LTIMindtree

Whitepaper

# Maximize Group Benefits Enrollments using Cloud Design Pattern

Author

**Dinar Senjit**

# Table of Contents

## Abstract

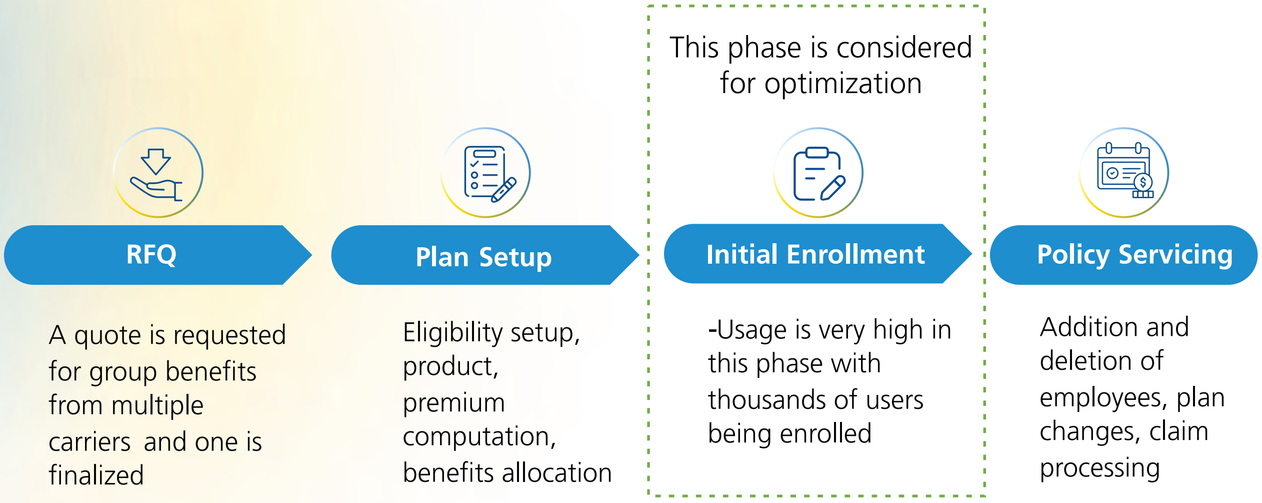
In most software systems, there is a significant skew in usage patterns. This skew is more evident in Group Benefits domain for the broker as the usage ranges in tens of thousands of users. Microservices architecture natively facilitates elastic demand handling by the ability to be packaged in a container and scale out as per demand. However, if not architected correctly, performance bottlenecks persist. This paper examines the use of CQRS design pattern to overcome the limitations and achieve highly optimized microservices that are highly optimized for performance. It also examines other facets affecting the benefits of CQRS.

# Introduction:

There has been a metamorphosis in software development over the period. As software gets more and more sophisticated, so does business, in its own manner to provide more and more excellence in service and deliver more and more value and thus drive an increase in revenue. Many times, this results in a skew in system usage to match business cycles or processes. This skew is more relevant in Group Benefits domain, where users are typically in tens of thousands.

## A big picture: Group Benefits Broker Life Cycle

Below are typical phases in a group benefits domain from a broker perspective-



A quote is requested for group benefits from multiple carriers and one is finalized

Eligibility setup, product, premium computation, benefits allocation

-Usage is very high in this phase with thousands of users being enrolled

Addition and deletion of employees, plan changes, claim processing

In typical group benefits phases, a high usage skew can be noted in the initial enrollment phase, highlighted in green. It is the most critical period for a broker system when large groups are trying to perform enrollment function. It is during this period that not only must the system be available, but also it needs to be at peak performance level. And in most cases this skew can be very high resulting in suboptimal performance.

