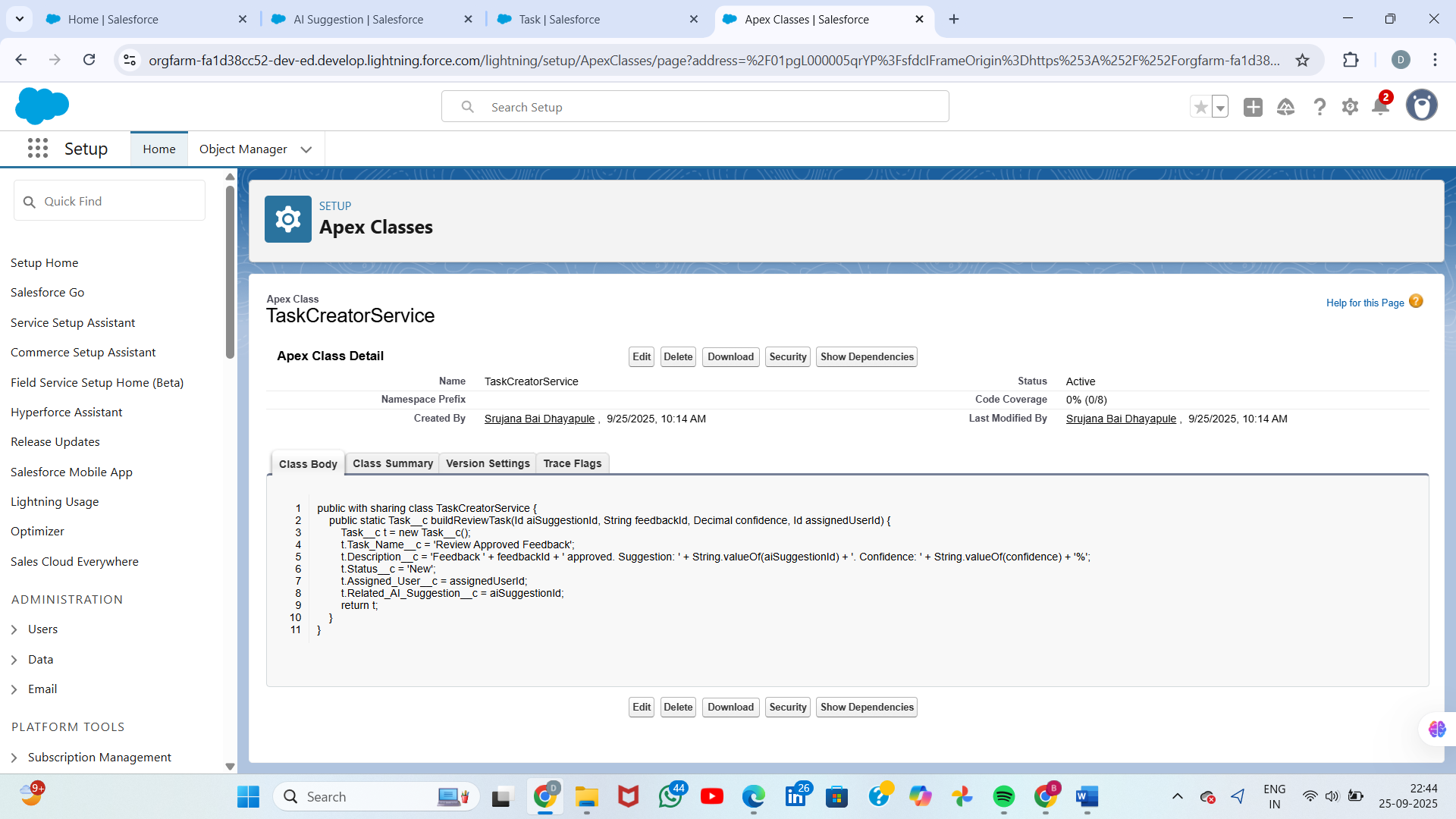
* **Data Type:** This should be a **Lookup Relationship** to the **User** object.
* **Related to:** User
* **Field Label:** Assigned User
* **Field Name:** Assigned\_User

  
**Related\_AI\_Suggestion\_\_c:**

* **Data Type:** This should be a **Lookup Relationship** to the AI\_Suggestion\_\_c object (or whatever your suggestion object is named).
* **Related to:** AI Suggestion (or the name of your object)
* **Field Label:** Related AI Suggestion
* **Field Name:** Related\_AI\_Suggestionc 

### Steps

1. Setup → Apex Classes → New.
2. Create service/handler class file names: e.g., AIProcessingService, TaskCreatorService, FeedbackHandler.
3. Keep methods public static for trigger handlers or instance methods for complex services.



## 2. Apex Triggers

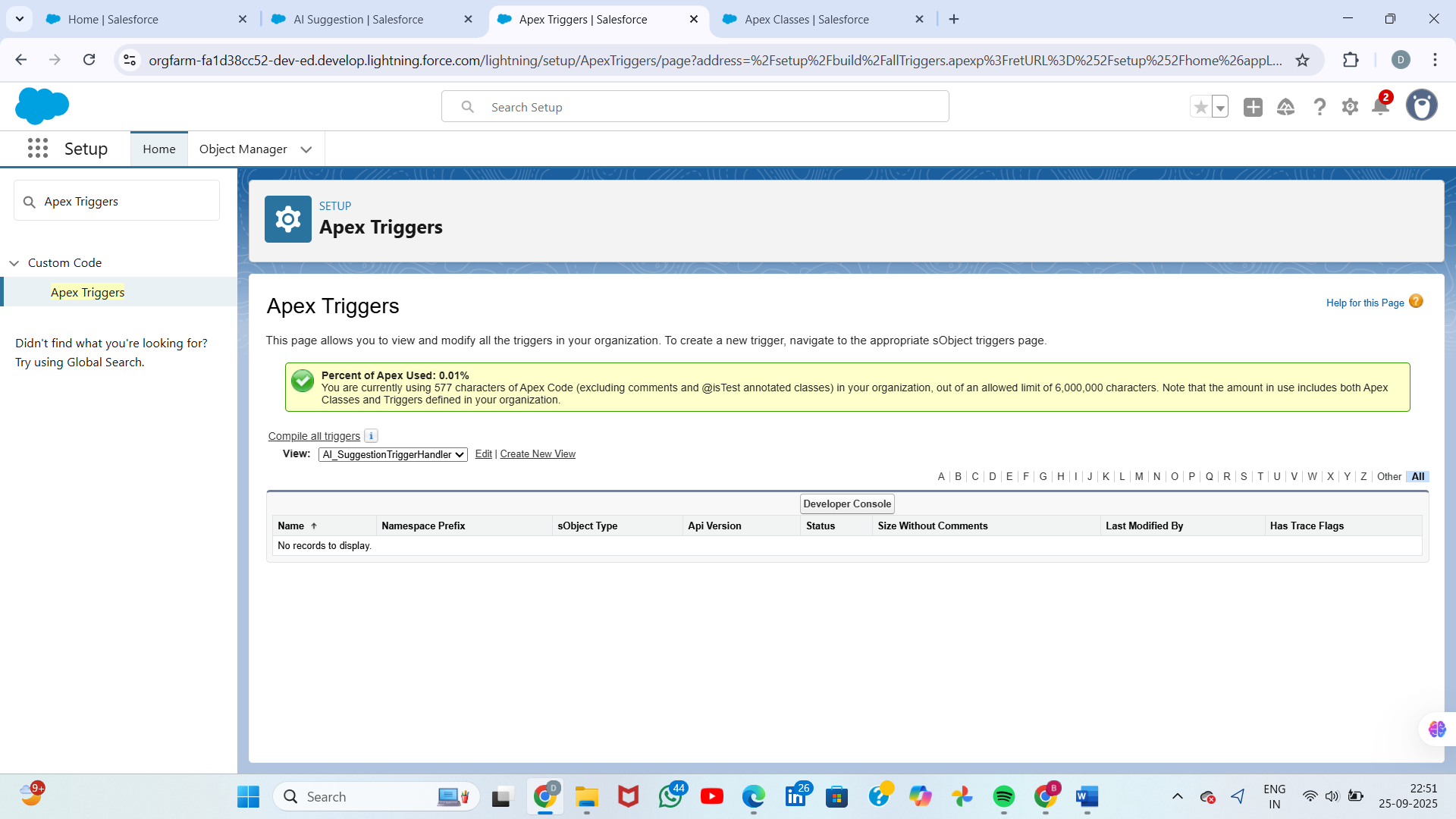
What: Triggers execute Apex before/after insert/update/delete on SObjects. Use triggers to respond to record lifecycle events.

Why: In this project, triggers can react to feedback creation (to create tasks), suggestion updates (to auto-create feedback), or to flag records for review. Always keep triggers thin and delegate logic to handler classes.

Trigger Example and Handler Pattern (after update on AI\_Suggestion\_\_c):

trigger AI\_SuggestionTrigger on AI\_Suggestion\_\_c (after update) {  
 if (Trigger.isAfter && Trigger.isUpdate) {  
 AI\_SuggestionTriggerHandler.afterUpdate(Trigger.new, Trigger.oldMap);  
 }  
}

Handler class example (bulkified):

public with sharing class AI\_SuggestionTriggerHandler {  
 public static void afterUpdate(List<AI\_Suggestion\_\_c> newList, Map<Id, AI\_Suggestion\_\_c> oldMap) {  
 List<Feedback\_\_c> toInsert = new List<Feedback\_\_c>();  
 for (AI\_Suggestion\_\_c s : newList) {  
 AI\_Suggestion\_\_c old = oldMap.get(s.Id);  
 if (s.Used\_\_c == true && (old == null || old.Used\_\_c != true)) {  
 Feedback\_\_c fb = new Feedback\_\_c();  
 fb.Related\_Suggestion\_\_c = s.Id;  
 fb.Approved\_\_c = false;  
 fb.Comments\_\_c = 'Auto-created feedback when suggestion marked used';  
 toInsert.add(fb);  
 }  
 }  
 if (!toInsert.isEmpty()) {  
 insert toInsert;  
 }  
 }  
}  


Notes: Always bulkify. Use maps to avoid N+1 queries. Avoid DML or SOQL inside for-loops. Add fault handling for DML operations.  **Navigate to Object Manager:** From Setup, go to **Object Manager**.

