What are the advantages of this project?

In Salesforce, we can create relationship fields on objects such as Lookup and Master-detail.

Both Lookup and Master-Detail are types of relationships in Salesforce that define how two objects (standard or custom) relate to each other. They determine how data can be linked or associated between these objects. Here's an explanation of Lookup and Master-Detail relationships in Salesforce:

Lookup Relationship:

- **Nature**: Lookup relationships are loosely coupled relationships between two objects. They are used when you want to associate one record with another, but the relationship is not as tightly bound as in a Master-Detail relationship.
- **Ownership**: In a Lookup relationship, the child record does not have an owner field directly linking it to a parent record. Each record has its owner, and there is no "cascade delete" behavior. Deleting a parent record does not necessarily delete the associated child records.
- **Cascade Delete**: Deleting a parent record does not automatically delete its related child records. The child records remain intact even if their parent is deleted.
- **Roll-Up Summary Fields**: You cannot create Roll-Up Summary Fields on objects with Lookup relationships.
- **Many-to-Many Relationships**: Lookup relationships can be used to create many-to-many relationships between objects through a junction object.
- **Cross-Object Formula Fields**: You can reference fields from the related parent object in cross-object formula fields.
- **Limitations**: Lookup relationships have certain limitations, such as a maximum of 25 Lookup fields per object and a limit on the number of relationships per object.

Master-Detail Relationship:

- **Nature**: Master-Detail relationships are tightly coupled relationships between two objects. They are used when you want to create a parent-child relationship where the child records are dependent on the existence of the parent record.
- **Ownership**: In a Master-Detail relationship, the child records are owned by the parent record. The owner of the child record is determined by the owner of the parent record.
- **Cascade Delete**: Deleting a parent record in a Master-Detail relationship automatically deletes its related child records. This is known as "cascade delete" behavior, and it helps maintain data integrity.
- **Roll-Up Summary Fields**: You can create Roll-Up Summary Fields on the master object to calculate values from related child records.
- **Aggregations**: Master-Detail relationships support aggregations like COUNT, SUM, MIN, and MAX in Roll-Up Summary Fields.

- **Security and Sharing**: Security and sharing settings are often inherited from the parent to the child records in a Master-Detail relationship.
- **Cross-Object Formula Fields**: You can reference fields from the related parent object in cross-object formula fields.
- **Limitations**: Master-Detail relationships also have certain limitations, such as a maximum of two Master-Detail relationships per object and restrictions on changing a Lookup to a Master-Detail relationship once data exists.

Since Salesforce limits Roll-Up Summary Fields on lookup relationship objects, which prevents us from performing aggregations such as COUNT, SUM, MIN, and MAX on lookup objects, I've written this trigger code. It calculates the sum of the amount from child records and displays it on the parent record. This code helps us provide users with an accumulated summary of the child records associated with the parent record.

How is the code structure?

I am using the Trigger Factory Framework, an approach in Salesforce to implement triggers in an organized manner. This framework separates concerns and manages trigger logic effectively, leading to clean and maintainable code.

- **Trigger Handler Classes**: In the Trigger Factory framework, separate Apex classes are created to handle trigger logic for each object. These classes are often referred to as "trigger handler" classes.
- **Trigger Factory Class**: A central "Trigger Factory" class is created, responsible for invoking the appropriate trigger handler class based on the trigger context (e.g., before insert, after update).

Using the Trigger Factory pattern simplifies testing, promotes code reusability, enhances readability, and efficiently handles bulk data operations.

What is not good about your project?

I believe the following two points are missing from the class:

- **Handling Delete and Undelete Operations**: Currently, the code triggers based on insert and update events. However, there may be situations where child records are deleted from the parent record, requiring a deduction of the amount from the parent record. Also, undelete events, where records are restored from the recycle bin, should update the amount on the parent record.
- **Recursive Checks in the Code**: The current code lacks a recursive check, which is crucial. Recursive checks prevent the code from running in a loop and hitting the Apex class error. Implementing a recursive check ensures that the trigger runs only once in a transaction.

What would you do better next time?

I will ensure that a recursive check is added to the code. Adding a recursive check in a trigger is essential to prevent trigger recursion, which can occur when a trigger update causes the same trigger to fire again, leading to an infinite loop. To implement a recursive check, a static variable or a static set can be used to keep track of processed records within the trigger execution.

```
public static void handle(String objectName, List<sObject> records, TriggerOperation operation) {
        if (objectName == 'ChildObject__c') {
            if (operation == TriggerOperation.AFTER_INSERT || operation == TriggerOperation.AFTER_UPDATE) {
                // Check for recursion using a static set
                Set<Id> processedRecordIds = new Set<Id>();
                for (ChildObject__c record : (List<ChildObject__c>) records) {
                    if (!processedRecordIds.contains(record.Id)) {
                        ParentObjectTriggerHandler.handleChildRecords(record);
                        processedRecordIds.add(record.Id);
                    }
                }
            }
        }
    }
```

A static set called **processedRecordIds** is created to keep track of the child records processed during the trigger execution. Before processing each child record, it is checked if its Id is already in the **processedRecordIds** set. If it is, processing that record is skipped to prevent recursion; if not, processing the record proceeds.

Thinking outside of the box?

Moving this trigger logic to Lightning Flows, a declarative tool, can be beneficial. With Lightning Flows, you don't need to write code; you can accomplish tasks using a drag-and-drop interface.

Advantages of moving the logic to Lightning Flows:

- Ease of use and understanding of the logic.
- Easy modification in production, as trigger code cannot be modified in production directly.
- All events are predefined by Salesforce; you need to drag and drop on the screen.
- Easy debugging of the flow by inputting the record's value.

Disadvantages:

- Unable to perform undelete operations in flows.
- Complex logic debugging in flows is limited.