What are the advantages of this project?

As we know, Salesforce operates on a multi-tenant architecture, which means that multiple organizations or tenants share the same underlying infrastructure and resources, including storage. The amount of storage you get in a Salesforce production org depends on your specific Salesforce edition and any additional storage purchases or allocations you may have.

For more information, please refer to the following article: Salesforce Storage Overview

- **Salesforce Essentials** typically provides 1 GB of data storage.
- **Salesforce Professional Edition** offers 1 GB of data storage per user.
- **Salesforce Enterprise Edition** usually includes 1 GB of data storage per user with a minimum of 20 users or a total minimum allocation of 20 GB.
- **Salesforce Unlimited Edition** generally provides 1 GB of data storage per user with a minimum of 20 users or a total minimum allocation of 20 GB.
- **Salesforce Performance Edition** typically includes 1 GB of data storage per user with a minimum of 20 users or a total minimum allocation of 20 GB.

Since we receive only limited storage from Salesforce, it is crucial to have a data archival strategy when planning for the Salesforce project. This strategy will help us prevent the situation of buying extra storage from Salesforce, which might cost $125/month for 500MB of additional data storage. For more information, you can refer to this link.

This project is developed using Asynchronous Apex Batch Class, which will delete converted lead records from Salesforce that are 180 days old. Once a lead is converted, we create Account, Contact, and Opportunity records in Salesforce, making the converted lead record data unnecessary in Salesforce.

It is essential to delete the records from Salesforce on a periodic basis since every record we create in Salesforce counts toward the storage we receive from Salesforce.

NOTE: As part of the Data Archival strategy, we need to first store the data in 3rd party databases such as DataLake, Metabase, and others. This will help us store the data in an external database and free up space in Salesforce storage.

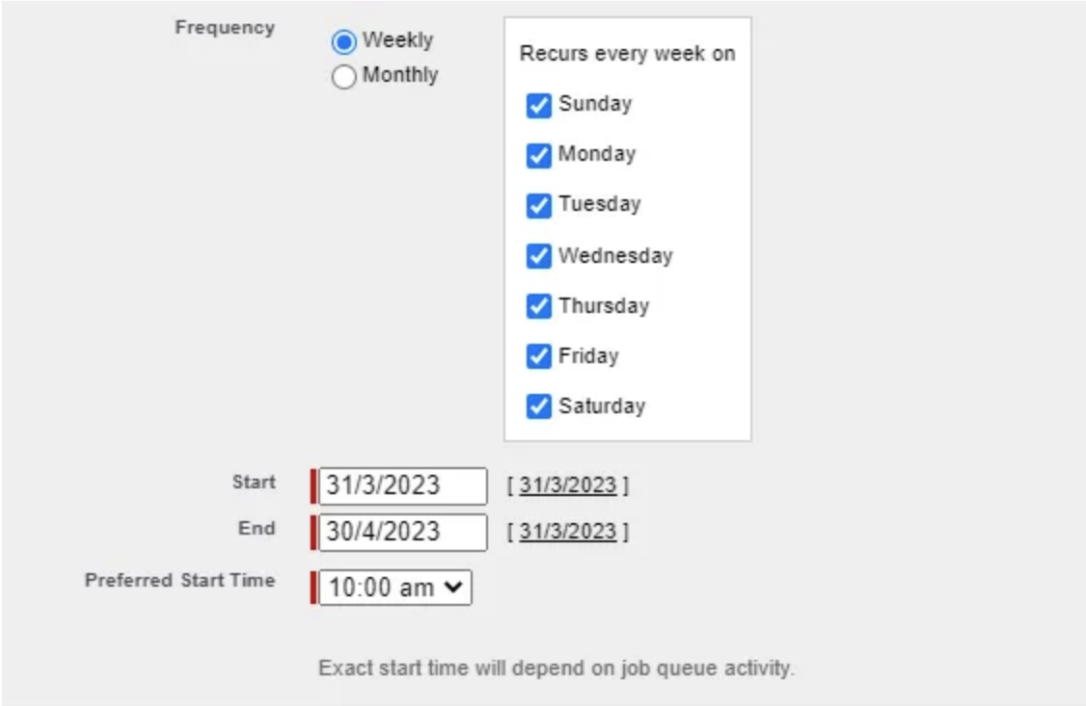
How is the code structure?