 **Select Your Object:** Find and click on the **Feedback\_\_c** custom object.

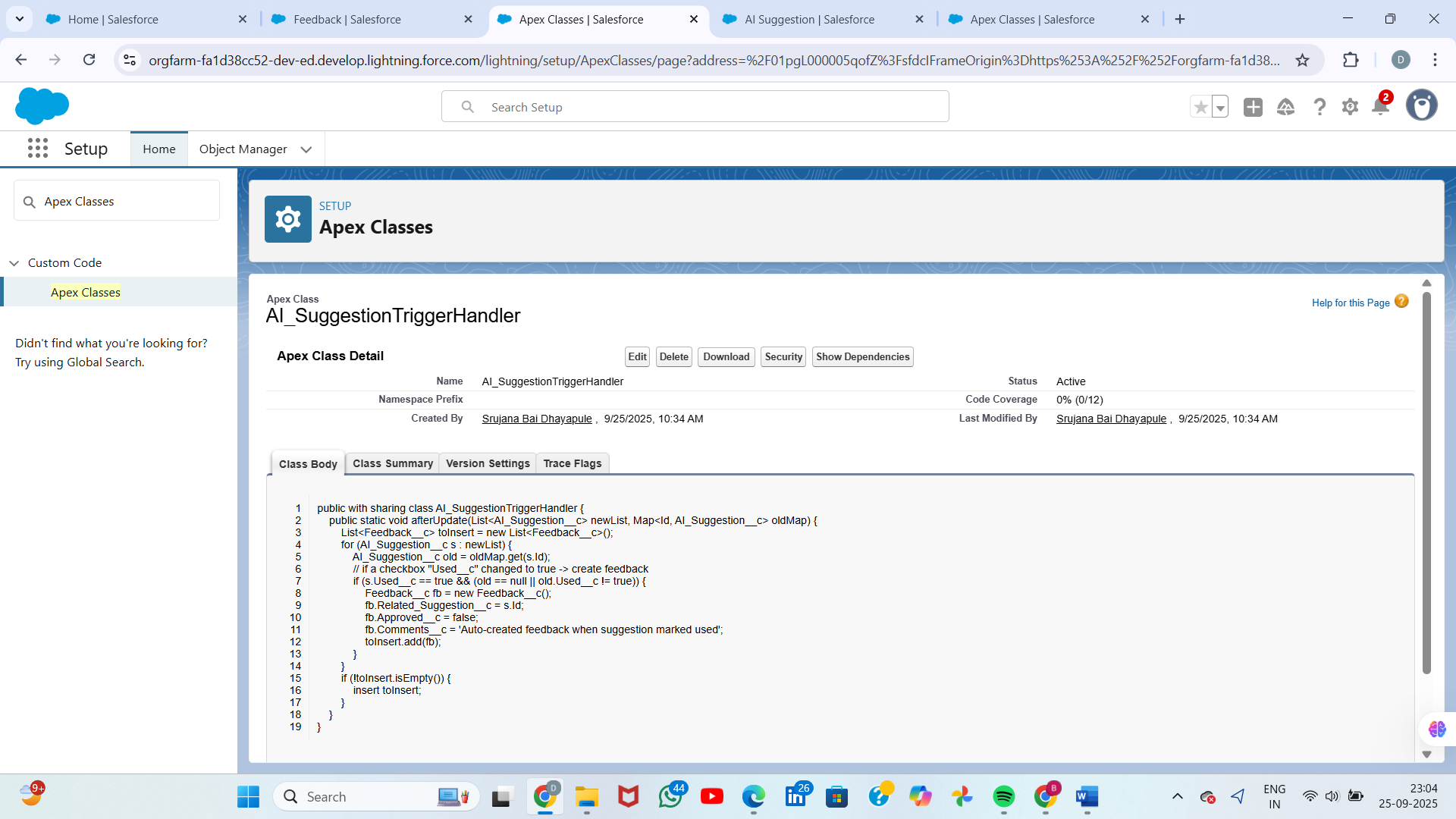
 **Create New Field:** Go to **Fields & Relationships** and click the **New** button.

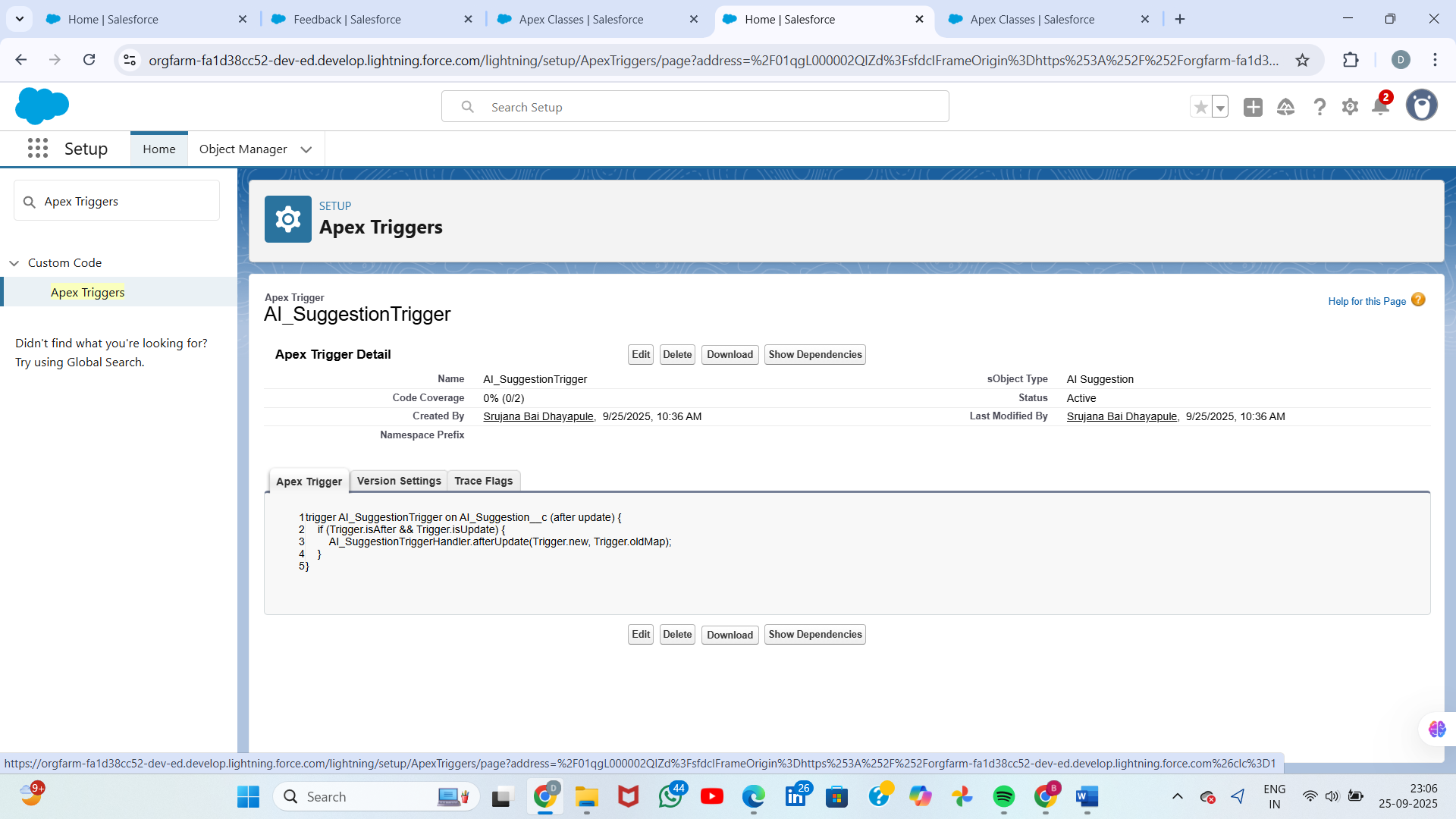
 **Choose Field Type:** Select **Lookup Relationship** as the data type and click Next.

 **Relate to AI\_Suggestion\_\_c:** From the dropdown menu, select **AI Suggestion** (or whatever you named the object) as the object you want to relate to. Click Next.

 **Name the Field:**

* For **Field Label**, enter Related Suggestion.
* The **Field Name** will automatically be populated as Related\_Suggestion. Ensure it ends with \_\_c.





## 3. Trigger Design Pattern

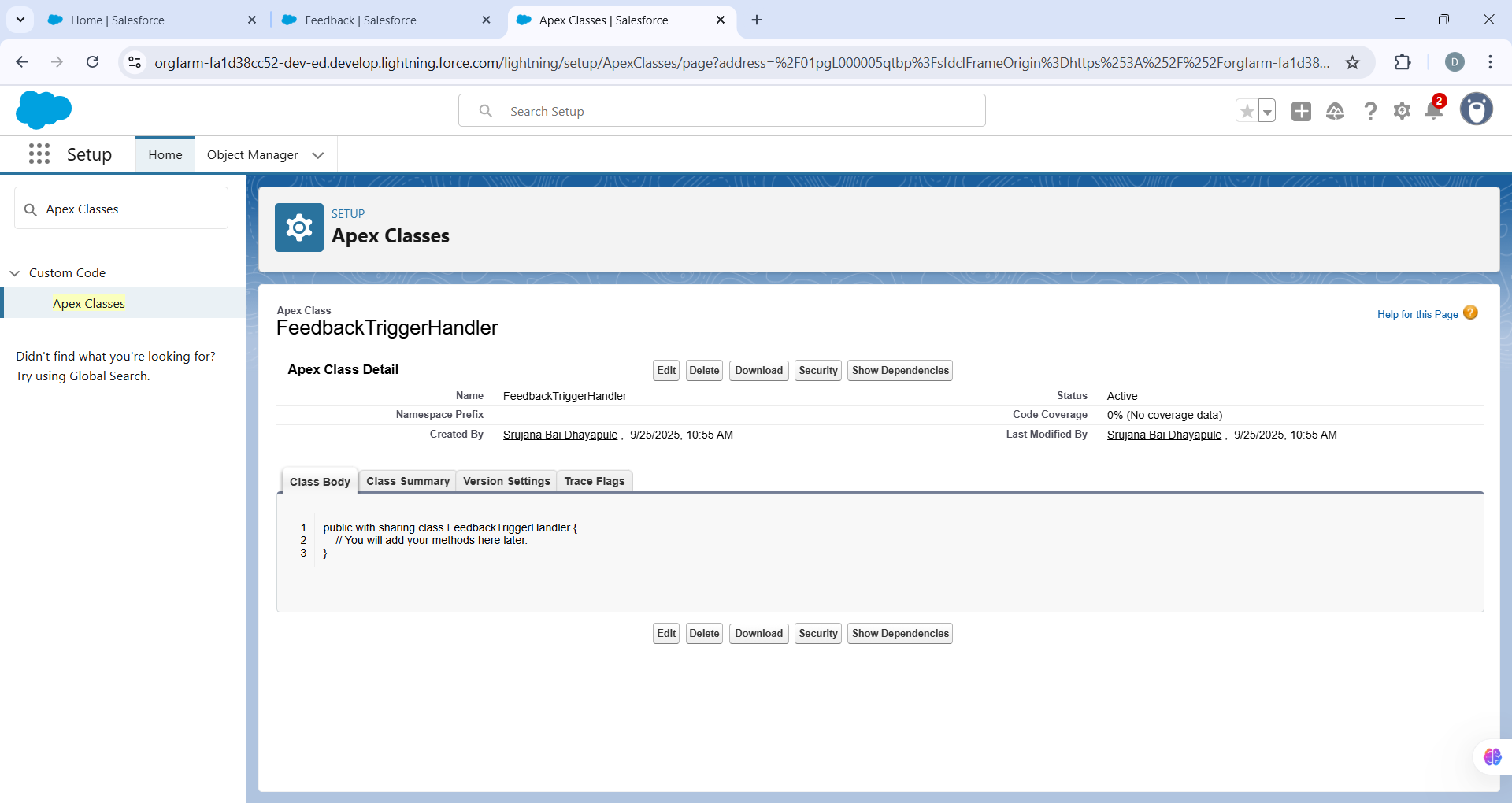
Purpose: Organize triggers so they are maintainable and testable. Use one trigger per object and delegate logic to handler classes. This simplifies unit testing and promotes reuse.

Typical structure:

- Trigger (object): calls methods on a Handler class depending on context (before/after, insert/update/delete).

- Handler class: contains static methods for each context (beforeInsert, afterInsert, beforeUpdate, afterUpdate...).

