**Task 1: Dynamics 365 Customization**

**1. Detail the process of customizing the Lead entity in Dynamics 365 Sales, including how to add new fields, form scripts, customization of Ribbon and business rules. How would you ensure these customizations are both scalable and maintainable?**

**Adding New Fields**

* **Identify Requirements:** Begin by gathering requirements to understand what new fields need to be added and how they will impact the lead management process.
* **Create New Fields:**
  + Go to Solutions in the Power Platform Admin Center.
  + Open your working solution and select the Lead entity.
  + Under the Fields tab, click Add Field and define the field's properties, such as Data Type, Format, and Requirement Level.
* **Define Field Security (if necessary):** If the new fields contain sensitive data, configure Field Security Profiles to limit access to authorized users.
* **Add Fields to Forms:** Add the new fields to appropriate Lead forms (e.g., Main Form) and arrange them in a way that enhances user experience.

**2. Implementing Form Scripts (JavaScript)**

* Identify Events: Determine which form events (e.g., OnLoad, OnSave, OnChange) should trigger your scripts.
* **Write** and Test JavaScript Code:
  + Use JavaScript to write functions that validate field values, auto-populate fields, or apply conditional formatting.
  + Use libraries like Xrm to access and manipulate form data and fields within Dynamics 365.
* **Add Scripts to the Form:**
  + Go to the form customization, add a new Web Resource (JS file) containing your script, and configure the appropriate Event Handlers for each form event.
  + Ensure thorough testing of each script for both functionality and performance.

**3. Customizing the Ribbon (Command Bar)**

* Use Ribbon Workbench: Ribbon Workbench is a popular tool for customizing Ribbon buttons in Dynamics 365.
* **Add Custom Buttons:**
  + Open Ribbon Workbench within your solution and select the Lead entity.
  + You can add new buttons, commands, or even hide/disable existing ones based on business requirements.
* **Define Button Actions:**
  + Configure custom button actions, like calling JavaScript functions or Power Automate flows.
  + Set up enable/disable rules to control when buttons are accessible based on Lead record status, field values, or security roles.

**4. Setting Up Business Rules**

* **Define Conditions and Actions:**
  + Use Business Rules to enforce logic without needing JavaScript, especially for simple conditional logic, like hiding/showing fields, enabling/disabling fields, or setting default values.
* **Add Business Rules:**
  + Go to the Lead entity and select Business Rules.
  + Define conditions (e.g., “If Lead Source is ‘Web’”) and actions (e.g., “Set Priority to ‘High’”).
  + Set the scope to “Entity” to apply the rule across all forms or specify it for a particular form.
* **Test Business Rules:** Make sure the rules function as expected under various scenarios. Business Rules run client-side, so ensure they don’t conflict with form scripts.

**5. Ensuring Scalability and Maintainability**

* **Documentation:**
  + Maintain documentation on each customization, including field definitions, scripts, Ribbon changes, and business rules, with version history.
* Standardize Naming Conventions: Use consistent naming conventions for custom fields, scripts, buttons, and workflows to make it easier to manage and reference them in future development.
* **Modularize Code:**
  + For JavaScript, create separate libraries for different functionalities (e.g., validations, formatting, calculations) to avoid code duplication and promote reusability.
* **Use Managed Solutions for Deployment:**
  + When deploying customizations, use Managed Solutions in production environments. This helps control and version customizations and reduces potential conflicts during future upgrades.
* **Testing and Validation:**
  + Regularly test all customizations, especially before deploying to production. Include regression testing to ensure existing features aren’t impacted.
* **Minimize Custom Code:** Where possible, rely on Power Automate and Business Rules for simpler logic, reducing the need for JavaScript. This approach lowers technical debt and is easier to maintain over time.

**2. Describe advanced techniques for automating the Lead qualification to Opportunity**

**process, including the use of Power Automate, custom workflows, and plugins. How would you handle complex business logic in this process? To note that the opportunity will be owned by the retail which another business unit with distinct access.**

**Power Automate for Lead Qualification Workflow**

* **Trigger the Flow:**
  + Use Power Automate to trigger on the Lead qualification action (e.g., when the Lead status changes to “Qualified”).
  + Ensure the trigger includes conditions that the Lead meets qualification criteria, such as specific fields being populated or validated.
* **Automate Opportunity Creation:**
  + Power Automate can create an Opportunity and set field values based on data from the Lead. Map Lead fields to Opportunity fields in the flow to ensure continuity of data.
  + Business Unit Assignment: Since the Opportunity will be owned by the Retail business unit, configure Power Automate to assign ownership to a retail-specific user or team by setting the “Owner” field to a retail team or individual.
* **Send Notifications and Alerts:**
  + Include email or Teams notifications to alert Retail sales managers of new Opportunities. Power Automate can dynamically include Opportunity details and alert specific users or teams based on the opportunity's details.
* **Handle Error Logging and Monitoring:**
  + Add error-handling steps, such as storing failed flow instances in a custom “Error Log” entity. This enables easy monitoring of flow errors for any records that may not have qualified correctly.

**2. Custom Workflows for Specialized Processing**

* **Define Workflow on the Lead Entity:**
  + If the qualification process involves complex or conditional logic (e.g., checking custom fields, compliance checks), you can configure an on-demand or real-time workflow to evaluate these criteria. For example, a workflow could check specific fields on the Lead to ensure the Lead meets retail-specific qualification criteria.
* **Use Conditional Logic for Complex Requirements:**
  + Within the workflow, use Conditional Branching to apply business logic. For instance, if the Lead is from a high-priority source, the workflow might assign a senior sales rep as the Opportunity owner.
* **Invoke Workflow with Power Automate:**
  + While custom workflows are powerful, you can also combine them with Power Automate flows. For example, use Power Automate to trigger the custom workflow after Opportunity creation if the workflow handles further downstream business logic, such as notifications or updates to related entities.

**3. Plugins for Complex Business Logic and Security**

* **Use Plugins for Data Validation and Business Rules Enforcement:**
  + Create plugins for complex data validation and logic enforcement that Power Automate or workflows cannot manage. For instance, if qualifying a Lead requires verification of multiple criteria across different related entities, a plugin can handle this multi-entity validation.
  + Register plugins on the Qualify Lead message or Create Opportunity message to intercept the process and apply business rules.
* **Implement Ownership Logic in Plugins:**
  + In cases where business units and permissions are strictly enforced, plugins can handle cross-business unit ownership assignments more securely. For example, upon Opportunity creation, a plugin can be triggered to check if the Lead originated from a specific region and then assign the Opportunity to the Retail business unit.
  + Ensure the plugin can handle permissions by adjusting Field Security Profiles and Role Privileges for cross-business unit data access.
* **Handle Security and Access Controls:**
  + Assign Opportunities to a Retail Team or Queue rather than individual users to streamline access control across the Retail business unit. This method ensures Retail users can access Opportunities without impacting other business units.
  + Consider implementing Access Teams if you need more granular, record-specific access settings.