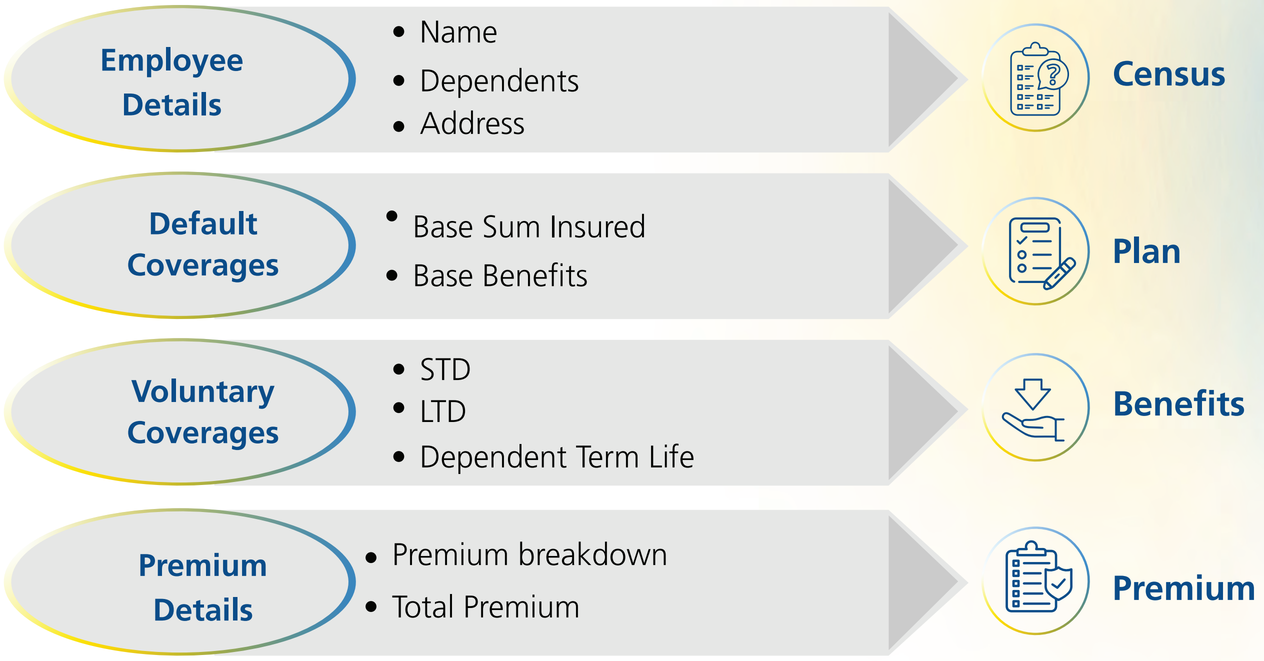
## Initial Enrollment

Initial enrollment refers to the onboarding of employees in a group. During this period, large number of employees are enrolled and subsequent functions such as data quality checks, data errors/rectifications, eligibility checks, product and benefits allocation, premium computations are performed.

As a corollary, group users will try to access more information about the coverage and amendments.

As the group policies are often highly tailored, every group will typically has its own criterion to set benefits based on age, gender, hierarchy/grade and accordingly allocate plan and premiums. From a system perspective, this implies more UI complexity.

### UI Composition

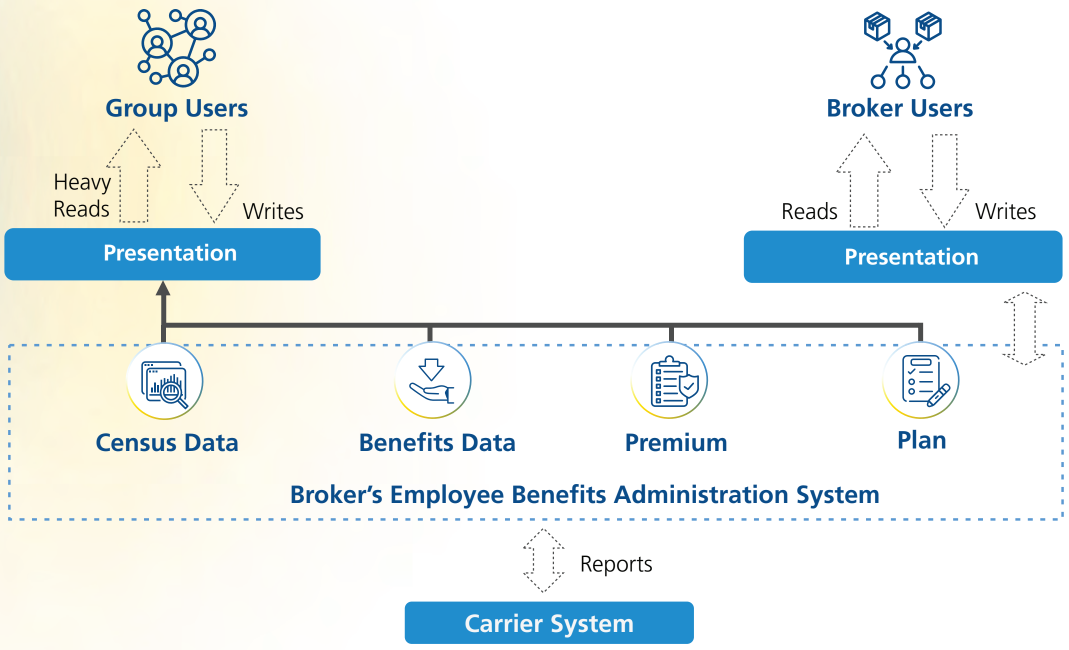


## Typical UI Composition-

The above diagram depicts a typical UI composition for a group user. It consists of several components. Employee details are fetched from census database. This involves employee's personal information, dependent details etc. This database is usually in-house data maintained by employer. Default coverages are base coverages available for employee based on position class or other strata. This information is typically fetched from plan database. Premium details indicate total premium for an employee with given choice of benefits including voluntary benefits.