- Service classes: encapsulate heavy logic, DML, callouts, and helper utilities.



Sample Trigger skeleton:

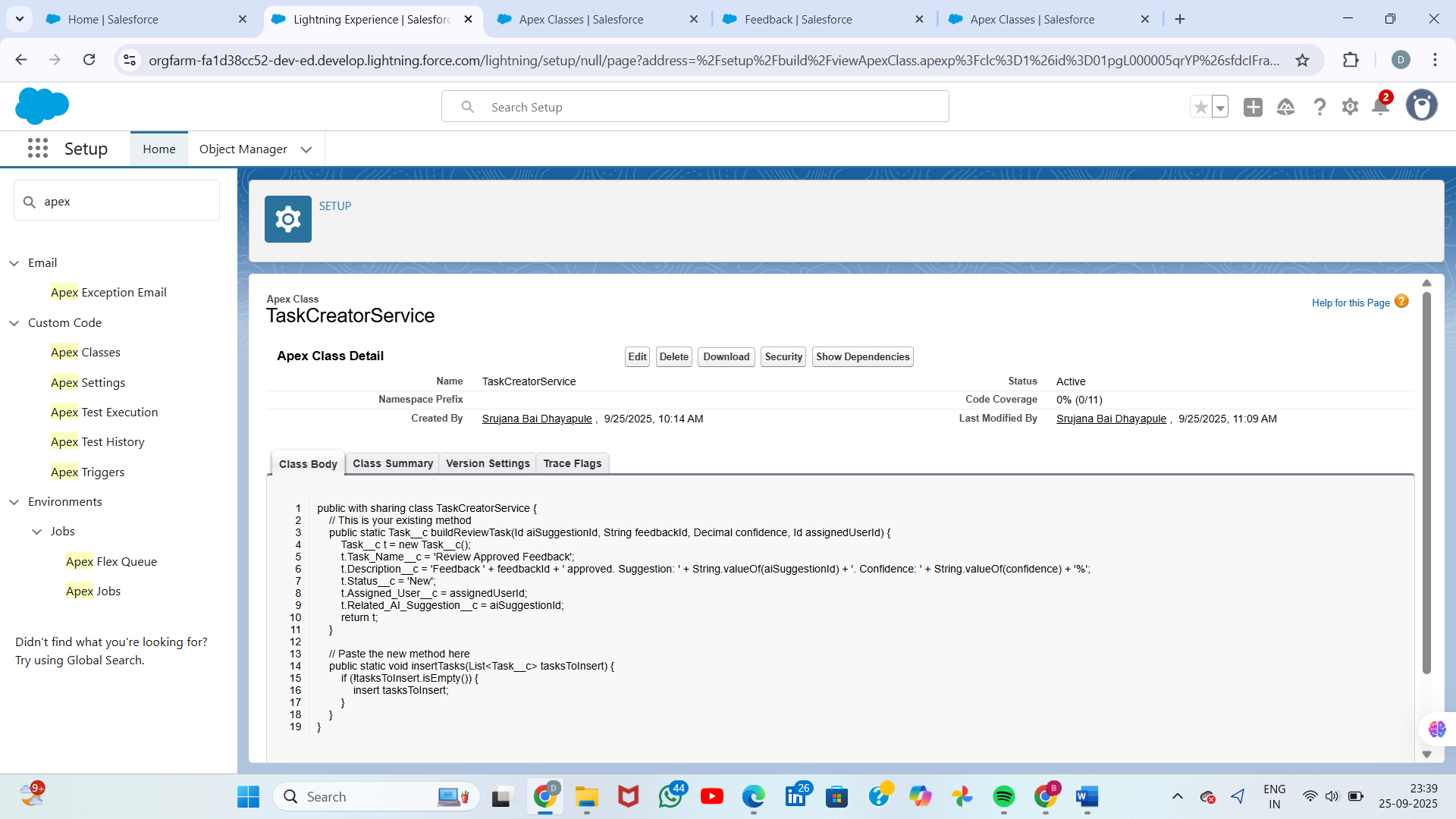
trigger FeedbackTrigger on Feedback\_\_c (before insert, after insert, before update, after update) {  
 if (Trigger.isBefore && Trigger.isInsert) {  
 FeedbackTriggerHandler.beforeInsert(Trigger.new);  
 }  
 if (Trigger.isAfter && Trigger.isInsert) {  
 FeedbackTriggerHandler.afterInsert(Trigger.new);  
 }  
 if (Trigger.isBefore && Trigger.isUpdate) {  
 FeedbackTriggerHandler.beforeUpdate(Trigger.new, Trigger.oldMap);  
 }  
 if (Trigger.isAfter && Trigger.isUpdate) {  
 FeedbackTriggerHandler.afterUpdate(Trigger.new, Trigger.oldMap);  
 }  
}

Best practices:

- Keep business logic out of triggers.   
 - Use a Trigger Handler class with context-specific methods.   
 - Provide 'isRunning' flags in handler to prevent recursion.   
 - Use custom settings/Custom Metadata to store configurable thresholds (e.g., confidence score).  
**Create the Handler class**

**Path (Setup)**

* Setup → Quick Find → **Apex Classes** → **New** → paste the handler code → Save.



## 4. SOQL & SOSL

SOQL (Salesforce Object Query Language) is used to query Salesforce data. SOSL searches across multiple objects and fields. Use indexes and selective filters for performance.

Common SOQL patterns:

- Simple query: [SELECT Id, Name FROM AI\_Suggestion\_\_c WHERE Confidence\_Score\_\_c < 70]

- Relationship query (parent-to-child): [SELECT Id, Name, (SELECT Id, Comments\_\_c FROM Feedbacks\_\_r) FROM AI\_Suggestion\_\_c WHERE Id IN :ids]

- Relationship query (child-to-parent): [SELECT Id, Suggested\_Text\_\_c, Related\_Task\_\_c, Related\_Task\_\_r.Task\_Name\_\_c FROM AI\_Suggestion\_\_c WHERE Id = :sId]

- Aggregate query: [SELECT Related\_Task\_\_c, COUNT(Id) FROM Feedback\_\_c WHERE Approved\_\_c = TRUE GROUP BY Related\_Task\_\_c]

SOQL performance tips:

- Always filter by indexed fields when possible (Id, OwnerId, CreatedDate, custom indexed fields).   
 - Avoid SELECT \*; only retrieve the fields you need.   
 - Use LIMIT when possible.   
 - Use binding variables (WHERE Id IN :idSet) for bulk processing.

SOSL example (text search across objects):

List<List<SObject>> sr = [FIND 'refund' IN ALL FIELDS RETURNING AI\_Suggestion\_\_c(Id, Suggested\_Text\_\_c), Task\_\_c(Id, Task\_Name\_\_c)];  
if (!sr.isEmpty()) {  
 List<AI\_Suggestion\_\_c> matchedSuggestions = (List<AI\_Suggestion\_\_c>)sr[0];  
}  


## 5. Collections: List, Set, Map

Collections are essential in Apex to process data in bulk and to avoid excessive SOQL/DML. Use List for ordered collections, Set for unique values, and Map for fast lookups.

Examples and patterns:

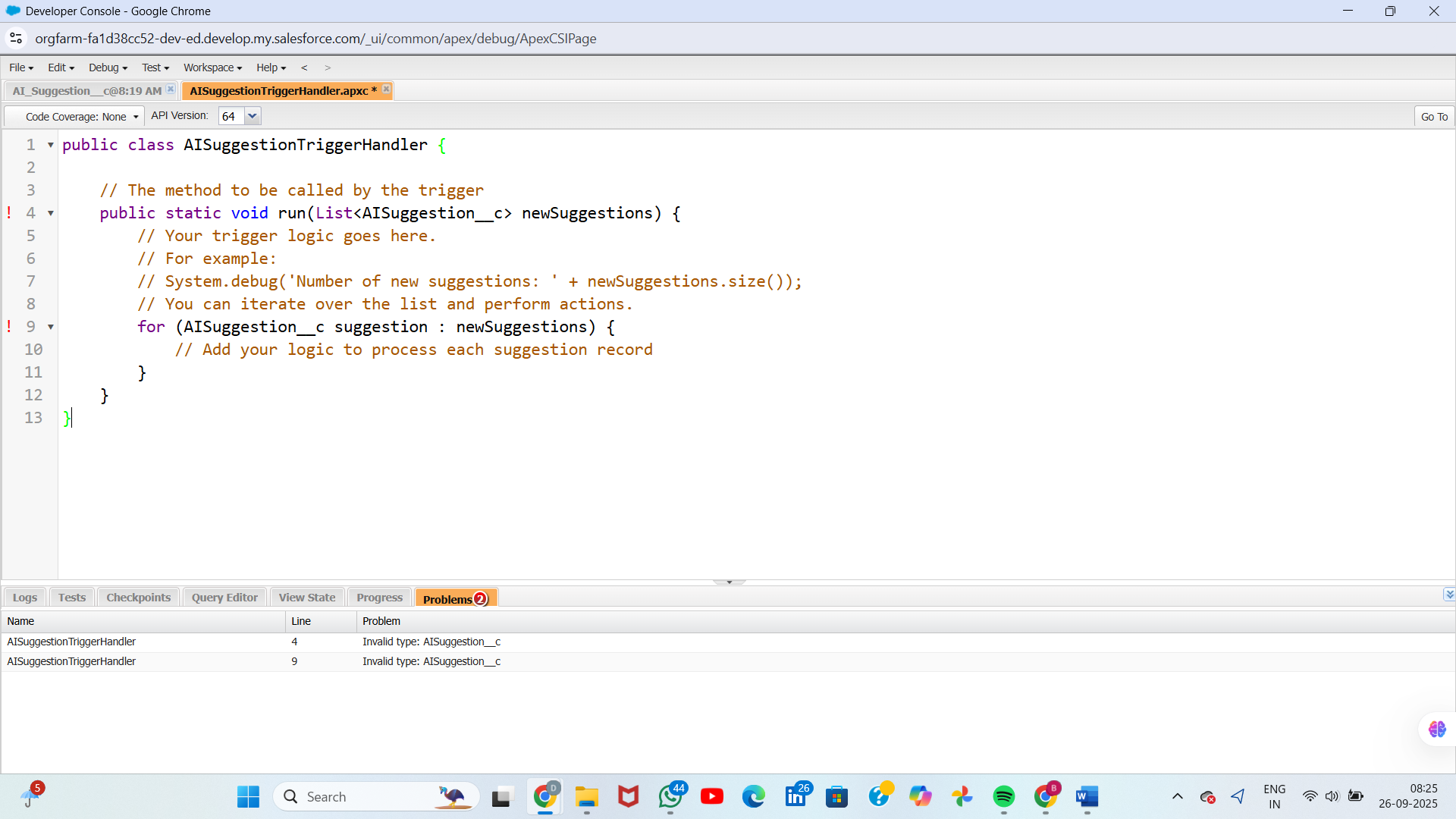
- Build a Set of IDs to query related records:

Set<Id> suggestionIds = new Set<Id>();  
for (AI\_Suggestion\_\_c s : Trigger.new) {  
 suggestionIds.add(s.Id);  
}  
List<AI\_Suggestion\_\_c> suggestions = [SELECT Id, Related\_Task\_\_c FROM AI\_Suggestion\_\_c WHERE Id IN :suggestionIds];

- Build a Map for quick lookup:

Map<Id, Task\_\_c> taskMap = new Map<Id, Task\_\_c>([SELECT Id, Task\_Name\_\_c FROM Task\_\_c WHERE Id IN :taskIds]);  
// Access: Task\_\_c t = taskMap.get(someId);

- Use Maps to group child records by parent:

Map<Id, List<Feedback\_\_c>> feedbackBySuggestion = new Map<Id, List<Feedback\_\_c>>();  
for (Feedback\_\_c f : [SELECT Id, Related\_Suggestion\_\_c FROM Feedback\_\_c WHERE Related\_Suggestion\_\_c IN :suggestionIds]) {  
 if (!feedbackBySuggestion.containsKey(f.Related\_Suggestion\_\_c)) feedbackBySuggestion.put(f.Related\_Suggestion\_\_c, new List<Feedback\_\_c>());  
 feedbackBySuggestion.get(f.Related\_Suggestion\_\_c).add(f);  
}  


**Collections: List, Set, Map**

**Why:** Use collections to bulk-process records and avoid repeated queries.

**Typical pattern**

* Collect IDs in a Set<Id>; then query related records with WHERE Id IN :idSet.
* Use Map<Id, SObject> to lookup fast without extra queries.