**4. Combining Techniques for Scalability and Maintainability**

* Use Environment Variables in Power Automate: Instead of hardcoding values (like specific team names or assignment criteria), use environment variables for easy updates across environments (e.g., Dev, UAT, Prod).

**3. Discuss strategies that should be implemented to ensure the Marketing department retains read access to the Lead when it is referred to another subsidiary where there is no sharing of the data. Note that the Marketing Department should see only leads referred to the subsidiary and the subsidiary should retain read write access on only leads created by them and referred to them. The Marketing Department should also have visibility of the opportunity created from the leads at the subsidiary company.**

**1. Security Roles and Business Unit Structure**

* **Define Separate Business Units for Marketing and Subsidiary:**
  + Organize Dynamics 365 so that the Marketing Department and Subsidiary each have distinct Business Units.
  + Assign each department or subsidiary-specific security roles that control access to Leads and Opportunities within their respective units.
* **Set Up Granular Security Roles:**
  + **Create two security roles:**
    1. **Marketing Read-Only Role:** Allows read access to Leads referred to the subsidiary and Opportunities created from those Leads.
    2. **Subsidiary Contributor Role:** Grants read/write access to Leads created and referred specifically to the subsidiary.
  + Adjust these roles to control which entities and fields are accessible, minimizing visibility to only the necessary data.

**2. Sharing Records Using Access Teams**

* **Leverage Access Teams for Record-Level Sharing:**
  + Use Access Teams to grant Marketing read-only access to specific Leads and Opportunities.
  + Create a Dynamic Access Team that automatically adds members of the Marketing Department as read-only users to Leads when a Lead is referred to the subsidiary. This setup ensures Marketing can see the Leads they’re authorized to view without giving full visibility to all Leads.
* **Define Access Team Templates:**
  + Create templates for read-only access (Marketing) and read/write access (Subsidiary).
  + Use workflows or plugins (discussed below) to dynamically add or remove users from these Access Teams based on record updates.

**3. Automating Record Sharing with Workflows or Power Automate**

* **Automate Access Team Assignments:**
  + Use a Power Automate flow or real-time workflow to assign Access Teams when specific conditions are met (e.g., a Lead is referred to the subsidiary).
  + **Trigger this automation to:**
    1. Add Marketing Department users to the Lead’s Access Team for read-only access.
    2. Add Subsidiary users with read/write permissions if they’re the designated owners.
  + Automate similar access when an Opportunity is created from a referred Lead, ensuring Marketing also receives read-only access to the Opportunity.
* **Ensure Accurate Referral and Assignment Tracking:**
  + Use custom fields or tags (e.g., Referred to Subsidiary) to track when a Lead is referred to a subsidiary. The workflow can then use these fields to manage which records are shared with the Marketing Department.
  + The workflow can be configured to check that each referred Lead or created Opportunity meets specific criteria before sharing permissions are applied.

**4. Custom Plugins for Complex Security Logic**

* **Develop Plugins for Detailed Security and Ownership Control:**
  + Implement a plugin on the Lead entity’s update message to monitor when a Lead is referred to a subsidiary.
  + The plugin can add the Marketing Department Access Team to referred Leads, automatically ensuring Marketing has read-only access.
  + Create a second plugin for Opportunity creation to automatically add Marketing’s Access Team with read-only rights when an Opportunity is created from a Lead referred to the subsidiary.
* **Set Up Ownership Validation Logic:**
  + A plugin can also enforce that only the subsidiary’s users retain read/write access to Leads and Opportunities created by or referred to them, thereby aligning security with the business unit structure.
  + The plugin should check that the assigned owner belongs to the subsidiary's business unit to enforce the proper data access control.

**5. Visibility Control with Filtering and Dashboards**

* **Filtered Dashboards and Views for Marketing:**
  + Create custom Views and Dashboards for the Marketing Department that filter Leads and Opportunities to only those relevant to them.
  + Use Advanced Find filters in views to limit records to those referred to the subsidiary, giving Marketing visibility into the data they need without accessing additional records.
* **Consider Custom Model-Driven Apps:**
  + If access is still too broad, consider creating a Model-Driven App for the Marketing Department that restricts available entities and records specifically to those Leads referred to subsidiaries and related Opportunities.
  + By tailoring the app, Marketing can focus only on what’s relevant to them, minimizing potential data leaks or access beyond their scope.

**Describe the strategy to migrate existing leads to the cloud while maintaining all**

**relationships and security privileges?**

**Planning and Assessment**

* **Assess the Source Environment:**
  + Identify all Lead-related data and entities, including related records (e.g., Activities, Accounts, Contacts), custom fields, and custom entities.
  + Map out security roles and access permissions tied to Leads in the current system.
* **Data Mapping:**
  + Define mappings between the source data structure and Dynamics 365’s cloud data structure, ensuring all custom fields, entities, and relationships are mapped accurately.
  + Create a data dictionary to document mappings for each field and relationship, specifying customizations to ensure no details are overlooked.

**2. Define Migration Approach and Tools**

* **Select Migration Tools:**
  + Use tools such as Microsoft Data Migration Services, KingswaySoft (SSIS), or Azure Data Factory for complex migrations, as they allow for flexible data handling and relationship management.
  + Dynamics 365 Data Import Wizard can work for simpler migrations, but more complex setups will require ETL tools for control over relationships and incremental data loads.
* **Set Up Staging Environment:**
  + Set up a staging database or environment to perform data transformations and tests. This enables you to refine data and relationships without impacting the production environment.