# Typical system design for composing the UI

**Let us look at typical design in such applications**



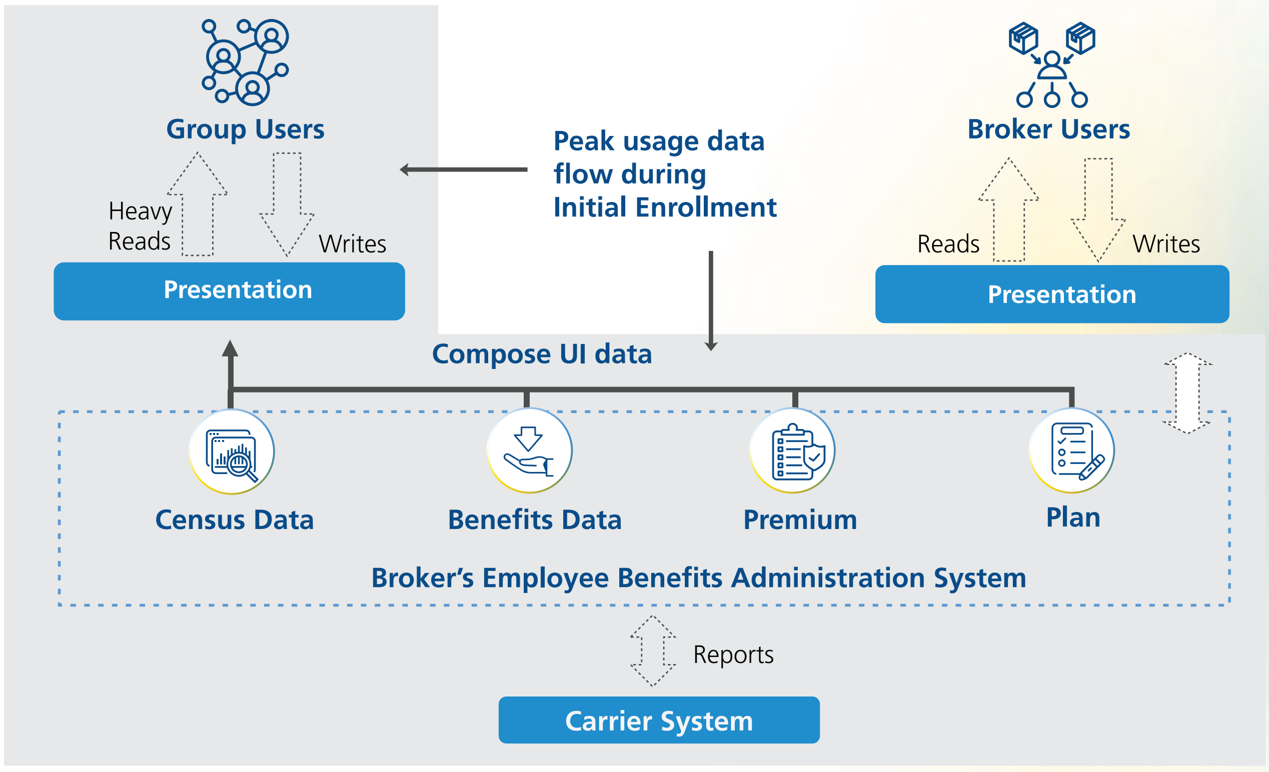
Typical Group Benefits System

As you can see in the above diagram, the architecture of a typical Group Benefits System for a broker involves a magnitude of complexity. The broker systems typically integrate with Carrier systems for reports. Additionally, there are subsystems for configuring benefits data. And there is a data store for premium calculation rules or parameters. Then there is a data store for census data which is often in bulk.

Broker users perform updates to plans and benefits, which eventually, post review, become live for group users.

Group users typically perform a lot of reads during initial enrollment. This is to know the benefits and offerings tailored for them before deciding on a selection.

## High data flow area during initial enrollment



In the above diagram, highlighted part of the system is used heavily during the initial enrollment period. During this period, large number of employees can be active simultaneously. This implies increased data reads from various databases to construct a complex UI for the users.