**Example**

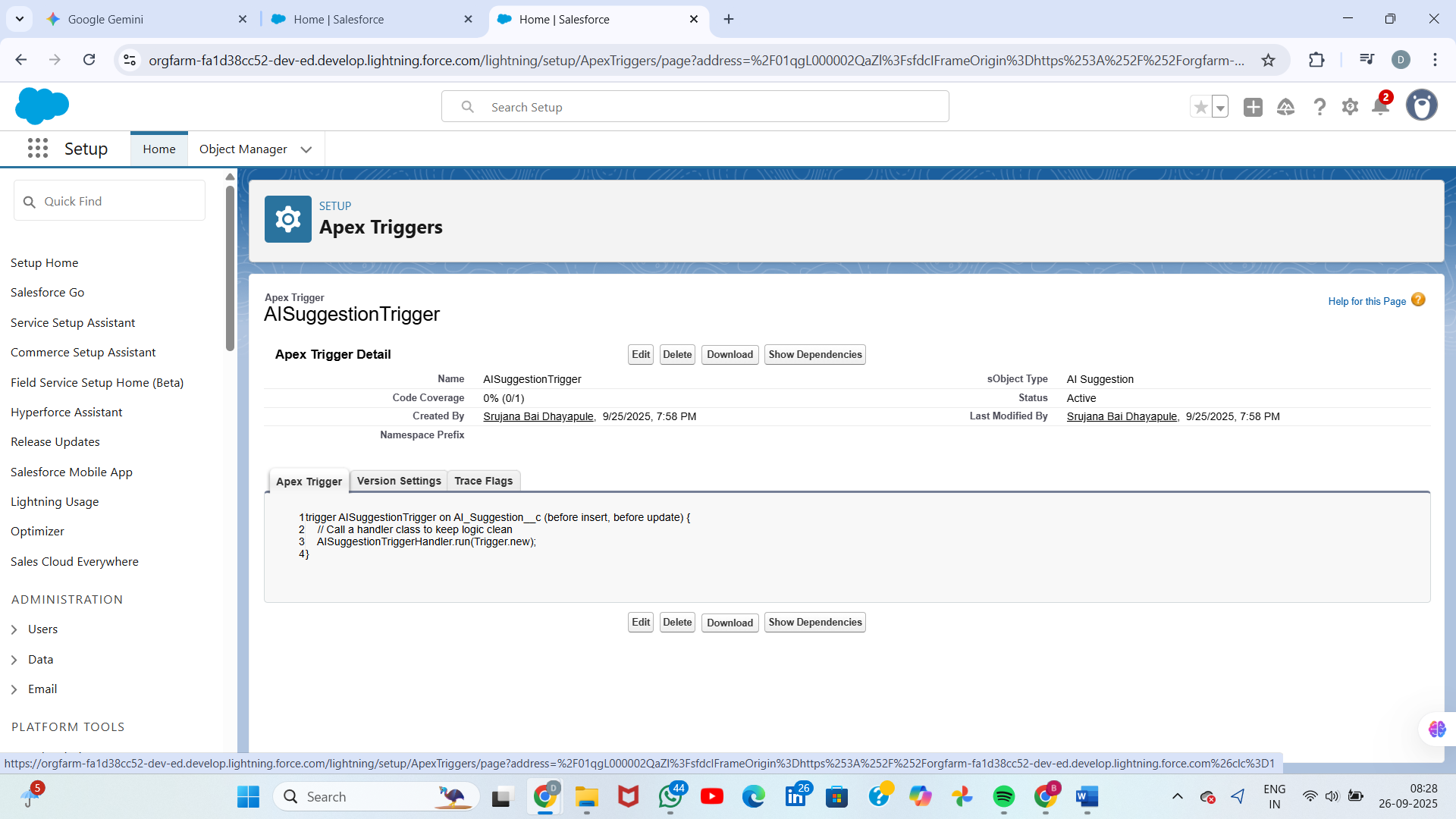
Set<Id> suggestionIds = new Set<Id>();

for (AI\_Suggestion\_\_c s : suggestions) suggestionIds.add(s.Id);

Map<Id, AI\_Suggestion\_\_c> suggestionMap = new Map<Id, AI\_Suggestion\_\_c>(

[SELECT Id, Related\_Task\_\_c FROM AI\_Suggestion\_\_c WHERE Id IN :suggestionIds]

);



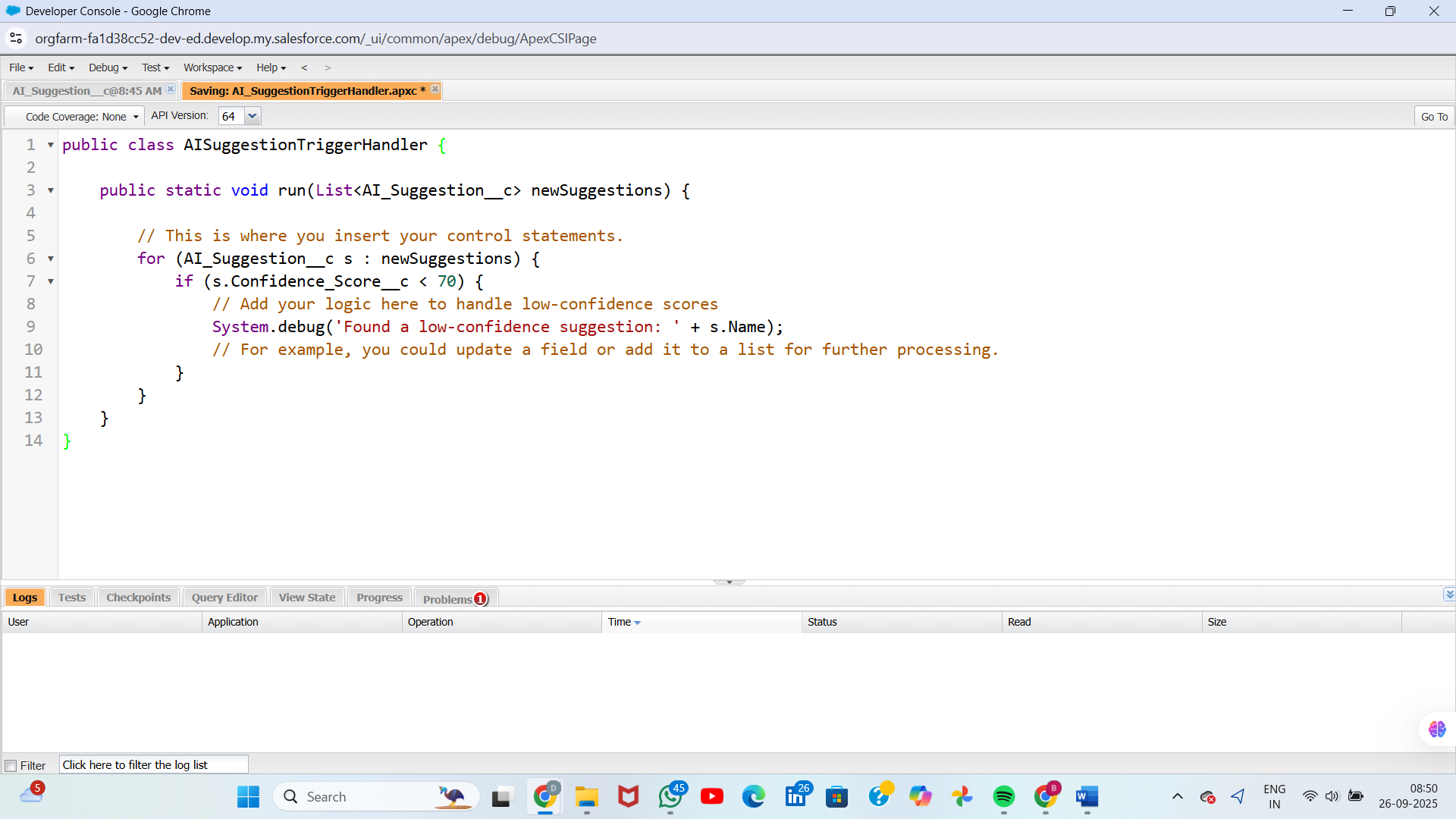
Best practices: Avoid re-initializing lists inside loops, use collection methods efficiently, and guard against nulls.

## 6. Control Statements

Control flow statements (if, for, while, switch) guide logic execution. Use for-each loops over collections to support bulk processing.

Examples:

for (AI\_Suggestion\_\_c s : suggestions) {  
 if (s.Confidence\_Score\_\_c < 70) {  
 // queue for supervisor review  
 } else if (s.Confidence\_Score\_\_c >= 70 && s.Confidence\_Score\_\_c < 90) {  
 // mark as moderate confidence  
 } else {  
 // high confidence  
 }  
}