**3. Preserve Relationships**

* **Define Relationships and Dependencies:**
  + Identify all related records that need to migrate alongside Leads, such as Activities, Opportunities, and Contacts. Note any many-to-many relationships or custom relationships that require careful handling.
* **Maintain Unique Identifiers:**
  + Ensure each record, especially Leads, has a unique identifier in both the source and target systems (like GUIDs). This allows accurate relationship mapping between Leads and related records during migration.
  + Use these unique identifiers to match related records, preserving one-to-many and many-to-many relationships.
* **Order of Migration for Relationship Integrity:**
  + Migrate parent entities first (e.g., Accounts, Contacts) to ensure related Leads can link to these records in the cloud.
  + Migrate Leads next, followed by child entities (e.g., Activities, Tasks) to maintain relationship integrity.
  + Leverage ETL tools with lookup capabilities to ensure each Lead is linked to the correct related entities.

**4. Preserve Security Privileges**

* **Replicate Security Roles and Teams:**
  + In the cloud environment, create or replicate existing security roles and business units to match the on-premises setup.
  + Assign users and teams to these roles, ensuring they have the correct permissions to access Leads and related entities.
* **Migrate Record-Level Access and Ownership:**
  + If record-level permissions exist, use Access Teams or Owner Teams to replicate these in Dynamics 365. This ensures specific Leads remain accessible to designated users or departments.
  + For each Lead, include its Owner field in the migration data to assign it to the correct user or team.
* **Validate Field Security Profiles:**
  + If there are field-level security settings on sensitive fields, recreate Field Security Profiles in Dynamics 365 and assign users or teams accordingly.
  + Ensure each profile restricts or grants access to sensitive data in the cloud environment as it did on-premises.

**5. Data Transformation and Validation**

* **Data Cleansing and Transformation:**
  + Clean and transform data in the staging environment to standardize formats, remove duplicates, and resolve inconsistencies, particularly in fields where data might not align directly with Dynamics 365 cloud requirements.
  + Use scripts or ETL transformations to handle data formatting changes (e.g., date formats, dropdown options).
* **Validation for Data Accuracy:**
  + Perform sample data migration runs in a sandbox environment and validate data for completeness and accuracy, including checking that all relationships and security settings are correctly applied.
  + Test permissions and security roles to ensure users have appropriate access to Leads and related records, just as in the original environment.

**Describe how to migrate the email messages towards the cloud and how to maintain same reliable email management service from Customer Service and on contacts timeline on the cloud?**

**1. Planning and Assessment**

* **Identify Source Email Data and Structure:**
  + Assess where emails are stored (e.g., in a legacy CRM, Exchange Server, or another third-party system).
  + Identify key entities related to email data, such as Contacts, Accounts, Cases, and email activity records, along with any customizations or specific data fields.
* **Define Data Mapping:**
  + Create a data map to link source email fields to Dynamics 365 fields, ensuring metadata (e.g., email subject, body, attachments, sender, recipients, date/time) is accurately mapped.
  + Document mappings and any transformations needed (e.g., converting email statuses or tags to fit Dynamics 365 fields).

**2. Choose the Migration Approach and Tools**

* **Select a Suitable Tool:**
  + Use a data migration tool like Microsoft’s Data Migration Services, KingswaySoft (SSIS), or Azure Data Factory that supports email data handling, relationship mapping, and incremental migrations.
  + For large volumes of email data, consider tools that can handle bulk migrations with support for attachments and activity records.
* **Set Up Staging Environment:**
  + Use a staging database to store, transform, and verify email data before final migration to Dynamics 365.
  + This allows for pre-processing, data cleansing, and testing of the email records without affecting the production system.

**3. Data Transformation and Email Record Preparation**

* **Format Email Data for Dynamics 365:**
  + Transform email fields to align with Dynamics 365's data structure, especially handling complex fields like sender/recipient information, which may need to link to existing Contacts or Accounts.
  + Ensure attachments are stored correctly and linked to email records.
* **Preserve Key Metadata:**
  + Retain metadata such as date/time, sender/recipient IDs, status (sent, received), and correspondence history to maintain the integrity of email communications.
  + Map email statuses from the source system to Dynamics 365 equivalents (e.g., “Read,” “Unread”).

**4. Migrate and Link Email Records to Dynamics 365**

* **Maintain Relationships with Contacts and Cases:**
  + Ensure emails are linked to the correct Contact, Account, or Case records. Use unique identifiers for matching and linking emails to the correct entities.
  + Migrate emails in phases (e.g., by date range or sender) to maintain organization and manage load.
* **Handle Attachments Carefully:**
  + Upload email attachments to Dynamics 365 and link them to their corresponding email records. For large attachments, consider storing them in Azure Blob Storage and linking them via Dynamics 365 to reduce storage costs.
* **Timeline Integration:**
  + Dynamics 365 uses the Activity Timeline on Contact and Case records to show related emails and interactions.
  + During migration, configure each email’s date, sender, and association fields so it appears chronologically on the Contact or Case timeline in the cloud.

**6. What are the tools that can be leveraged to integrate with On-premises API and to synchronize with their core system data which is on-premises? Elaborate on the monitoring strategies that you would implement and how.**

**Tools for Integration and Synchronization with On-premises APIs**

1. **Azure Service Bus**
   * Purpose: Azure Service Bus is a robust messaging service that enables asynchronous communication between on-premises systems and Dynamics 365. It can queue and relay messages, making it ideal for high-volume or complex workflows.
   * How to Use: Service Bus is often set up between cloud applications and on-premises services, acting as an intermediary to send data securely without requiring a constant connection.
   * Advantages: It supports fault tolerance, high throughput, and queuing, which helps manage fluctuating loads and avoid system overloads.
2. **Azure API Management (APIM)**
   * Purpose: Azure API Management allows secure exposure of on-premises APIs to Dynamics 365. It enables the management of API calls, including rate-limiting, security, and caching.
   * How to Use: APIM can create a single, secure endpoint for Dynamics 365 to call on-premises services. It works well in conjunction with Azure VPN or ExpressRoute for secure access.
   * Advantages: APIM provides centralized control over API traffic, logging, security, and version management, making it an excellent choice for regulated or high-security data.
3. **Data Gateway (Power Platform/Data Gateway)**
   * Purpose: The On-premises Data Gateway enables secure data transfer between on-premises data sources and cloud services like Power Automate, Power Apps, and Power BI.
   * How to Use: The gateway connects Dynamics 365 to on-premises databases (e.g., SQL Server) or web APIs, allowing real-time and scheduled data sync.
   * Advantages: It supports a variety of data sources and is fully integrated with the Power Platform, making it ideal for complex integrations without major firewall changes.
4. **Azure Logic Apps**
   * Purpose: Azure Logic Apps is a cloud-based integration platform that can connect cloud and on-premises services using pre-built connectors, including for Dynamics 365 and on-premises APIs.
   * How to Use: Logic Apps can be configured to retrieve or send data between Dynamics 365 and on-premises systems via the Data Gateway or Service Bus.
   * Advantages: It’s highly flexible, provides a visual workflow interface, and integrates well with Azure’s suite of tools for automation and monitoring.
5. **Custom Connectors and APIs**
   * Purpose: When pre-built connectors are insufficient, custom APIs and connectors can be developed to connect Dynamics 365 and on-premises systems.
   * How to Use: Use Azure API Management and custom authentication mechanisms to securely expose custom APIs from the on-premises system.
   * Advantages: Custom connectors allow tailored access to specific features and data structures that may not be available through pre-built tools.
6. **On premise environment contains already some records from different entities that have been teams shared. How will you migrate these records to cloud.**