It is worth noting that data is not necessarily co-located. This introduces a network latency factor that can affect the overall rendering performance.

## Factors to consider for performance optimization-

1. Complex UI Composition for Group User- UI composition for a group user typically involves reads from multiple databases. The data is joined before consuming for presentation. Also, this data may be at different geo-location.
2. Versioned or infrequent updates- Broker users do update the data, but often it is versioned, reviewed and then published.
3. Integrations- The system typically has multiple integration points.
4. Caching Complexity- Since the reads are done from multiple sources, caching the output will be difficult. Also, it may not yield maximum benefit as view data extraction and join still needs to be performed.

# Architecting Microservices to tackle performance and scaling needs

## What is microservice architecture?

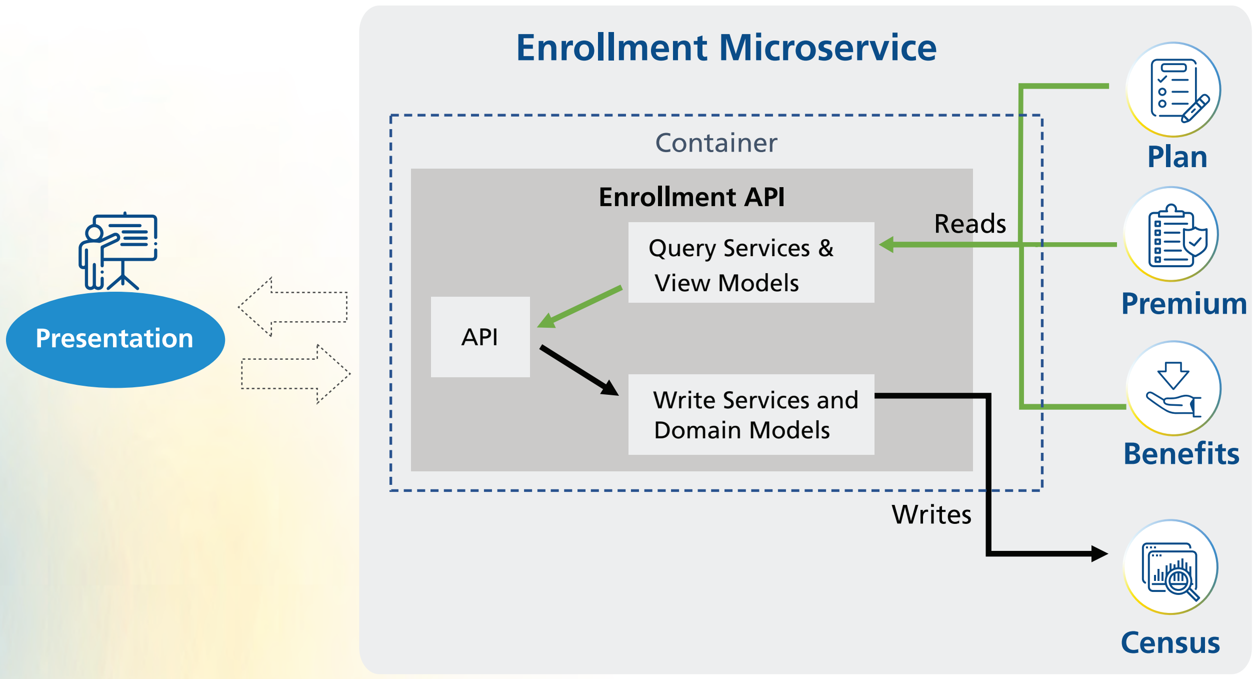
Microservice architecture - is an architectural style that models an application as a collection of small independent services. Micro-services are autonomous and specialized. Microservices can easily be packaged in a container and hence can scale out easily on demand spikes.

## Why do microservices help in tackling performance?

Microservices being independent  can be hosted in a container. In case of a demand spike, more containers can be instantiated. Thus, it is easy to manage demand spikes with microservices.

# Typical Microservice Design for Peak Usage Flow

Below diagram depicts a typical microservice implementation for the same-



As you can see in the above diagram, enrollment microservice would consist of-

1. **Enrollment API**- This consists of API that is exposed and consumed by UI to generate "View Model" data. It also contains methods to read, write as well as view model objects.
2. **Container-** It can be any container that will be provisioned and deployed.
3. **Databases-** Although not part of the microservice itself, the service will typically interact with various databases to read and write data.