## 7. Batch Apex

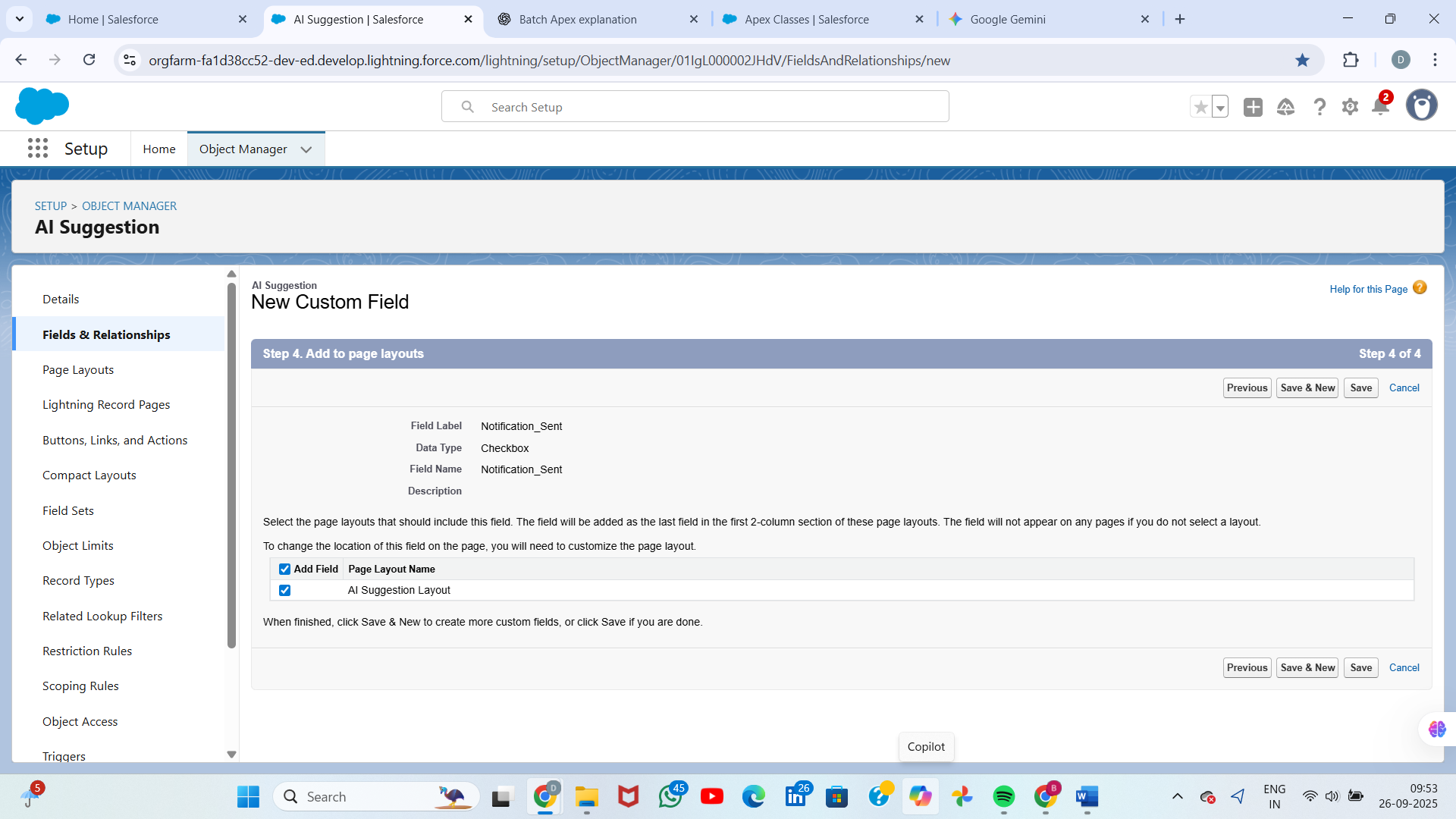
Purpose: Process large data volumes asynchronously, handle records in chunks to stay within governor limits.

When to use: Nightly auditing jobs, processing thousands of AI suggestions, or recalculating aggregated metrics.

Implementing a Batch Apex class: create a class that implements Database.Batchable<SObject> with start, execute, and finish methods. Optionally implement Database.Stateful to preserve state across batches.

**Verify the Field Name in Salesforce**

1. Navigate to **Setup** in your Salesforce org.
2. In the Quick Find box, search for **Object Manager** and select it.
3. Find and click on the **AI\_Suggestion** object.
4. On the left-hand side, click on **Fields & Relationships**.
5. Look for a field related to Notification\_Sent. Pay close attention to the **Field Name**. It might have a different name, such as Notification\_Status\_\_c, or a similar name with a different spelling.

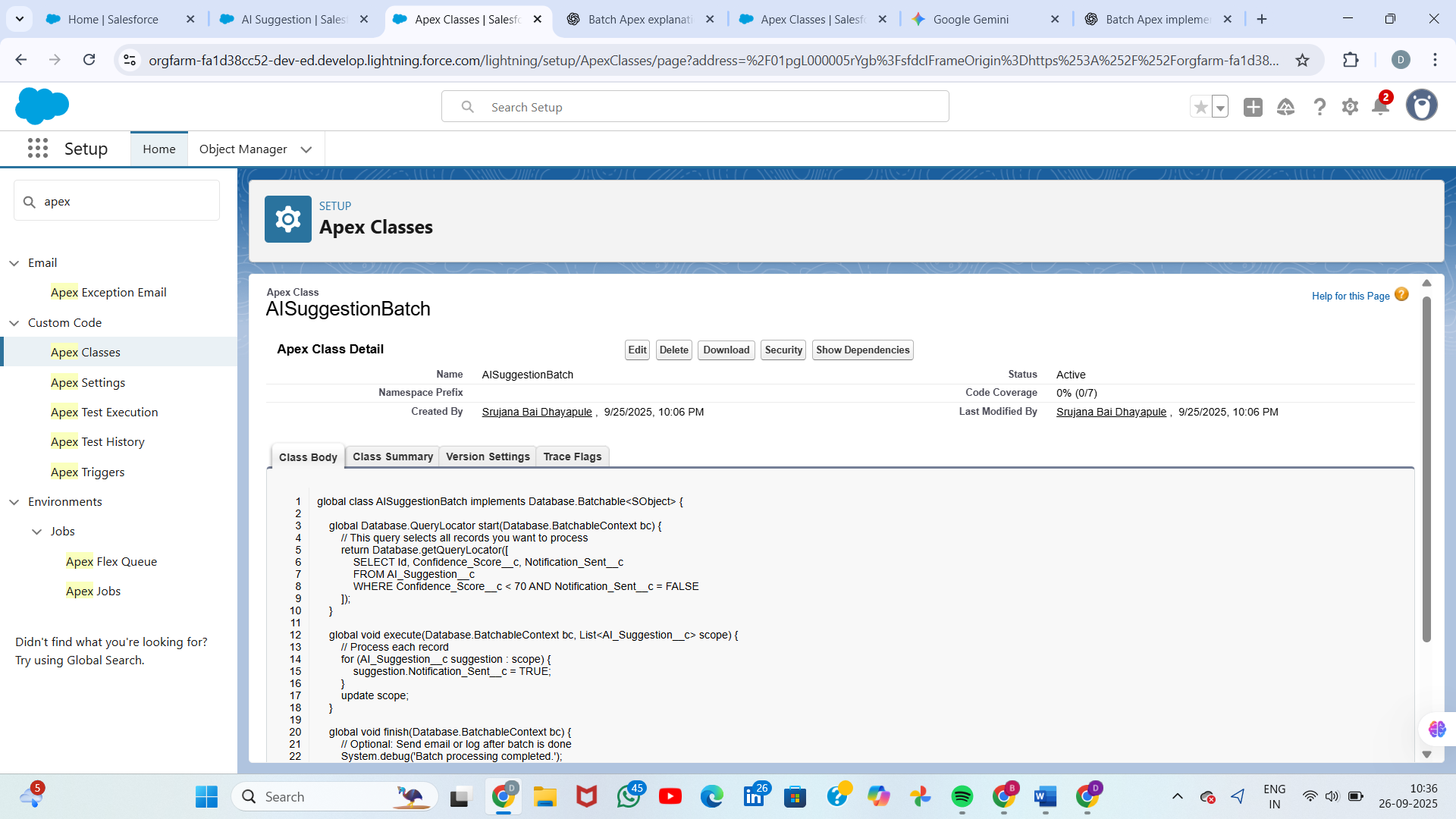
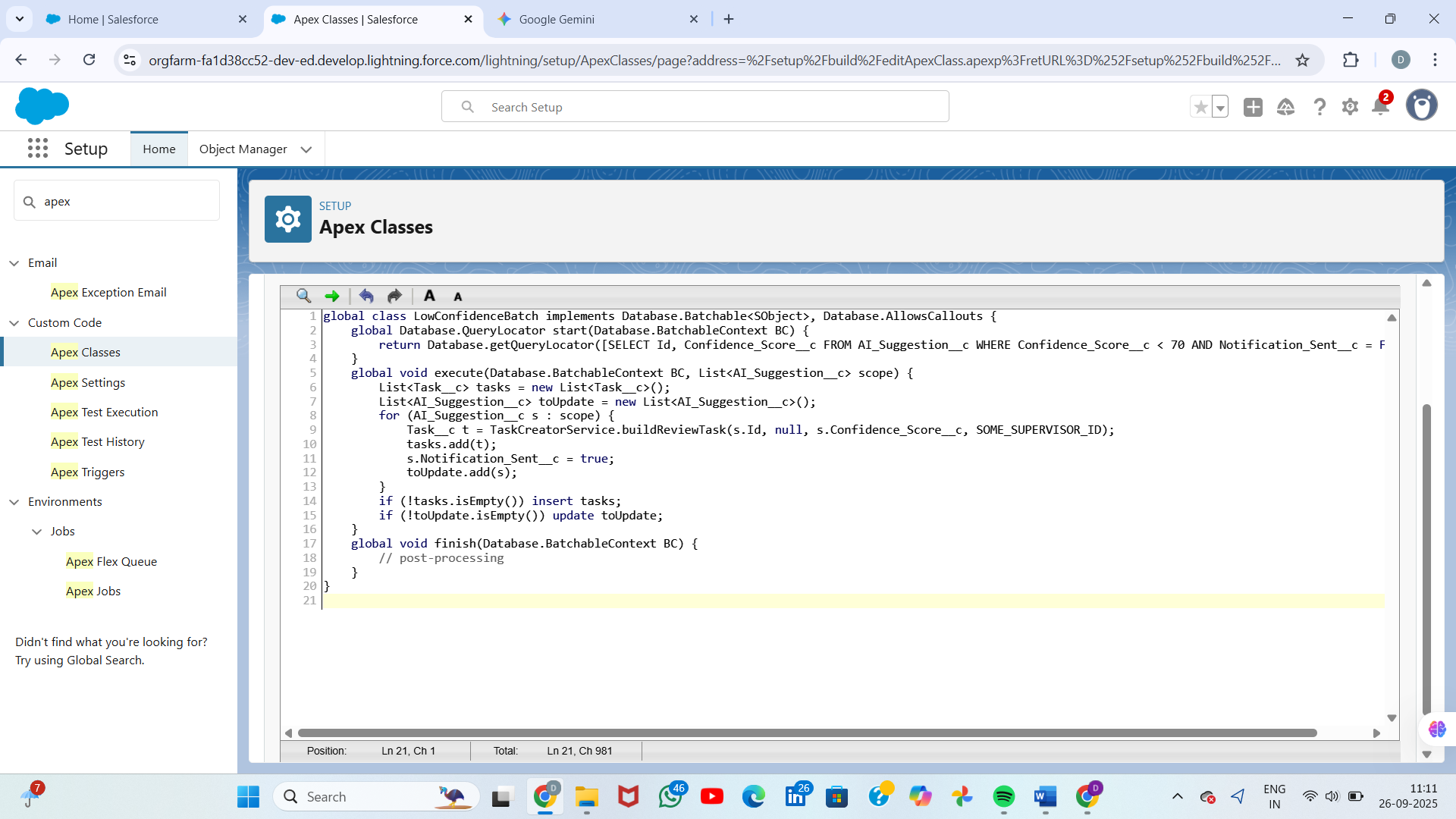


Example Batch class:

global class LowConfidenceBatch implements Database.Batchable<SObject>, Database.AllowsCallouts {  
 global Database.QueryLocator start(Database.BatchableContext BC) {  
 return Database.getQueryLocator([SELECT Id, Confidence\_Score\_\_c FROM AI\_Suggestion\_\_c WHERE Confidence\_Score\_\_c < 70 AND Notification\_Sent\_\_c = FALSE]);  
 }  
 global void execute(Database.BatchableContext BC, List<AI\_Suggestion\_\_c> scope) {  
 List<Task\_\_c> tasks = new List<Task\_\_c>();  
 List<AI\_Suggestion\_\_c> toUpdate = new List<AI\_Suggestion\_\_c>();  
 for (AI\_Suggestion\_\_c s : scope) {  
 Task\_\_c t = TaskCreatorService.buildReviewTask(s.Id, null, s.Confidence\_Score\_\_c, /\*supervisor id\*/ UserInfo.getUserId());  
 tasks.add(t);  
 s.Notification\_Sent\_\_c = true;  
 toUpdate.add(s);  
 }  
 if (!tasks.isEmpty()) insert tasks;  
 if (!toUpdate.isEmpty()) update toUpdate;  
 }  
 global void finish(Database.BatchableContext BC) {  
 // Optional post-processing, send summary email, etc.  
 }  
}

Deploy & run: To run: Database.executeBatch(new LowConfidenceBatch(), 200); The second parameter is the batch size (default 200). Choose an appropriate size based on expected DML and heap usage.