**1. Assess Current Sharing Configuration**

* Identify Shared Entities: Review which entities have team-based sharing configurations. Ensure you understand the full scope of records, including any custom entities or specific fields with unique access permissions.
* Analyze Team Memberships and Security Roles: List all teams and users with sharing access to these records. Note any specific security roles associated with these teams that grant access to the records.

**2. Extract Records and Sharing Data**

* Export Core Record Data: Use tools like SSIS (SQL Server Integration Services), KingswaySoft, or Scribe to extract data from on-premises entities into a staging database. Include primary fields, relationship fields, and unique record identifiers to maintain data integrity.
* **Export Team-Based Sharing Permissions:**
  + Capture team-to-record sharing relationships, including which team has access to each record and the level of access (e.g., Read, Write).
  + Document this as metadata alongside the core record data, either in the same staging database or a separate file for reference during migration.

**3. Prepare Cloud Environment for Migration**

* **Create Teams and Security Roles:**
  + Replicate the teams and security roles from the on-premises environment in Dynamics 365 in the cloud, ensuring each team has appropriate access levels to match the permissions in the on-premises system.
* **Set Up Team-Based Security:**
  + Configure security roles for each team in Dynamics 365 to grant similar levels of access to the migrated records. Ensure any custom roles are created to match the permissions from the on-premises environment.

**4. Migrate Records to Dynamics 365**

* **Load Records into Dynamics 365:**
  + Using tools like KingswaySoft, Azure Data Factory, or Dynamics 365 Data Import Wizard, import the records from the staging database into the cloud instance.
  + Ensure that all relationships, lookups, and reference data (e.g., Account, Contact associations) are intact and align with the cloud instance’s record identifiers.

**5. Migrate Sharing Permissions**

* **Use API or Custom Code to Apply Sharing Permissions:**
  + In Dynamics 365, sharing permissions are applied at the record level via Access Teams or Owner Teams. Use the Dynamics 365 SDK or Power Automate to programmatically assign teams to records with the correct sharing permissions.
  + For each record, replicate the sharing permissions by adding teams with the necessary privileges (e.g., Read, Write, Append).
  + Custom Workflow or Power Automate: Set up a workflow or flow to loop through each record and reassign the team permissions based on the metadata captured during extraction.

**6. Verify Data Integrity and Permissions**

* **Perform Data Validation:**
  + Check that all records are successfully migrated and retain the correct data values. Verify relationships between records to ensure data consistency.
* **Test Sharing Permissions:**
  + Conduct testing by logging in as a user in each team to confirm that team-based access is accurately replicated in the cloud. Verify that users can access records according to the permissions defined.

**7. Establish Ongoing Synchronization (Optional)**

* If some data will continue to be stored on-premises while the cloud environment is live, consider using Azure Service Bus or Data Gateway to maintain synchronization of new records and updated permissions.

1. **There are personal views and dashboards created by the users on the on-premises environment, how will you migrate these views towards the cloud and what are the tools you will use?**

**1. Identify Personal Views and Dashboards to Migrate**

* **Audit User Customizations**: Use on-premises tools, such as **SQL queries** or the **Dynamics 365 SDK**, to extract a list of personal views and dashboards, noting the associated user IDs and specific configurations.
* **Document the Scope**: Record all views and dashboards that need migration, identifying key components like filters, columns, sorting, charts, and any linked entities. It’s essential to understand these configurations fully to ensure they’re accurately replicated in the cloud.

**2. Export Personal Views and Dashboards**

* **Access Customization Data via SQL or API**:
  + In an on-premises environment, use SQL queries to extract data from tables related to user views and dashboards. For example, personal views are usually stored in the UserQuery table, while dashboards are stored in UserForm.
  + If using the **Dynamics 365 SDK**, retrieve these entities programmatically to capture details about each user’s personal views and dashboards, including layout, filtering criteria, and chart configurations.

**3. Prepare Personal Views and Dashboards for Migration**

* **Transform Data into a Compatible Format**:
  + Convert extracted views and dashboard data into a format that can be used in the cloud environment. Typically, this involves creating an XML or JSON representation of each personal view or dashboard configuration.
  + If possible, map each view’s or dashboard’s attributes to the fields and entities in the cloud instance, ensuring compatibility (for instance, checking for any changes in entity names, fields, or data types).

**4. Migrate Personal Views and Dashboards to Dynamics 365 in the Cloud**

* **Use Configuration Migration Tool (XrmToolBox Plugin)**:
  + XrmToolBox offers plugins like **User View Transfer** or **Dashboard Transfer**, which can simplify the migration of personal views and dashboards. These plugins help to replicate views and dashboards from the on-premises environment to the cloud, applying each customization under the correct user account.
  + **How to Use**: Connect to the on-premises environment, export the personal views and dashboards, then connect to the cloud instance and import these customizations. The plugins handle much of the re-mapping needed for cloud compatibility.
* **Direct Use of Dynamics 365 SDK (Advanced)**:
  + For a more tailored approach, especially if large volumes of customizations are involved, consider writing custom scripts with the **Dynamics 365 SDK**. These scripts can automate the export of views and dashboards and apply them directly to user accounts in the cloud environment.
  + **How to Use**: Use SDK scripts to replicate UserQuery (for views) and UserForm (for dashboards) configurations, assigning each to the correct cloud-based user.

**5. Assign Views and Dashboards to Users**

* Ensure each personal view and dashboard is correctly assigned to the intended user in the cloud environment. This may require **mapping user IDs** from the on-premises system to new user IDs in Dynamics 365 (if they have changed during migration).