## Inadequacies of this approach-

Although microservice with container can be easily scaled out by deploying more container instances to catch up with a sudden spike in usage, this may not yield expected performance gains in this case. Below

1. **Multiple data stores**- Please note that a typical read operation, in this case, involves composition from multiple data stores. Hence, simply scaling out UI won't be sufficient. Need to scale out supporting services to read for other DBs as well. This contributes to increased costs.
2. **Data Joins-** Joining data from multiple stores can still be performance-intensive operation. This necessitates high-performance resource. This again contributes to extra cost for CPU power.
3. **Read locks**- Although read heavily outnumber writes, they can still pose a contention for write operation. These further impacts performance as either of the operations may starve.

Hence, even if containers are scaled out, it does not yield performance gains in the same proportion.

# Considering CQRS (Command Query Responsibility Segregation) pattern:

## Overview:

CQRS is a design pattern that calls for the segregation of command and query. So CQRS recommends separating write and read operations into a separate store. This design helps separate the data model used for a read operation from the data model for write operations. For writes, commands are used. So basically, with CQRS, microservices will be split further based on read/write.

## Related Concepts

### Command

Commands encapsulate operations that modify data. These are typically autonomous and map to CRUD operations. Commands can also be implemented as events. Commands are typically task-based and should not be data-oriented.

e.g., Commands should be "add a dependent", "add benefits" etc.

### Query

Queries are the read operations. Queries do not modify data. CQRS calls for a separate data store mapping to the UI view model. Queries should return objects that are simple data objects without any domain context.

## Domain-Driven Design (DDD)

Domain-driven design is an approach for software development that focuses on domain. In DDD, the design, structure, modules, and other artifacts of software correspond to a function or group of functions. All decisions are driven by the domain.

### Bounded Context:

In DDD terms, a bounded context is a boundary within a domain. Typically bounded context is the largest microservice that can be designed.

To identify bounded context, a context mapping pattern is often used. Context mapping helps identify various contexts in the application.

## Eventual Consistency

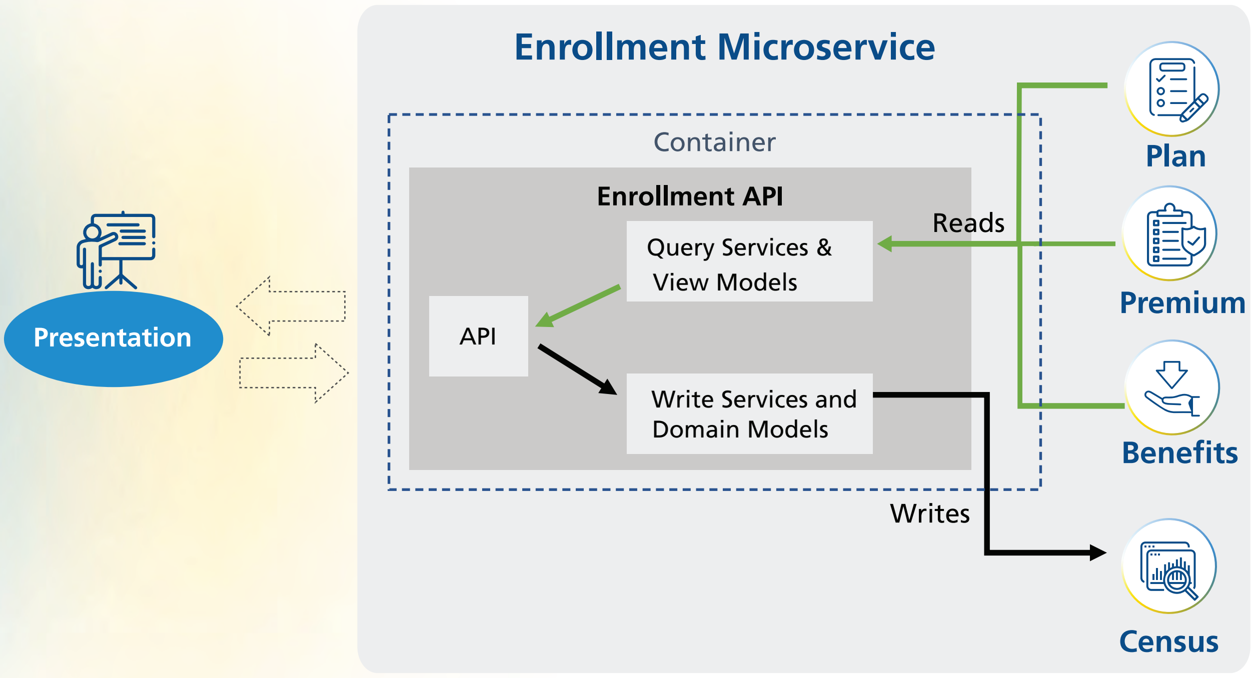
Eventual consistency is an important concept in the context of CQRS. In strict consistency, a write operation immediately propagates the change to all parts and data stores of a system. But with eventual consistency, the data may not be consistent in the beginning. In other words, writes are not immediately propagated to all parts of the system. So, the system is not immediately consistent. But it will eventually be consistent. Write operations will be propagated to other parts of the system either in a scheduled manner or on the event or some other way to other parts of the system. Eventual consistency provides fast reads with replicas. All replicas are not consistent, to begin with. But after a delay, the data becomes consistent.