Testing Batch Apex: use Test.startTest() and Test.stopTest() to execute the batch synchronously in tests:

Test.startTest();  
Database.executeBatch(new LowConfidenceBatch(), 200);  
Test.stopTest();  
  


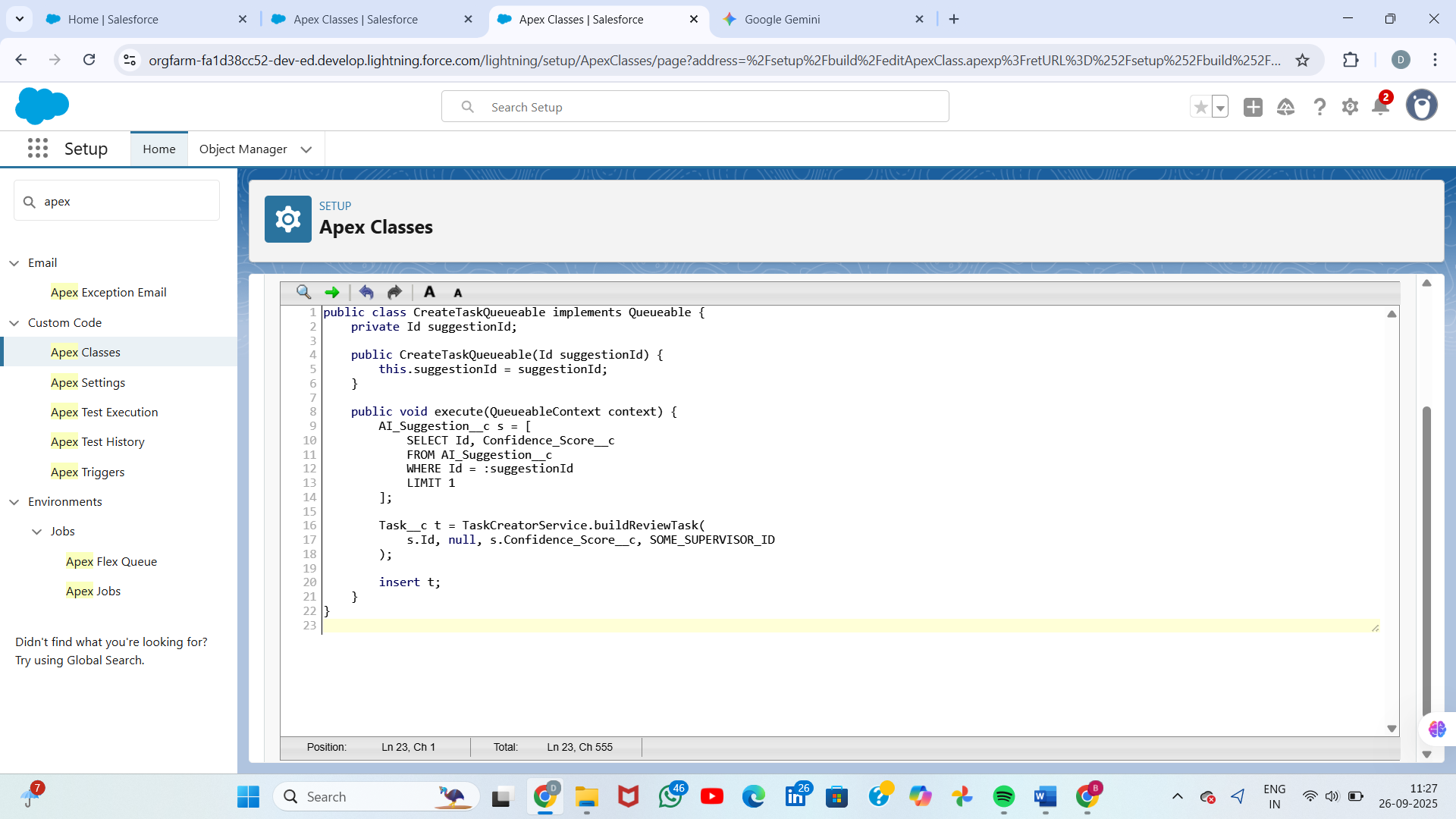
## 8. Queueable Apex

Purpose: Flexible asynchronous jobs that can be enqueued and chained. Preferred over @future for complex async logic.

Example Queueable class:

public class CreateTaskQueueable implements Queueable {  
 private Id suggestionId;  
 public CreateTaskQueueable(Id suggestionId) { this.suggestionId = suggestionId; }  
 public void execute(QueueableContext context) {  
 AI\_Suggestion\_\_c s = [SELECT Id, Confidence\_Score\_\_c FROM AI\_Suggestion\_\_c WHERE Id = :suggestionId LIMIT 1];  
 Task\_\_c t = TaskCreatorService.buildReviewTask(s.Id, null, s.Confidence\_Score\_\_c, UserInfo.getUserId());  
 insert t;  
 }  
}

Use System.enqueueJob(new CreateTaskQueueable(someId)); to run. You can chain queueables by enqueueing another job inside execute().

Limits: up to 50 jobs added to the queue per transaction; queueable jobs are subject to asynchronous limits.  


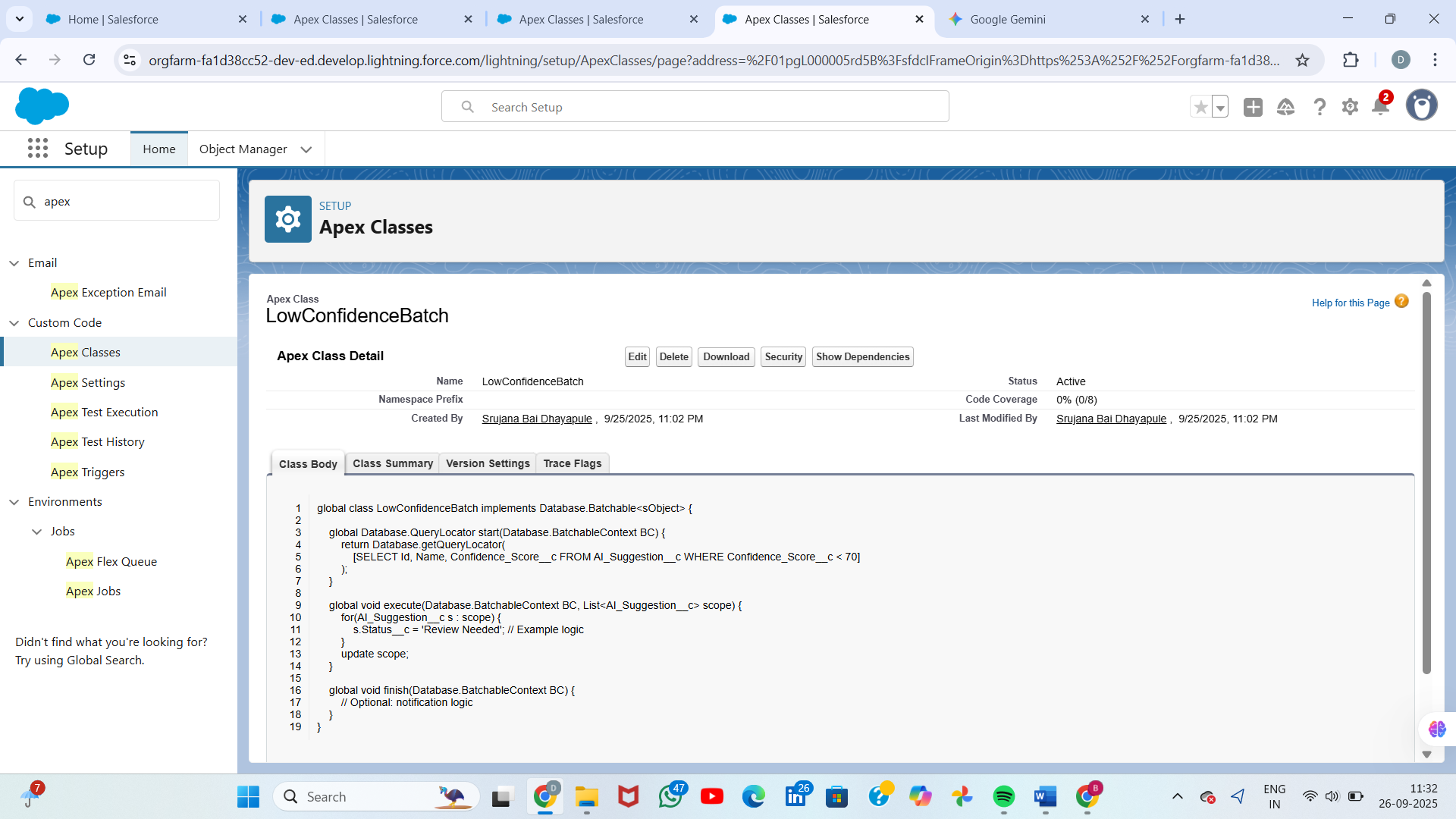
## 9. Scheduled Apex

Purpose: Schedule Apex classes to run at specified times using cron expressions. Good for daily audits or periodic batch launches.

Example schedulable class that starts the Batch Apex:

global class NightlyLowConfidenceScheduler implements Schedulable {  
 global void execute(SchedulableContext sc) {  
 Database.executeBatch(new LowConfidenceBatch(), 200);  
 }  
}

Schedule via UI: Setup → Apex Classes → Schedule Apex. Provide a name and CRON schedule (example: '0 0 2 \* \* ?' to run at 2:00 AM daily).

Testing scheduled apex: schedule it in tests with Test.startTest/stopTest and System.schedule to trigger synchronous execution in tests.  