**6. Test and Validate the Migration**

* **User Acceptance Testing (UAT)**:
  + Encourage users to log in and confirm that their personal views and dashboards appear correctly, with all configurations intact.
  + Verify that each view’s filters, sorting, columns, and dashboards’ charts and layouts function as expected in the cloud.
* **Validation Against Source System**:
  + Cross-check the migrated views and dashboards with the original on-premises environment to confirm consistency and ensure that key configurations are accurately replicated.

**7. Monitoring and Documentation**

* **Monitor User Feedback**: Track any issues or discrepancies reported by users after migration and address them promptly to ensure a smooth transition.
* **Document the Migration Process**: Create a reference guide or FAQ for users to understand any minor changes in functionality, especially if Dynamics 365 Online handles views or dashboards differently than the on-premises version.

**Tools for Migrating Personal Views and Dashboards**

* **XrmToolBox Plugins**:
  + **User View Transfer**: Migrates personal views created by users, with an option to adjust view properties for the cloud environment.
  + **Dashboard Transfer**: Allows migration of user-specific dashboards, replicating charts and layout configurations for each user.
* **Configuration Migration Tool (Microsoft)**:
  + This tool can assist in moving views and dashboards for managed solutions or customized views, though it has more limitations with strictly personal views and dashboards. It’s still useful if combined with additional customization tools or scripts.
* **Dynamics 365 SDK**:
  + For highly customized or large-scale migrations, custom scripts can be written using the SDK to handle complex mappings and automate the transfer of personal views and dashboards.

**Task 2: Development of C# Plugins &amp; Angular**

**1. Create a plugins project that will record the Decline into a custom table for the specific email address targeted and demonstrate the authentication process that will be implemented.**

****

**2. Create an Angular web application that will be displayed to the contact receiving the invite and choose to decline the event invitation. The Angular web application shall make the API call to update a custom table in Dynamics 365.**

Prerequisites

1. Azure AD App Registration: Register the application in Azure Active Directory (Azure AD) to handle authentication.
2. Dynamics 365 Environment: Access to Dynamics 365 instance with a custom table, DeclineLog, where declines will be recorded

**Steps to Create the Angular Web Application**

**1. Set Up Angular Application**

* **Create a New Angular App:**

ng new DeclineInviteApp

cd DeclineInviteApp

**Install Required Packages:**

* **Install @azure/msal-angular and @azure/msal-browser to handle authentication with Azure AD.**

npm install @azure/msal-angular @azure/msal-browser

* **Set Up MSAL Authentication Configuration**: This enables authentication against Azure AD to access Dynamics 365.

**2. Configure MSAL for Azure AD Authentication**

* Open src/app/app.module.ts and configure MSAL as follows:

import { MsalModule, MsalService, MSAL\_INSTANCE } from '@azure/msal-angular';

import { IPublicClientApplication, PublicClientApplication } from '@azure/msal-browser';

function MSALInstanceFactory(): IPublicClientApplication {

return new PublicClientApplication({

auth: {

clientId: 'YOUR\_CLIENT\_ID', // Azure AD Application (client) ID

authority: 'https://login.microsoftonline.com/YOUR\_TENANT\_ID', // Tenant ID

redirectUri: 'http://localhost:4200',

},

cache: {

cacheLocation: 'localStorage',

storeAuthStateInCookie: true,

},

});

}

@NgModule({

declarations: [/\* components here \*/],

imports: [

BrowserModule,

MsalModule.forRoot({

auth: {

clientId: 'YOUR\_CLIENT\_ID', // Application (client) ID from Azure AD

authority: 'https://login.microsoftonline.com/YOUR\_TENANT\_ID',

redirectUri: 'http://localhost:4200',

},

}),

],

providers: [

{

provide: MSAL\_INSTANCE,

useFactory: MSALInstanceFactory,

},

MsalService,

],

bootstrap: [AppComponent],

})

export class AppModule { }

**3. Create Decline Invitation Component**

* Generate a new component for handling the decline action:

ng generate component DeclineInvite

Implement the UI in decline-invite.component.html:

<div class="decline-invite">

<h1>Decline Event Invitation</h1>

<p>Are you sure you want to decline this invitation?</p>

<button (click)="declineEvent()">Decline</button>

</div>

Define the component logic in decline-invite.component.ts:

import { Component } from '@angular/core';

import { HttpClient, HttpHeaders } from '@angular/common/http';

import { MsalService } from '@azure/msal-angular';

@Component({

selector: 'app-decline-invite',

templateUrl: './decline-invite.component.html',

styleUrls: ['./decline-invite.component.css']

})

export class DeclineInviteComponent {

constructor(private http: HttpClient, private authService: MsalService) {}

declineEvent() {

// Authenticate and acquire token

this.authService.instance.acquireTokenPopup({

scopes: ['https://YOUR\_ORG.crm.dynamics.com/.default'],

}).then(response => {

const token = response.accessToken;

// Prepare data payload

const declineData = {

email: "contact\_email@example.com", // Replace with dynamic email or passed data

declinedate: new Date(),

reason: "User declined event invitation"

};

// Call Dynamics 365 API to create a DeclineLog record

this.http.post('https://YOUR\_ORG.api.crm.dynamics.com/api/data/v9.0/new\_declinelogs',

declineData,

{

headers: new HttpHeaders({

'Authorization': `Bearer ${token}`,

'Content-Type': 'application/json'

})

}

).subscribe({

next: (response) => console.log('Decline logged:', response),

error: (error) => console.error('Error logging decline:', error)

});

});

}

}

**4. Setting Up API Permissions in Azure AD**

* **Register the Angular app** in Azure AD and assign API permissions for Dynamics 365.
* Grant the app **delegated permissions** to access Dynamics 365, such as user\_impersonation for CRM.
* **Expose an API** and add Dynamics 365 as a resource.

**5. Testing the Application**

* Run the Angular app using:

ng serve

 Open http://localhost:4200 in a browser.

 Test the "Decline" button to ensure it triggers the API call and logs the decline event in Dynamics 365.

**6. Deployment**

* Deploy the Angular app to a web server (e.g., Azure App Service) and ensure the redirect URI matches the URI registered in Azure AD.
* Update the Azure AD app registration with the deployed URL as the new **redirect URI**.