# Applying CQRS in Initial Enrollment microservice-

## 1. Identify bounded context-

Bounded context in DDD terms in a boundary within a domain where a domain model is relevant. Bounded context also indicates the smallest possible functional area in a domain. Identifying bounded context can be a complex process.

In the case of initial enrollment, enrollment UI loading can be considered as a functional boundary or bounded context. The view model can be considered as a candidate domain object. This is depicted in the diagram below-



The area highlighted with a red dotted line is a bounded context. It indicated composing the UI with all data fetched from various sources.

## 2. De-normalize the data involved into a read model implemented as a logical or physical data store-

This involves identifying frequently joined data to form a view model. Denormalization refers to the process of storing data in aggregated form mapping to query output. This often results in faster read performance. In CQRS, data is typically stored in de-normalized form so that reads are fast. Also, since data updates are versioned, this data can be cached to enhance performance further.

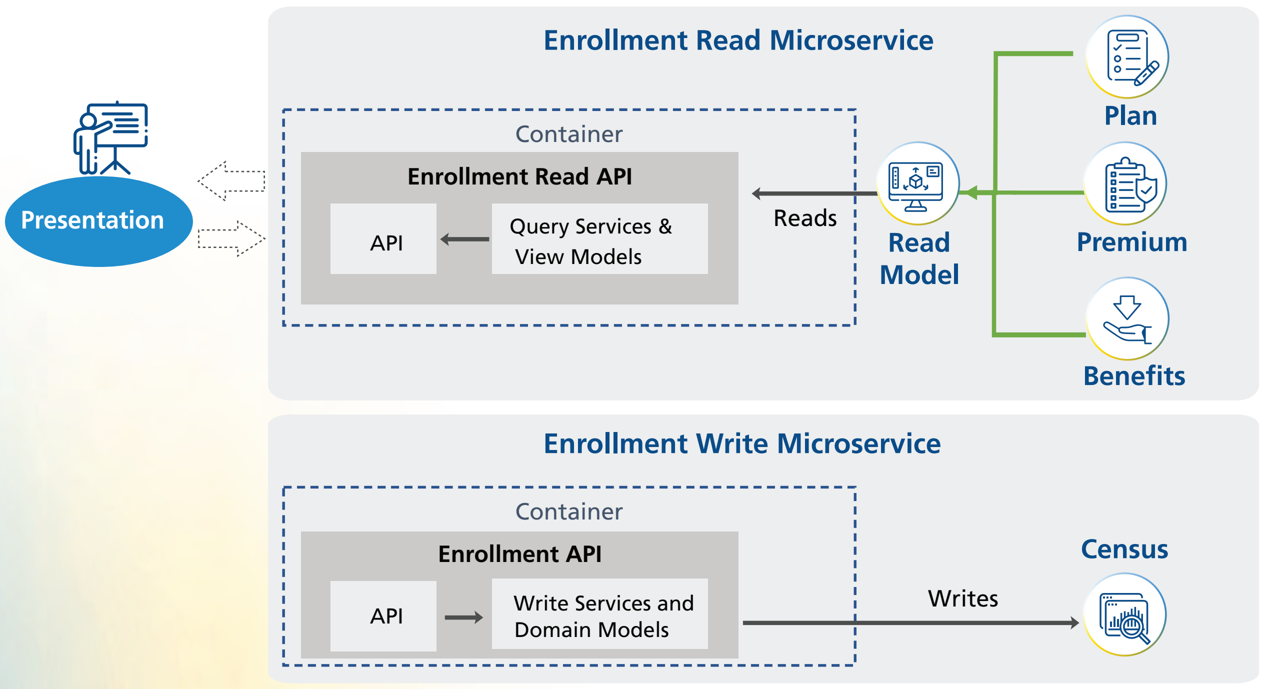
## 3. Ensure consistency requirements are met with separated data store-

De-normalized data can't be maintained at the most current level. There will be a delay in propagating updates from the source to de-normalized store. However, the data will be updated eventually. When applying CQRS, it is very important to evaluate if eventual consistency is acceptable. If the system demands strict consistency i.e., data updates must be immediately available, then it is important to consider the frequency of such writes. E.g., In a ticket booking system, booking updates must be propagated with strict consistency. If such writes requiring strict consistency are done very often, then CQRS implementation will not work.