**Create the Schedulable Class**

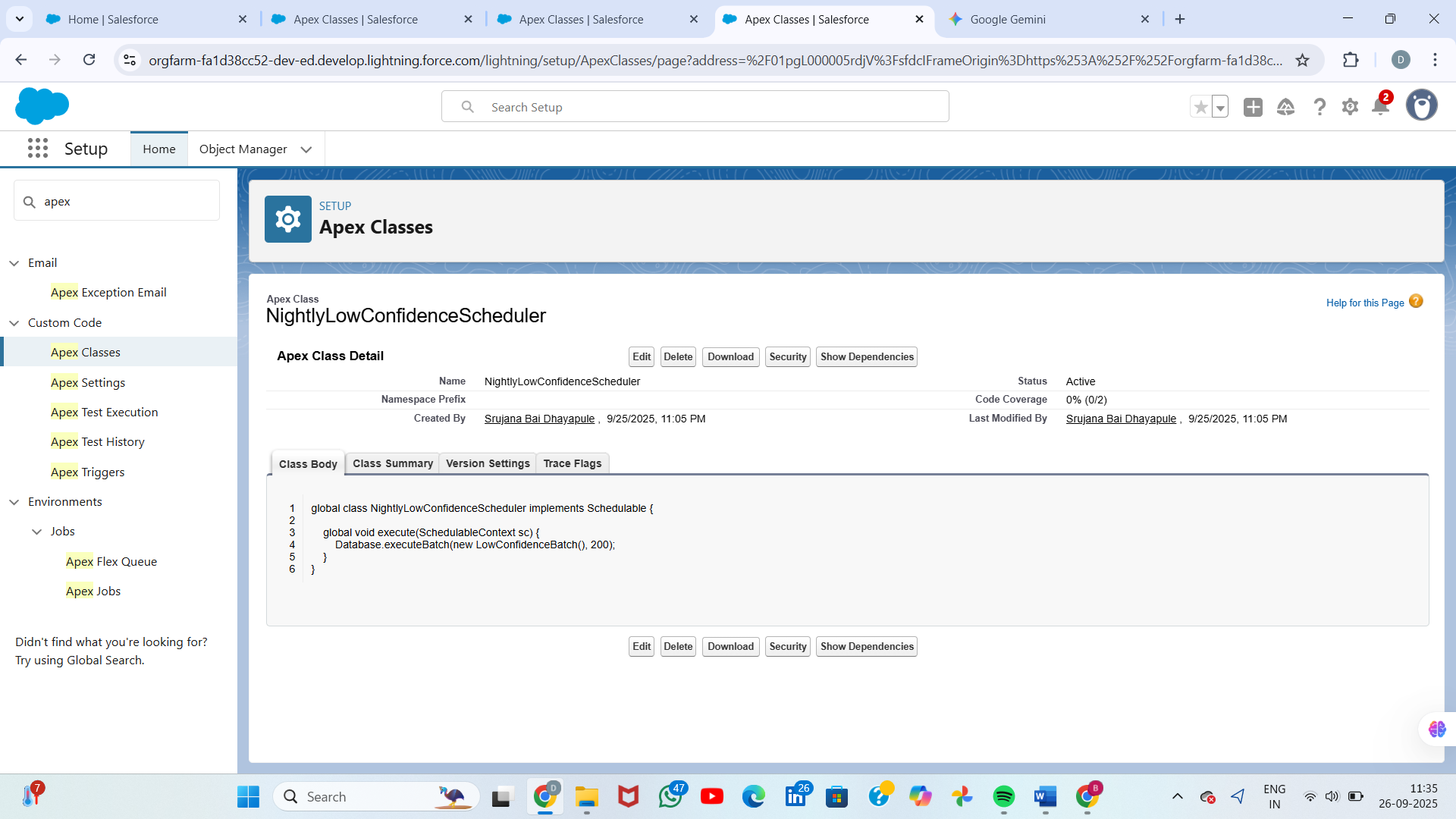
Now go to **Setup → Apex Classes → New** and create the scheduler:

global class NightlyLowConfidenceScheduler implements Schedulable {

global void execute(SchedulableContext sc) {

Database.executeBatch(new LowConfidenceBatch(), 200);

}

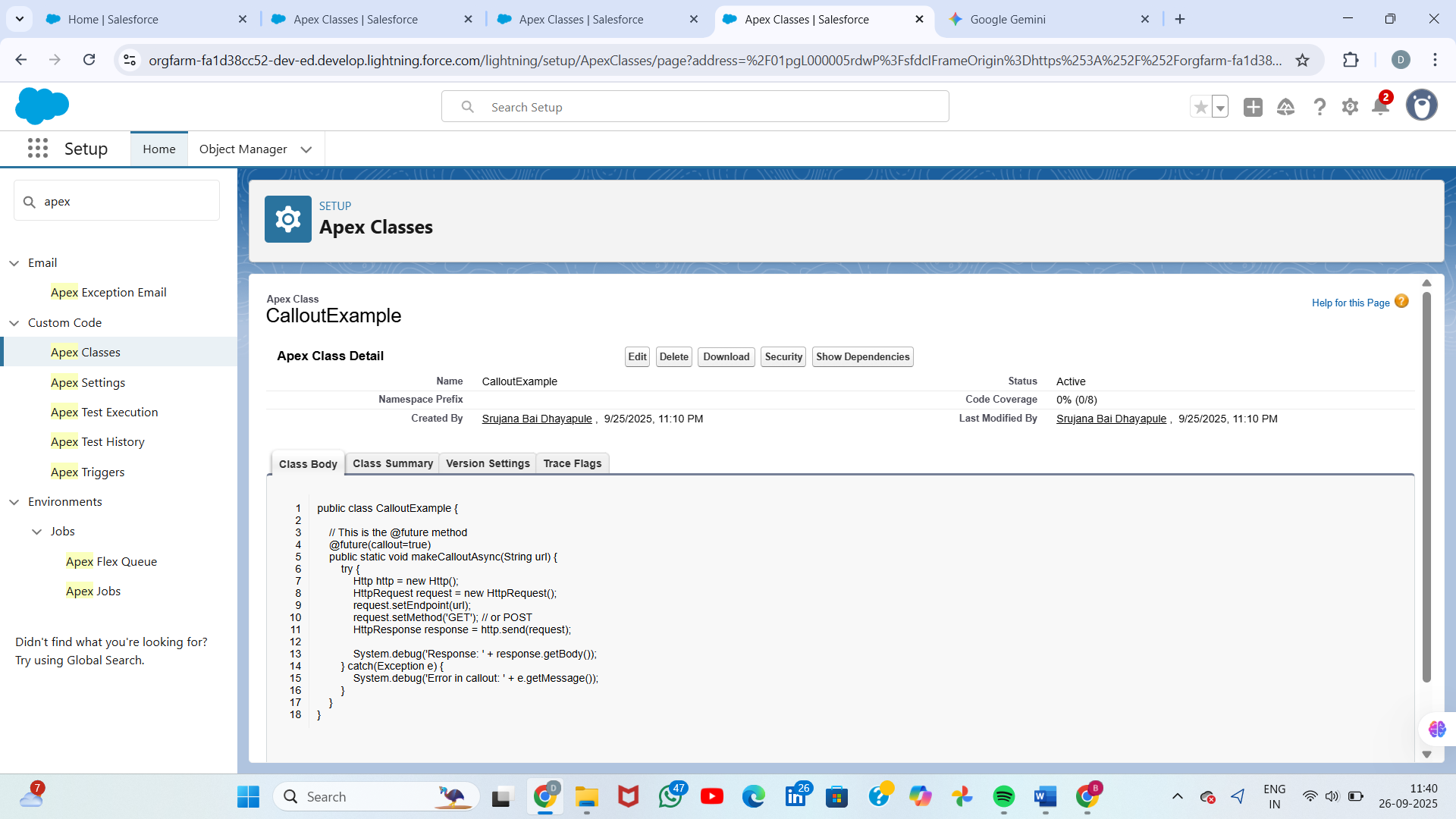


## 10. Future Methods

Purpose: Legacy approach to run asynchronous methods, primarily for callouts. Use @future only when necessary; prefer Queueable Apex.

Signature example:

@future(callout=true)  
public static void makeCalloutAsync(String endpoint) {  
 HttpRequest req = new HttpRequest();  
 req.setEndpoint(endpoint);  
 req.setMethod('GET');  
 Http http = new Http();  
 HTTPResponse res = http.send(req);  
 // handle response  
}  
  
Limitations: no chaining, limited parameter types (primitive or arrays of primitive), limited to 50 future calls per 24 hours per org typically (limits vary).

Create Apex Class  


## 11. Exception Handling

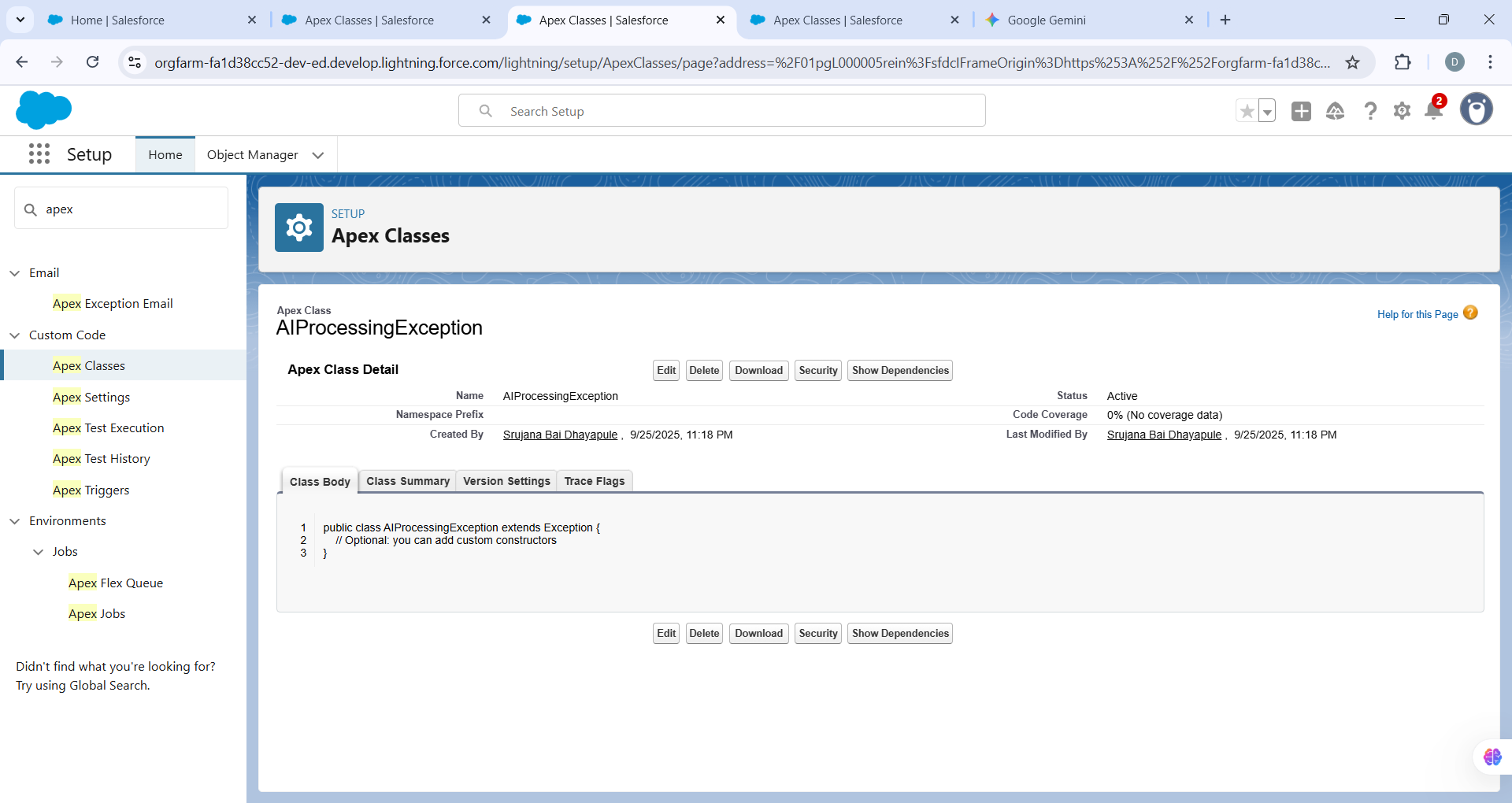
Purpose: Catch and handle runtime errors gracefully, log problems, and optionally rethrow custom exceptions for clarity.