1. **Define an alternative technical approach that you will use to capture the accept and decline from the email.**

**Steps for Alternative Approach**

**1. Send an Email with Accept/Decline Options**

* Use Dynamics 365 or a third-party email service to send out event invitations.
* The email can include Accept and Decline buttons or links that can be tracked.
* These buttons or links should be designed to trigger an automated process upon being clicked. A simple way to achieve this is by embedding links with unique identifiers (like a GUID or event ID) that allow you to match the response to the corresponding Dynamics 365 record.

Example:

* "Accept" button link: https://yourdomain.com/accept?eventID={EventGUID}&email={ContactEmail}
* "Decline" button link: https://yourdomain.com/decline?eventID={EventGUID}&email={ContactEmail}

**2. Process Responses via Webhook or Power Automate**

* When a recipient clicks on the Accept or Decline link, the web service (e.g., a simple API endpoint) will capture the response.
* You can configure a Webhook to capture the click events (if using a custom email solution) or use Power Automate for a more out-of-the-box solution.

**3. Using Power Automate for Response Handling**

Power Automate can be configured to automatically process the accept and decline responses via a combination of the following triggers and actions:

* Trigger: "When an HTTP request is received": Power Automate can be triggered by an HTTP request when the recipient clicks the Accept/Decline button. You can send the necessary data (event ID, contact email, and decision) as part of the query string or body of the HTTP request.
* Action: "Parse JSON": Parse the incoming data (event ID, email, accept/decline decision) to extract the relevant details.
* Action: "Dynamics 365 – Update Record": Use Power Automate to call Dynamics 365's Web API to update the relevant record (e.g., Event or Invitation record) based on the event ID, storing the recipient’s response (Accepted/Declined) in the custom table (DeclineLog or EventResponse).

**Example Power Automate Flow:**

1. Trigger: HTTP request received (with parameters like event ID, email, response).
2. Parse JSON: Extract event ID, contact email, and response (Accepted or Declined).
3. Condition: Check if the response is "Accepted" or "Declined".
4. Action: Use the "Dynamics 365" connector to create or update a record in the custom DeclineLog table or another related entity (like EventResponse).

**4. Implement the Email Response Handling System**

If you want the system to directly process accept and decline actions in email responses, you can use the following methods:

* Email Parsing: Use Microsoft Power Automate or Azure Logic Apps to parse incoming emails and detect keywords such as "accept" or "decline".
  + Power Automate’s Email Trigger action can monitor incoming emails, check if they contain specific phrases (like "Accepted" or "Declined"), and then automatically update the DeclineLog entity in Dynamics 365.
  + For example, create a flow that triggers when a new email is received in the inbox, checks for keywords like "Decline" in the subject or body, and then records the response into the custom entity.
* Exchange Web Services (EWS): If you're using a local Exchange server, EWS can be used to directly interact with incoming emails. You can set up a service to listen to these incoming responses, parse the email body for keywords, and update Dynamics 365 accordingly.

**5. Create Custom Workflow in Dynamics 365**

* Create a Custom Workflow in Dynamics 365 that listens for changes to the status of a related record (like Event or Invitation) and updates the DeclineLog table accordingly.
* For example, once a decline or acceptance response is captured via email, trigger a workflow that updates a field like InvitationStatus and creates an entry in the DeclineLog table, recording the contact’s response and reason.

1. **Create a small and basic Angular notification system which will prompt the user’s attention to tasks due, pending records with FetchXML and details of how you will publish same in Dynamics 365 Form.**

**1. Set Up Angular Application**

First, create a basic Angular application:

ng new NotificationSystem

cd NotificationSystem

**Install required packages for MSAL (to authenticate with Azure AD):**

npm install @azure/msal-angular @azure/msal-browser

2. Angular Authentication Setup (MSAL)

We will use **MSAL** (Microsoft Authentication Library) to authenticate and authorize users to access Dynamics 365 data.

import { MsalModule, MsalService, MSAL\_INSTANCE } from '@azure/msal-angular';

import { IPublicClientApplication, PublicClientApplication } from '@azure/msal-browser';

function MSALInstanceFactory(): IPublicClientApplication {

return new PublicClientApplication({

auth: {

clientId: 'YOUR\_CLIENT\_ID', // Replace with your Azure AD Application (client) ID

authority: 'https://login.microsoftonline.com/YOUR\_TENANT\_ID', // Tenant ID

redirectUri: 'http://localhost:4200', // Your redirect URI

},

cache: {

cacheLocation: 'localStorage',

storeAuthStateInCookie: true,

},

});

}

@NgModule({

declarations: [/\* components here \*/],

imports: [

BrowserModule,

MsalModule.forRoot({

auth: {

clientId: 'YOUR\_CLIENT\_ID',

authority: 'https://login.microsoftonline.com/YOUR\_TENANT\_ID',

redirectUri: 'http://localhost:4200',

},

}),

],

providers: [

{

provide: MSAL\_INSTANCE,

useFactory: MSALInstanceFactory,

},

MsalService,

],

bootstrap: [AppComponent],

})

export class AppModule { }

**3. Notification System Component in Angular**

Create a new component for the notification system.

ng generate component NotificationSystem

This component will display notifications to the user based on the data retrieved from Dynamics 365.

**3.1 Component HTML (notification-system.component.html)**

<div class="notification-system">

<h2>Pending Tasks/Records</h2>

<ul>

<li \*ngFor="let notification of notifications">

<div>{{ notification.subject }} - Due: {{ notification.dueDate | date }}</div>

</li>

</ul>

</div>

3.2 **Component TypeScript (notification-system.component.ts)**

import { Component, OnInit } from '@angular/core';

import { MsalService } from '@azure/msal-angular';

import { HttpClient, HttpHeaders } from '@angular/common/http';

import { Observable } from 'rxjs';

@Component({

selector: 'app-notification-system',

templateUrl: './notification-system.component.html',

styleUrls: ['./notification-system.component.css']

})

export class NotificationSystemComponent implements OnInit {

notifications: any[] = [];

constructor(private msalService: MsalService, private http: HttpClient) {}

ngOnInit(): void {

this.fetchNotifications();

}

// Function to fetch pending tasks/records from Dynamics 365 using FetchXML

fetchNotifications() {

// Step 1: Acquire Token

this.msalService.instance

.acquireTokenPopup({

scopes: ['https://YOUR\_ORG.crm.dynamics.com/.default'],

})

.then((response) => {

const token = response.accessToken;

// Step 2: Fetch Notifications from Dynamics 365 using FetchXML

const fetchXml = `

<fetch>

<entity name="task">

<attribute name="subject" />

<attribute name="scheduledstart" />

<attribute name="scheduledend" />

<filter>

<condition attribute="scheduledstart" operator="on-or-after" value="2024-11-08" />

