Phase 7:Integration and External Access

We will enhance the application by integrating it with an external, third-party service. The goal was to build a meaningful feature that adds significant value, moving beyond simple data display to provide intelligent, actionable insights for advisors. The chosen feature was an **"AI-Powered Skill Gap Analysis."**

7.1. Web Services (REST/SOAP) & Callouts

- **Service Type:** A **REST (REpresentational State Transfer)** web service was used for this integration. This is the modern, industry-standard protocol for public web APIs, using standard HTTP methods. The Google Gemini API, which we integrated with, is a RESTful service.
- **Callout Implementation:** The integration was implemented as an **outbound Apex Callout**. An @InvocableMethod was created in a new Apex class (AICareerSuggesterController) which is called from a Screen Flow. This method is responsible for constructing and sending the HTTP request.
    - **HttpRequest:** An HttpRequest object was instantiated in Apex.
    - **Method:** The POST method was used, as required by the Gemini API's generateContent endpoint to send our prompt in the request body.
    - **Endpoint:** The endpoint URL was dynamically constructed in Apex to include the API key. After significant debugging, the final, correct endpoint for a valid model was determined to be: https://generativelanguage.googleapis.com/v1beta/models/gemini-pro-latest:generateContent?key=...
    - **Body:** A JSON-formatted string was created to encapsulate the dynamic prompt sent to the AI.
    - **Http().send(request):** The callout was executed using the standard Http class. The code was wrapped in a try/catch block to gracefully handle potential CalloutExceptions, such as the Read timed out error we encountered.
- **Response Handling:** The HttpResponse was processed by checking the status code (response.getStatusCode()). For a successful , a custom Apex wrapper class (GeminiResponseParser) was used to deserialize the complex JSON response into strongly typed Apex objects, making it easy to extract the AI-generated text.

7.2. OAuth & Authentication

- **Authentication Method:** The Google Gemini API uses a simple **API Key** authentication method. This requires the secret key to be passed either in the URL as a query parameter or in a custom HTTP header.
- **Implementation:** We implemented authentication by passing the API Key as a key parameter in the endpoint URL. This proved to be the most reliable method.
- **OAuth 2.0 was considered but was not the appropriate protocol** for this specific API, which requires server-to-server authentication rather than user-delegated authorization.

7.3. Named Credentials & Remote Site Settings (Security & Endpoint Management)

This was a major focus of our debugging process. We explored two methods for managing the endpoint and credentials.

- **Method 1: The Modern External Credential Framework (Attempted)**
    - An External Credential was created to store the API key, and a Named Credential was created to store the URL. An Apex class was written to use the callout:Your_Named_Credential syntax and the '{!$Credential...}' merge field.
    - **Outcome:** This method failed. Extensive debugging, including analyzing debug logs, proved that the framework was not correctly injecting the secret key into the required custom header for this specific API, resulting in a persistent 400 Bad Request (API key not valid) error from the server. This was a critical learning experience in the limitations of a new framework.
- **Method 2: The Custom Setting & Remote Site Setting (Successful Implementation)**
    - **This was our final, successful approach.**
    - **Remote Site Setting:** A Remote Site Setting was created for https://generativelanguage.googleapis.com, which "whitelisted" this domain, giving Apex permission to make callouts to it.
    - **Hierarchy Custom Setting:** A Custom Setting (AI_Settings__c) was created to securely store the Gemini_API_Key__c outside of the Apex code. This follows the best practice of separating configuration from code.
    - **Apex Logic:** The final version of the Apex class queries this Custom Setting at runtime to retrieve the API key, making the code secure and maintainable.

7.4. API Limits

- **Salesforce Governor Limits:** The implementation is highly conscious of Salesforce governor limits. The entire feature is initiated by a single user click, resulting in a **single**

Apex transaction that performs **one SOQL query** and **one callout**. This is extremely efficient and well within the limit of 100 callouts per transaction.

- **External API Limits:** The code is also designed to be mindful of the external API's limits. By running on demand (when a user clicks a button) rather than in a trigger or a batch job, we avoid making excessive, automated calls that could quickly exhaust the free tier limit of the Gemini API.

Apex code:

1.AICareerSuggesterController.cls: for web services and callout

```
public with sharing class AICareerSuggesterController {


    // Inner classes for Flow Input and Output

    public class FlowInput { @InvocableVariable(label='Student Record ID' required=true)
public Id studentId; }

    public class FlowOutput { @InvocableVariable(label='AI Suggestions') public String
suggestions; }


    @InvocableMethod(label='Get AI Career Suggestions')

    public static List<FlowOutput> getSuggestions(List<FlowInput> inputs) {

        // Step 1: Get Input and Query Skills

        Id studentId = inputs[0].studentId;

        List<String> skillNames = new List<String>();

        for (Student_Skill__c ss : [SELECT Skill__r.Name FROM Student_Skill__c WHERE
Student__c = :studentId]) {

            skillNames.add(ss.Skill__r.Name);

        }


        if (skillNames.isEmpty()) {

            FlowOutput noSkillsResponse = new FlowOutput();
```

```
        noSkillsResponse.suggestions = 'This student has no skills listed. Please add skills to their record to get AI suggestions.';
        return new List<FlowOutput>{ noSkillsResponse };
    }

    // Step 2: Build the AI Prompt
    String existingSkills = String.join(skillNames, ', ');
    String prompt = 'You are an expert career counselor for university students in India. ' +
        'A student has the following technical skills: ' + existingSkills + '. ' +
        'Based ONLY on these skills, suggest the top 3 most relevant career paths. ' +

        'For each career path: ' +
        '1. List exactly 5 additional key technical skills, frameworks, or tools they should learn to be job-ready. ' +
        '2. For each of those 5 skills, provide a brief, one-sentence explanation of *why* it is important for that specific career path. ' +

        'Your response MUST be formatted as simple HTML. Do not include markdown. ' +
        'Use an <h3> tag for each career path title. ' +
        'Use an unordered list (<ul> with <li> tags) for the skills to learn under each path. ' +
        'Inside each <li> tag, make the skill name bold using a <strong> tag, followed by a colon and then the one-sentence explanation. ' +

        'Do not include any introductory or concluding sentences outside of the HTML structure. ' +
        'Do not use ```html code blocks in your response. ' +
        'Ensure the entire output is a single, valid HTML block.';
```

```
String responseBody = '';

try {

    // Step A: Get the API Key from our Custom Setting
    AI_Settings__c settings = AI_Settings__c.getOrgDefaults();
    String apiKey = settings.Gemini_API_Key__c;

    // Step B: Safety check if the key is configured
    if (String.isBlank(apiKey)) {
        responseBody = 'Error: The Gemini API Key has not been configured in Custom Settings.';
    } else {
        // Step C: Build the request with the API Key
        HttpRequest request = new HttpRequest();

        // We build the full URL and include the key as a query parameter
        request.setEndpoint('https://generativelanguage.googleapis.com/v1beta/models/gemini-pro-latest:generateContent?key=' + apiKey);
        request.setMethod('POST');
        request.setTimeout(60000); // Set timeout to 60,000 milliseconds (60 seconds)

        request.setHeader('Content-Type', 'application/json');

        String requestBodyJson = '{"contents":[{"parts":[{"text": "' + prompt + '"}]}]}';
        request.setBody(requestBodyJson);
```

```
            // Step D: Make the callout
            Http http = new Http();
            HttpResponse response = http.send(request);

            if (response.getStatusCode() == 200) {
                Map<String, Object> results = (Map<String, Object>)
JSON.deserializeUntyped(response.getBody());
                List<Object> candidates = (List<Object>) results.get('candidates');
                Map<String, Object> firstCandidate = (Map<String, Object>) candidates[0];
                Map<String, Object> content = (Map<String, Object>)
firstCandidate.get('content');
                List<Object> parts = (List<Object>) content.get('parts');
                Map<String, Object> firstPart = (Map<String, Object>) parts[0];
                responseBody = (String) firstPart.get('text');
            } else {
                responseBody = 'Error from AI Service: ' + response.getStatus() + ' - ' +
response.getBody();
            }
        }
    } catch (Exception e) {
        responseBody = 'A Salesforce error occurred: ' + e.getMessage();
    }

    // Step 7: Return the result to the flow
    FlowOutput finalResponse = new FlowOutput();
    finalResponse.suggestions = responseBody;
    return new List<FlowOutput>{ finalResponse };
```

```
    }
}
```

2. GeminiResponseParser: custom Apex wrapper class to handle the response

```
public class GeminiResponseParser {
    public class Candidate {
        public Content content;
    }
    public class Content {
        public List<Part> parts;
    }
    public class Part {
        public String text;
    }
    public List<Candidate> candidates;

    // This is a special method to create an instance of the class from JSON
    public static GeminiResponseParser parse(String json) {
        return (GeminiResponseParser) System.JSON.deserialize(json,
GeminiResponseParser.class);
    }
}
```

Here we used a(Screen flow) flow to implement the AI gap analyzer

1. **Create Flow:** A **Screen Flow** was created with the label "AI Career Suggester."
2. **Create Resources:** Several resources were created to manage data within the flow:
    - **recordId (Variable, Text, Input):** Receives the ID of the Student record from the Lightning Page.
    - **varStudentRecord (Variable, Record, Contact):** A container to hold the full Student record after it is fetched.

- **varAIResponse (Variable, Text):** A container to store the final HTML-formatted response from the Apex action.

3. **Add Get Records Element:**
    - The first element in the flow is a **Get Records** labeled "Get Student Record."
    - It queries the Contact object for the record where the Id equals the incoming {!recordId}.
    - The result is stored in the {!varStudentRecord} variable. This step makes the student's data (like their name) available to the flow.
4. **Add "Launch" Screen:**
    - A **Screen** element labeled "Confirm Career Analysis" is presented to the user.
    - It contains a **Display Text** component with a dynamic greeting: Click 'Analyze' to use AI to analyze existing skills for {!varStudentRecord.FirstName} ...
    - This provides a good user experience by confirming the context before the process begins.
5. **Add Apex Action Element:**
    - After the first screen, an **Action** element is used to call our custom Apex logic.
    - **Action:** The custom @InvocableMethod labeled **"Get AI Career Suggestions"** is selected.
    - **Input Mapping:** The action's Student Record ID input parameter is mapped to the flow's {!recordId} variable.
    - **Output Mapping:** The action's AI Suggestions output is stored in the {!varAIResponse} text variable.
6. **Add "Results" Screen:**
    - A final **Screen** element labeled "AI Career Suggestions" is used to display the outcome.
    - It contains a **Display Text** component that renders the content of the {!varAIResponse} variable. Because the Apex action returns clean HTML, this component correctly displays the formatted headings, lists, and bold text.
7. **Deployment (As a Quick Action):**
    - A **Quick Action** was created on the Contact object.
    - **Action Type:** Flow.

- **Flow:** The AI Career Suggester flow was selected.
- **Label:** "Suggest Careers from Skills."
- This action was then added to the "Mobile & Lightning Experience Actions" section of the Student's **Page Layout**, making it available as a button.