## 4. Separate out read and write services

This typically involves separating out read and write API in the independent container. Read API will query the read model. And write API will write to the relevant data store.

# CQRS Microservice Implementation-



Enrollment Read Microservice
Presentation
Container
Enrollment Read API
API
Query Services & View Models
Reads
Read Model
Plan
Premium
Benefits
Enrollment Write Microservice
Container
Enrollment API
API
Write Services and Domain Models
Writes
Census

The above diagram depicts a CQRS microservice.

1. **Enrollment read microservice**

This microservice is used to query the view mode used by UI. It reads from de-normalized data store. It is implemented as an independent container.

2. **Enrollment write microservice**

This service is used to perform CRUD operations. CRUD operations in this scenario can be relatively simple.

## Advantages of CQRS

1. **Scalability**- Separating read and write as two separate microservices allows for independent scaling as per application demand requirements. In times of heavy load for read calls, it is easy to spin off new resources to contain the spike in demand.

2. **Caching-** With CQRS, read APIs directly map to view the model. With read API separated, caching can yield maximum performance gains as the data can now be consumed directly by the presentation. So in the above example, the read microservice can read from a cache when applicable.

3. **Cost savings-** Read microservice lighter as compared to initial microservice. This means that it will need fewer resources to deliver the same performance. Hence, it results directly in cost savings.

4. **Availabilit**y- With CQRS, data used for reading is typically maintained in a separate database So the likelihood of failures in both read and write is reduced. In the event of a catastrophe, the writes can be disabled but reads can still be available because the data is served separately for reads.

## Disadvantages of CQRS

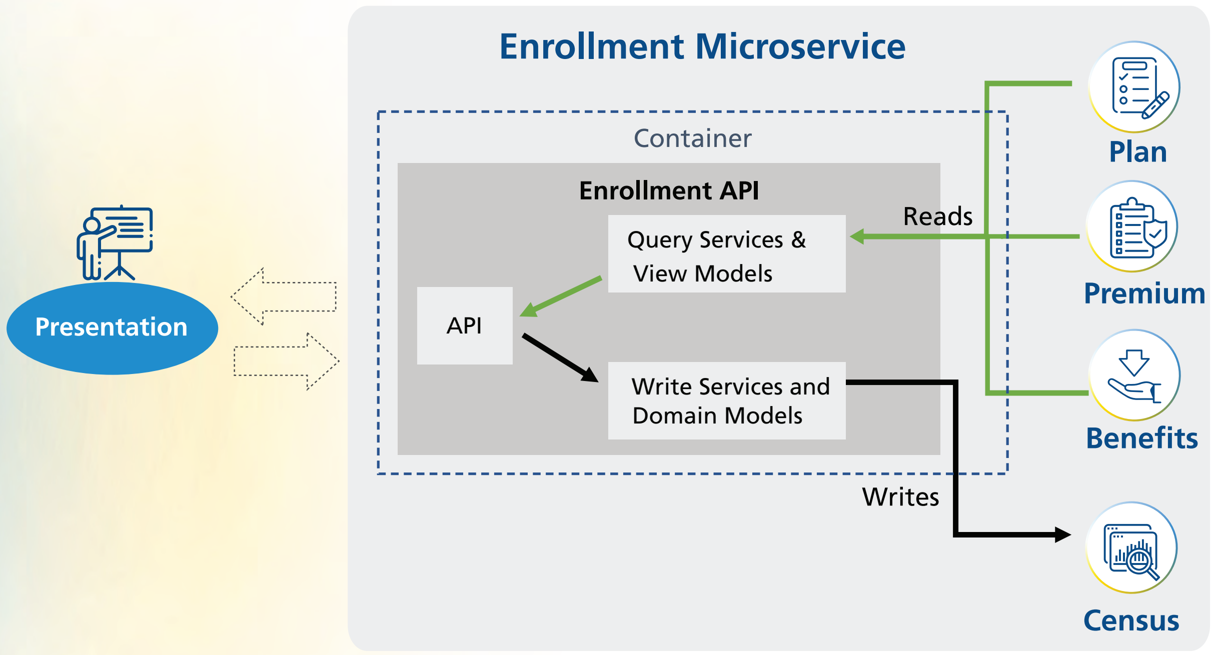
1. **Complexity-** Although the idea of CQRS is simple at conceptual level, implementation does involve a degree of complexity.