<condition attribute="statuscode" operator="neq" value="5" /> <!-- Status != Completed -->

</filter>

</entity>

</fetch>`;

// Step 3: Make HTTP call to Dynamics 365 Web API

this.http

.post(

'https://YOUR\_ORG.api.crm.dynamics.com/api/data/v9.0/tasks',

{

fetchXml: fetchXml,

},

{

headers: new HttpHeaders({

'Authorization': `Bearer ${token}`,

'Content-Type': 'application/json',

}),

}

)

.subscribe((data: any) => {

this.notifications = data.value; // Store the retrieved notifications

});

});

}

}

Creating a small and basic Angular notification system that prompts the user's attention to tasks, pending records, or other due items in **Dynamics 365** using **FetchXML** is a practical way to engage users and help them stay on top of important activities. The system will leverage **FetchXML** to retrieve the necessary data from Dynamics 365 and display notifications on the Angular frontend. These notifications will then be embedded or published on a Dynamics 365 form to show task-related updates.

**Steps to Build the Notification System**

**1. Set Up Angular Application**

First, create a basic Angular application:

bash

Copy code

ng new NotificationSystem

cd NotificationSystem

Install required packages for MSAL (to authenticate with Azure AD):

bash

Copy code

npm install @azure/msal-angular @azure/msal-browser

**2. Angular Authentication Setup (MSAL)**

We will use **MSAL** (Microsoft Authentication Library) to authenticate and authorize users to access Dynamics 365 data.

**2.1 Configure MSAL in app.module.ts:**

typescript

Copy code

import { MsalModule, MsalService, MSAL\_INSTANCE } from '@azure/msal-angular';

import { IPublicClientApplication, PublicClientApplication } from '@azure/msal-browser';

function MSALInstanceFactory(): IPublicClientApplication {

return new PublicClientApplication({

auth: {

clientId: 'YOUR\_CLIENT\_ID', // Replace with your Azure AD Application (client) ID

authority: 'https://login.microsoftonline.com/YOUR\_TENANT\_ID', // Tenant ID

redirectUri: 'http://localhost:4200', // Your redirect URI

},

cache: {

cacheLocation: 'localStorage',

storeAuthStateInCookie: true,

},

});

}

@NgModule({

declarations: [/\* components here \*/],

imports: [

BrowserModule,

MsalModule.forRoot({

auth: {

clientId: 'YOUR\_CLIENT\_ID',

authority: 'https://login.microsoftonline.com/YOUR\_TENANT\_ID',

redirectUri: 'http://localhost:4200',

},

}),

],

providers: [

{

provide: MSAL\_INSTANCE,

useFactory: MSALInstanceFactory,

},

MsalService,

],

bootstrap: [AppComponent],

})

export class AppModule { }

**3. Notification System Component in Angular**

Create a new component for the notification system.

bash

Copy code

ng generate component NotificationSystem

This component will display notifications to the user based on the data retrieved from Dynamics 365.

**3.1 Component HTML (notification-system.component.html)**

html

Copy code

<div class="notification-system">

<h2>Pending Tasks/Records</h2>

<ul>

<li \*ngFor="let notification of notifications">

<div>{{ notification.subject }} - Due: {{ notification.dueDate | date }}</div>

</li>

</ul>

</div>

**3.2 Component TypeScript (notification-system.component.ts)**

typescript

Copy code

import { Component, OnInit } from '@angular/core';

import { MsalService } from '@azure/msal-angular';

import { HttpClient, HttpHeaders } from '@angular/common/http';

import { Observable } from 'rxjs';

@Component({

selector: 'app-notification-system',

templateUrl: './notification-system.component.html',

styleUrls: ['./notification-system.component.css']

})

export class NotificationSystemComponent implements OnInit {

notifications: any[] = [];

constructor(private msalService: MsalService, private http: HttpClient) {}

ngOnInit(): void {

this.fetchNotifications();

}

// Function to fetch pending tasks/records from Dynamics 365 using FetchXML

fetchNotifications() {

// Step 1: Acquire Token

this.msalService.instance

.acquireTokenPopup({

scopes: ['https://YOUR\_ORG.crm.dynamics.com/.default'],

})

.then((response) => {

const token = response.accessToken;

// Step 2: Fetch Notifications from Dynamics 365 using FetchXML

const fetchXml = `

<fetch>

<entity name="task">

<attribute name="subject" />

<attribute name="scheduledstart" />

<attribute name="scheduledend" />

<filter>

<condition attribute="scheduledstart" operator="on-or-after" value="2024-11-08" />

<condition attribute="statuscode" operator="neq" value="5" /> <!-- Status != Completed -->

</filter>

</entity>

</fetch>`;

// Step 3: Make HTTP call to Dynamics 365 Web API

this.http

.post(

'https://YOUR\_ORG.api.crm.dynamics.com/api/data/v9.0/tasks',

{

fetchXml: fetchXml,

},

{

headers: new HttpHeaders({

'Authorization': `Bearer ${token}`,

'Content-Type': 'application/json',

}),

}

)

.subscribe((data: any) => {

this.notifications = data.value; // Store the retrieved notifications

});

});

}

}

**Explanation:**

1. **Authentication**: We use MSAL to authenticate the user and acquire an access token that grants us permission to call Dynamics 365's Web API.
2. **FetchXML Query**: The fetchNotifications() function constructs a **FetchXML** query to retrieve tasks where the start date is after a specified date and the task status is not "Completed".
3. **HTTP Request**: The HttpClient makes a POST request to the Dynamics 365 Web API, passing the FetchXML query. The response is an array of tasks, which are displayed as notifications.

**Task 3: Development on Azure Data Factory**

**1. Define the steps that you will undertake and deploy on Azure API Management to connect to an on-premises get and post APIs.**

**1. Set Up Azure API Management (APIM)**

If you haven't already set up an instance of Azure API Management (APIM), you need to create one.

**1.1 Create Azure API Management Instance**

* Go to the Azure Portal.
* Search for API Management and select Create.
* Fill in the required details like the name of the APIM instance, resource group, region, and organization name.
* Choose the appropriate pricing tier based on your needs (Developer, Basic, Standard, Premium, etc.).
* Click Create to provision the instance.

**1.2 Create API in Azure APIM**

Once the APIM instance is ready, create the APIs that will represent the on-premises services you want to expose.