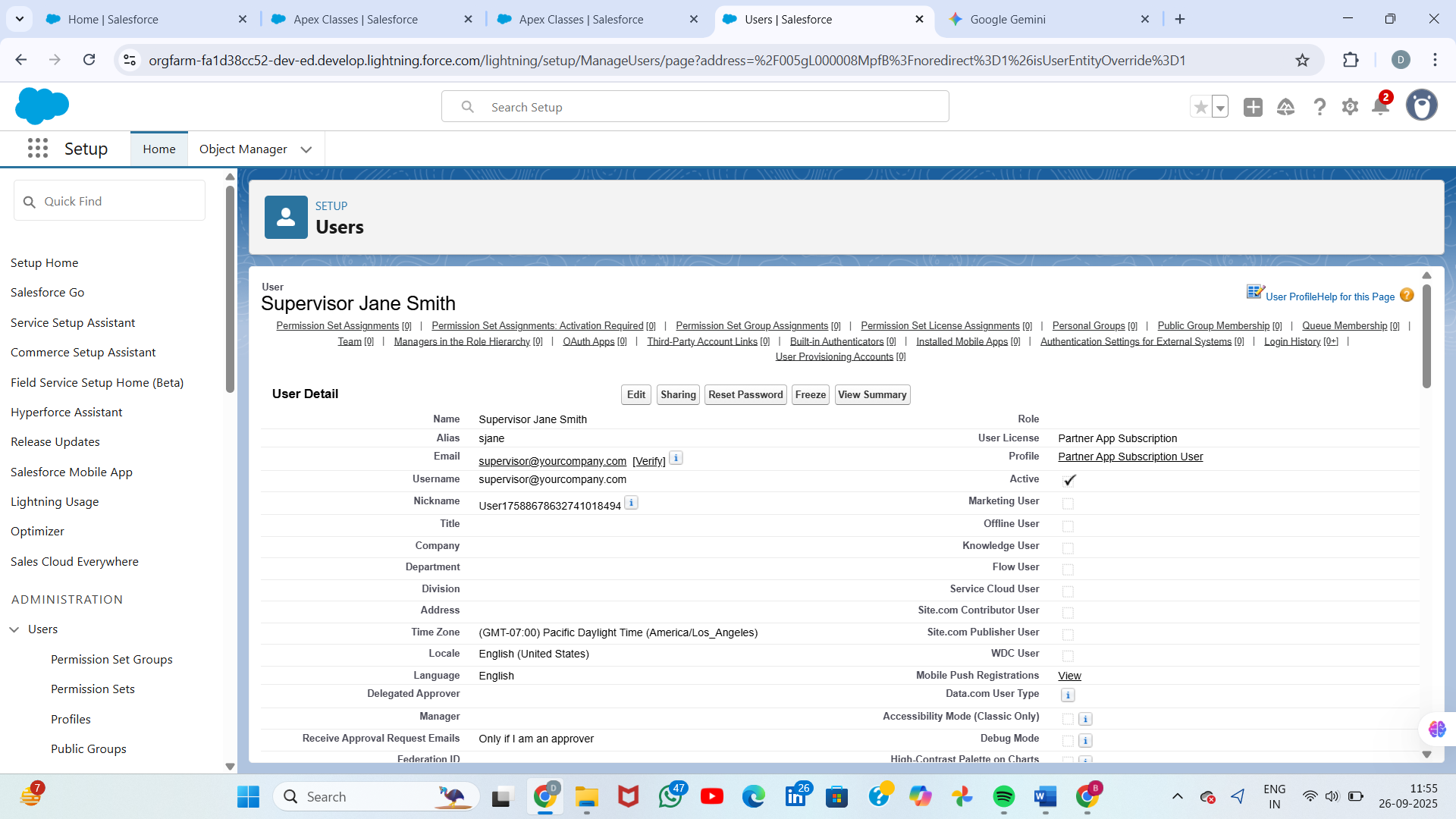
Pattern and example:

public class AIProcessingException extends Exception {}  
  
try {  
 // risky code that may throw DmlException or CalloutException  
} catch (DmlException de) {  
 // Log or handle DML-specific issues  
 System.debug('DML failed: ' + de.getMessage());  
 // Optionally notify admin via email or create an error task  
} catch (Exception e) {  
 // General fallback  
 System.debug('Error: ' + e.getMessage());  
 throw new AIProcessingException('AI processing failed: ' + e.getMessage());  
} finally {  
 // clean up resources if needed  
}  
Best practices: avoid swallowing exceptions silently; capture contextual information (record Ids, user id) in logs; consider creating an Error\_\_c custom object to track failures for admin review.

**Create a Custom Exception Class**

1. In Developer Console: **File → New → Apex Class**.
2. Name it AIProcessingException.





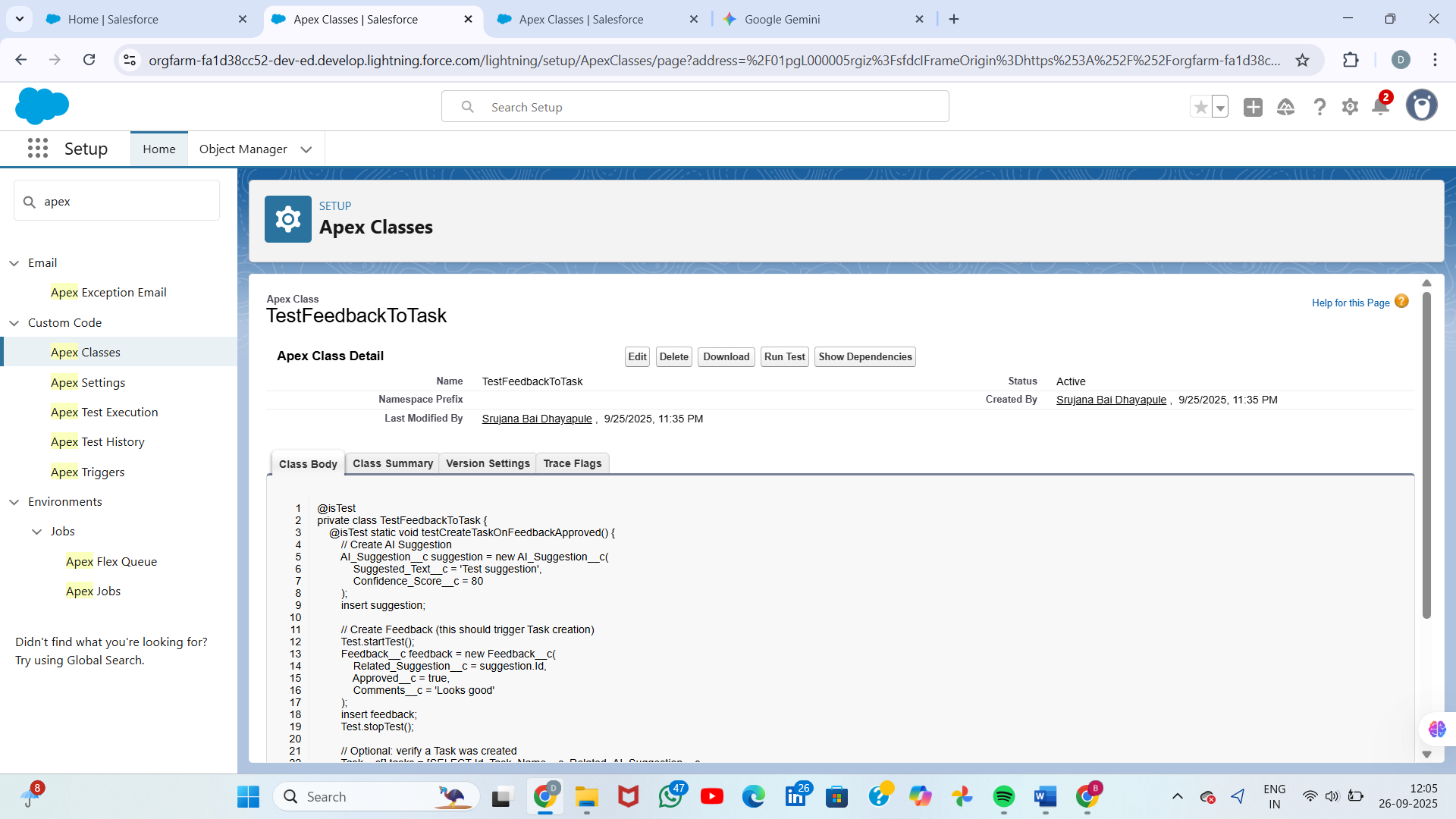
## 12. Test Classes

Purpose: Unit tests verify logic and provide required code coverage. Use @isTest annotation and seeAllData=false to create isolated test data.

Structure and tips:

- Create test data within the test class (no reliance on org data).   
 - Use Test.startTest() and Test.stopTest() around asynchronous calls to force execution.   
 - Use System.assert to validate outcomes.   
 - Test positive and negative scenarios, and bulk scenarios (insert 200 records to simulate bulk behavior).

Example test class for the Feedback->Task creation scenario:

@isTest  
private class TestFeedbackToTask {  
 @isTest static void testCreateTaskOnFeedbackApproved() {  
 // Create AI Suggestion record  
 AI\_Suggestion\_\_c s = new AI\_Suggestion\_\_c(Suggested\_Text\_\_c='Test Suggestion', Confidence\_Score\_\_c=80);  
 insert s;  
  
 Test.startTest();  
 // Create feedback that should trigger task creation  
 Feedback\_\_c fb = new Feedback\_\_c(Related\_Suggestion\_\_c = s.Id, Approved\_\_c = true, Comments\_\_c = 'Looks good');  
 insert fb;  
 Test.stopTest();  
  
 // Assert that task is created  
 List<Task\_\_c> tasks = [SELECT Id, Task\_Name\_\_c, Related\_AI\_Suggestion\_\_c FROM Task\_\_c WHERE Related\_AI\_Suggestion\_\_c = :s.Id];  
 System.assertEquals(1, tasks.size(), 'A review task should be created for approved feedback');  
 System.assertEquals('Review Approved Feedback', tasks[0].Task\_Name\_\_c);  
 }}  


## 13. Asynchronous Processing

Choices: Synchronous Apex for small operations, Queueable for single-job async needs, Batch for large volume, Scheduled to run batches on a schedule, Future for simple callouts.

Decision guide:

- If processing > 50,000 records or long-running operations: Batch Apex.   
 - If need simple async job or chain jobs: Queueable.   
 - If scheduling periodic work: Scheduled Apex launching a Batch.   
 - If you must do HTTP callouts asynchronously and cannot use Queueable: @future(callout=true).

Monitoring: Use Setup → Apex Jobs to view running/completed jobs, failures, and logs. Use the AsyncApexJob and CronTrigger objects to query job statuses if needed.

## Governor Limits & Best Practices

Apex runs in a multi-tenant environment and must obey governor limits. Follow these practices to keep within limits:

- Bulkify all logic; process lists instead of single records.   
 - Avoid SOQL/DML inside loops.   
 - Use Collections (Set/Map/List) to minimize queries.   
 - Use batch/queueable for heavy processing.   
 - Prefer declarative automations (Flows) where suitable to reduce custom code surface.   
 - Keep transactions small and focused.   
 - Add meaningful unit tests that run in bulk and single-record scenarios.