I am using Asynchronous Apex to write a Batch Class to delete records from the Salesforce org. Batch classes in Salesforce are executed asynchronously, which means they run in the background and do not block the user interface. This makes them suitable for handling resource-intensive tasks without impacting system performance.

First, we need to implement the batch class with the **Database.Batchable** interface. To implement the **Database.Batchable** interface, your class must define three key methods: **start**, **execute**, and **finish**.

- **start**: This method identifies the records to be processed and returns a Database.QueryLocator or an iterable collection of records. It defines the initial set of records to be processed.
- **execute**: This method processes a batch of records. It receives the records from the start method and executes your custom logic on them. You can specify how many records are processed in each batch using the batchSize attribute.
- **finish**: This method is called after all batches have been executed. It is typically used for post-processing tasks, such as sending notifications or logging results.

Once the batch class is ready, we need to schedule it to run every day at midnight when no operations are performed in Salesforce. This allows it to delete the records from Salesforce without interrupting any activity. You can schedule the batch class through the UI.



What is not good about your project?

I believe there are two critical aspects missing from the batch class:

- **Error Handling**: Error handling is critically important in batch classes in Salesforce. Effective error handling ensures that your batch jobs can handle exceptions and unexpected situations gracefully, maintain data integrity, and provide meaningful feedback to administrators and users.
- **Sending Email When Records Fail to Delete**: Currently, we are only sending an email to the system administrator once the batch job is finished. There might be a situation where the batch job is complete, but the records are not deleted. This can happen due to different reasons in Salesforce, for example, when the record we are trying to delete has dependencies on another record. Therefore, it is essential to send an email to the admin with details of failed records.

What would you do better next time?

I will ensure that I have proper error handling in place while writing the code. Additionally, I will implement sending an email to the system admin when records fail to be deleted so that the admin can debug and resolve any issues.

Error Handling:

```
global void execute(Database.BatchableContext BC, List<SObject> scope) {
        try {
            // Delete the converted lead records
            List<Lead> convertedLeadsToDelete = (List<Lead>)scope;
            delete convertedLeadsToDelete;
        } catch (Exception e) {
            // Handle any exceptions during record deletion
            for (Lead lead : (List<Lead>)scope) {
                failedRecordIds.add(lead.Id);
            }
        }
    }
```

Sending an Email to the system admin once the record is failed to delete.

```
// If there were failed record deletions, include them in the email
        if (!failedRecordIds.isEmpty()) {
            body += '\n\nThe following record(s) failed to delete:\n';
            for (Id failedId : failedRecordIds) {
                body += failedId + '\n';
            }
        }
```

Thinking outside of the box?

There might be situations where we need to delete records from multiple objects in Salesforce. Writing separate batch classes for each object may not be a scalable solution. To handle this, we can create custom metadata where we store the objects API names from which we need to delete records. Then, we can pass the objects API names to the batch class one by one to delete the records from those objects, ensuring a more efficient and scalable solution.

```
global Database.QueryLocator start(Database.BatchableContext BC) {
    // Query the Custom Metadata records based on your criteria
    Object_Configuration__mdt config = [SELECT Object_API_Name__c FROM Object_Configuration__mdt WHERE Id = 'your_custom_metadata_record_id' LIMIT 1];
    String objectApiNameFromMetadata = config.Object_API_Name__c;
    // Construct a dynamic query based on the object API name
    String query = 'SELECT Id FROM ' + objectApiNameFromMetadata + ' WHERE CreatedDate < LAST_N_DAYS:30';
    return Database.getQueryLocator(query);
}
```