* In the Azure Portal, go to your APIM instance.
* Under the APIs section, click + Add API.
* Choose the appropriate option (e.g., Blank API, OpenAPI, WSDL, etc.) depending on how your on-premises APIs are defined.
* Define the API URL for the external facing API.

**2. Set Up Hybrid Connectivity (On-Premises and Azure)**

Azure APIM needs to securely communicate with your on-premises API. This can be achieved using Azure API Management’s On-Premises Data Gateway or a Self-hosted Gateway.

2.1 Use the Self-Hosted Gateway

Azure provides a Self-hosted Gateway that acts as a proxy between your Azure APIM and on-premises resources. This allows the traffic to flow securely between the cloud and on-premises systems without opening inbound ports on your internal firewall.

* **Step 1: Download and install the Self-hosted Gateway on your on-premises server.**
  + In your APIM instance, under the Self-hosted Gateway section, click Download and follow the installation instructions.
  + The installation will require an authorization key to link the gateway to your APIM instance.
* **Step 2: Configure the gateway on the on-premises server.**
  + After installation, configure the gateway to authenticate with the Azure APIM instance using the API Management subscription key and gateway authentication key.
* **Step 3: Link the Self-hosted Gateway to your APIM instance.**
  + Once installed and configured, go back to the Azure Portal and link the on-premises Self-hosted Gateway to your Azure APIM instance.

**2.2 Verify the Connectivity**

* Ensure the Self-hosted Gateway is operational by checking its status in the Azure Portal under the Self-hosted Gateway tab.
* If successful, the gateway can route API calls from APIM to your on-premises APIs securely.

**3. Define API Operations (GET/POST)**

Once you’ve established a secure connection to your on-premises APIs, you need to define operations for GET and POST calls within Azure APIM.

**3.1 Create API Operations**

* Go to your APIM instance in the Azure Portal.
* Under APIs, select the API you created earlier.
* **Define the operations for the GET and POST requests:**
  + **GET Operation:**
    - Click Add Operation and choose GET as the method.
    - Define the URL path, query parameters, and backend service URL pointing to the on-premises API (e.g., http://<on-prem-server>/api/endpoint).
  + **POST Operation:**
    - Similarly, click Add Operation and choose POST as the method.
    - Define the URL path and request body schema (if needed).
    - Specify the backend POST URL and any headers or authorization required for your on-prem API.

**3.2 Define Request and Response Policies**

Azure API Management allows you to define various policies for processing requests and responses. This step involves adding policies for logging, transformations, rate limiting, etc.

* **Under the Design tab for your API, you can:**
  + Request validation: Ensure only valid data is forwarded to your on-premises API (e.g., validating request headers, parameters, or body).
  + Rate limiting: Control the number of requests that can be made to your on-prem API.
  + Logging: Add diagnostic policies to log requests and responses for troubleshooting.
  + Request transformations: If needed, transform the request body (for example, convert XML to JSON) before sending it to the on-prem API.

**4. Test the API**

Once the API operations are configured in Azure APIM, you need to test whether the requests are correctly routed to your on-premises API.

* Use the Test feature in Azure APIM to send GET and POST requests.
  + Navigate to your API in the Azure portal.
  + Under Test, choose the operation (GET or POST), fill in the required parameters (e.g., query parameters for GET, body content for POST), and click Send.
  + Verify that the response matches what the on-premises API would return.

**5. Deploy and Secure API**

**5.1 Deploy API to Production**

* Once tested, you can publish the API to the production environment in APIM by adding it to the Products section.
* A product in APIM allows you to group related APIs and expose them to consumers (e.g., internal applications, partners).

**2. Discuss how you will implement an ETL data pipeline on Azure Data Factory to lookup for:**

**i. a record from another tables of Dynamics 365 and update the source data prior to**

**update to dynamics 365.**

**ii. an appropriate GUID from another Dynamics 365 and update the source data.**

**Support same with some snapshots.**

To implement an ETL (Extract, Transform, Load) data pipeline using Azure Data Factory (ADF) for interacting with Dynamics 365 and updating source data, you will follow a series of steps to integrate data and perform the required lookups and updates. In this scenario, you are dealing with two tasks:

1. Looking up a record from another table of Dynamics 365 and updating the source data before sending it back to Dynamics 365.
2. Looking up an appropriate GUID from another record in Dynamics 365 and updating the source data.
3. **Create and demonstrate an Azure Data pipeline to create &amp; update existing contacts record from the core system to dynamics 365 based on below ERD. Discuss why you have been using the component.**

**Step 1: Define the Data Flow and Components Needed**

To implement the pipeline for creating and updating contacts in **Dynamics 365**, the key components that will be used are:

1. **Linked Services**:
   * A **Linked Service for the Core System** (where contact data originates) to connect Azure Data Factory to the source system (database, flat files, etc.).
   * A **Linked Service for Dynamics 365** (using either **OData** or **REST API**) to connect ADF to the Dynamics 365 environment.
2. **Datasets**:
   * A **Source Dataset** that defines the structure of the contact data (e.g., SQL, CSV, or JSON schema).
   * A **Sink Dataset** that defines the destination (Dynamics 365 table for Contact) and allows ADF to write data into Dynamics 365.
3. **Activities**:
   * **Copy Activity** to extract data from the source system and load it into an interim storage or directly into Dynamics 365.
   * **Lookup Activity** to check whether the contact already exists in Dynamics 365 based on some key (e.g., **email address** or **phone number**).
   * **Conditional Split** to decide whether the data should be **inserted** (if the record does not exist) or **updated** (if the record already exists).
4. **Create an Azure Data Pipeline to migrate Leads &amp; tasks from on-premises Dynamics 365. One key requirement is to retain the creationOn, creationBy and modifiedBy values and also maintain the relationship with existing contacts lookup and Team sharing on the records.**

**Key Considerations for the Migration:**

* Data Extraction: Extract Leads and Tasks from your on-premises Dynamics 365 using FetchXML or OData.
* Data Transformation: Ensure that all fields, including the system fields like createdOn, createdBy, and modifiedBy, are mapped properly to the target environment.
* Data Loading: Load the data into Dynamics 365 (Cloud) while retaining field values and relationships.
* Relationship Maintenance: Ensure that Lookups (e.g., Contacts, Users, and Teams) are correctly handled during the migration.
* Team Sharing: Ensure that the Team sharing of records is replicated correctly in the cloud environment.