1. CQRS Stands for Command and Query Responsibility Segregation pattern that basically separate the logic between read and Insert/Update/Delete for the data store. As In our application we are using separate logic for Command Controller and separate logic for Query controller. So In simple words I can say we are separating or dividing the logic between Query handlers and the Command handlers for that we are using mediatr pattern in .net . If I am talking about the traditional webapi they mainly do the same things that means CRUD OPERATION. In our case we are using cloud based scalable application where we need to decouple interfaces for the data access layer. As we know when we add multiple features in future the application size also grow exponentially, so that the dependencies also grow equally. Now we need to inject multiple interfaces for decouple, so that complexity also increases at the same time. In our to solve that we are using CQRS pattern. So here the command means Write and Query means Read models so that we can scale up read and write opration independently. Here for that we are using 2 different piece. In traditional architecture the same data model is used to query and update a database while here we are using two different handlers. In tax based our application we have lots of read operations as compared to write so when we need to scale up we can do that also.(We want to divide the workload). On the other side while writing code the model may implement complex validation and business logic. As a result, sometimes we can end up with an overly complex model that does too much. So to solve the mismatch between read and write operations we are end of here with cqrs event driven pattern. So for the security and we want developers can work simultaneously asynchrous way also.(permission also).

So the solution here is while calling api we are using task based command rather then data centric. and as i explained in the above we are using asynchrouns processing of data. It simplifies the design and implementation.

Some of the benefits what i know -

1. Independent scaling(up or down)

2. It optimzes the data access layer

3. Security and separation of concern between different module and data

In our Scenarios where one team of developers can focus on the complex domain model that is part of the write model, and another team can focus on the read model and the user interfaces.

CQRS allows us to define commands with enough granularity to minimize merge conflicts in git repos at the domain level, and conflicts that do arise can be merged by the command.

We are using some Scenarios where performance of data reads we must be fine-tuned separately from performance of data writes.

Some of the implementation issue also there like Messaging failure while publish new message inside background tasks like azure webjob or functions.

2. The Saga pattern is distributed implementation, transactions by designing them as a sequence of local transactions I can say. Although they focused on a centralized system, Sagas have been proposed for updating data in multiple services in a Microservices Architecture without using any distributed transactions.

For example of Trip booking system holiday:

What I think Implementing the Saga means that the long lived transaction is split into three local transactions: Save Trip Information, Booking Hotel, and Booking Flight. The Save Trip Information transaction is executed upon a trip booking request locally by the Travel Service to initiate the Saga and ensure the Durability. Here It is part of the Begin Saga step and does not require communication with another service api. If the Booking Hotel transaction and the Booking Flight transaction are executed locally in the Hotel Service and Flight Service. Untill transaction completed here each local transaction updates only the data within one service and then triggers the next one. If one transaction fails, the Saga aborts.(for each local transaction).

So I can say the saga pattern is the way to manage the data consistency across microservices in distributed transaction scenarios like above one. here It do it by sequentially and publishes a message or event to trigger the next transaction step.

Some of the advantage of saga pattern is It avoid's cyclic dependencies, everything happens in a single place. If we want to develop an a orchestrator then linear kind of transaction happen. It follows the ACID property.It doesn't introduce a single point of failure, since the responsibilities are distributed across the saga participants.

and as I explained everything will be on one go the flow happen.

The main disadvantages while adding new services and maintain that.As It is contain both data and logic part and It is distributed so little bit of tricky i can say. Integration testing is difficult because all services must be running to simulate a transaction.

3. CQRS segregates reads and writes into different models. It uses the commands to update data and queries to read the data.Here the operation can either read data or it can write data but it cannot do both. This separation makes it easier to manage data. The read and write operations are much more manageable because their functions are focused. Secondly, these operations can be developed individually by different teams and optimized separately, evolving independently over time following customers requirements.The CQRS pattern I can say leverages the APIs for accessing data and the models for the management of data. The database itself is segregated into a read/write database that is write-only and one or more read-only databases.

Here, the write model of the system manages the persistence of the event while acting as the source of information for the read side. Whereas the read model offers materialized views of the data as highly denormalized views.

The Saga pattern provides transaction management using a sequence of local transactions. Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga. Here If a local transaction fails, the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions.

Benefits of Saga pattern are:

It doesn't require any additional service implementation or maintenance and It doesn't also introduce a single point of failure. For simple workflows It's best.

Ensure data consistency in a distributed system without any tight coupling.

Disadvantage s of saga pattern are: Integration testing is difficult bcz all services must be running to simulate the transaction. Sometimes workflow also confusing . There isa risk factor for cyclic dependency. i think the Saga pattern is particularly hard to debug, and the complexity grows as participants increase. Data can't be rolled back, because saga participants commit changes to their local databases.(durability challenges are also there). Sometimes some annomallies like semantic lock . If the application or project is tightly coupled transaction then it may cause problem.(Cyclic dependency also).

Advantages of the CQRS Pattern:

Single Responsibility Principle - As I explained, there are separate models, read & write. It means a method should be either Command or Query. With this separation, we get the Single Responsibility Principle by design.

Independent Scaling- One of the major benefits of the CQRS pattern is Independent scaling. Let's assume you have a website which has more reads than write (something like our application wolterskluwer.com). In such a case, we can scale your read models to get better performance.Also, we can use two separate databases or apis for the read & write models. With this approach, the write model will send a command to the write database (then it will asynchronously update the read database), and the read model will fetch data from the read database. We can denormalize a read database that will result in simple queries, less complex joins, quick response time. we are even using NoSQL(Azure CosmosDb) for the read database. then, scale your models independently.

Separation of Concern - Since we have two separate models to read & write, we are keeping validations, complex logic into the write model only and keeping read models simple to fetch the data. Similarly, we write model that will be responsible for writing to the model.

Disadvantages of the CQRS Pattern :

Although the CQRS pattern is exciting and easy to understand, in real scenarios it can be painful if not used correctly. Below are few downsides of this pattern What I think:

- It adds unnecessary complexity if applications have simple CRUD operations, which can be achieved by traditional architectural styles.

- As we require separate models for read & write, code duplication is inevitable.

- In the case of two separate databases for read & write, the write database needs to update the read database that could result in Eventually Consistent Views.