2. **Consistency-** This pattern cannot be used where eventual consistency is not acceptable

# Estimating performance gains with CQRS

The performance of an application depends on numerous factors. It also depends on the amount of data, indexes, network bandwidth, processing power at disposal and many more. In order to estimate performance gains with CQRS, we will try to do a heuristic estimation by comparing the most common factors involved in both case i.e., with and without CQRS.

## Estimating total time without CQRS



Referring to the above diagram, the total time before applying CQRS can be estimated as-

```
Total Time per request = f(max( (f(plan record time)
                                , f(premium record time)
                            , f(benefits record time)
                          , f(census record time)
                   + f(network latency)
                   + f(data join time))
```

f(plan record time) = time required to fetch a record from plan DB

f(premium record time) = time required to fetch a record from premium DB

f(benefits record time) = time required to fetch a record from benefits DB

f(census record time) = time required to fetch a record from census DB

f(network latency) = time required to transport data over the network

f(data join time) = time required to join data

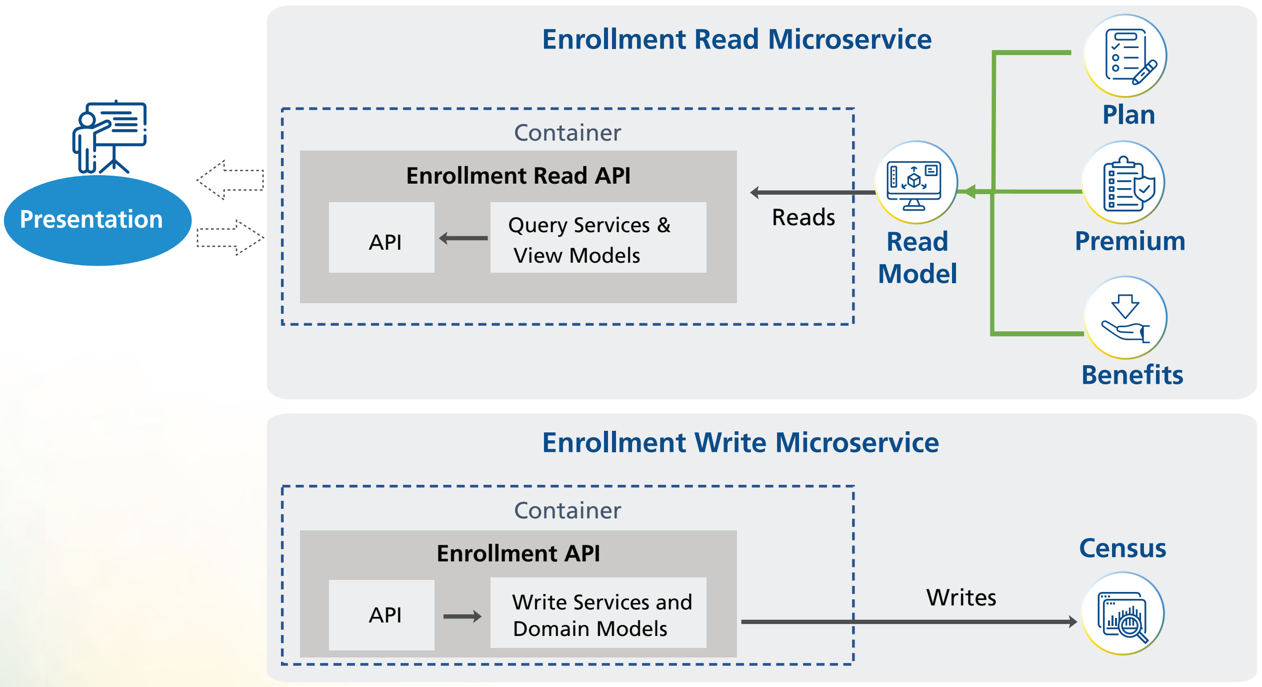
f(plan record time), f(premium record time), f(benefits record time) happen in parallel

f() denotes different fetch times. Since all data fetches can happen in parallel, we are considering the max of all the fetch times. There will be a factor of network latency. Also, there will be some time needed to join all the data.

Also, it is worth noting that due to the nature of data fetched from various sources, caching cannot be applied. Even if applied, it would yield minimum benefit.

So it is likely that the performance of first request will be more or less the same as the performance of the nth request.

## Estimating total time with CQRS:


Enrollment Read Microservice
Container
Enrollment Read API
API
Query Services & View Models
Reads
Read Model
Plan
Premium
Benefits
Presentation
Enrollment Write Microservice
Container
Enrollment API
API
Write Services and Domain Models
Writes
Census

Referring to above diagram, the total time before applying CQRS can be estimated as-

> Total Time per request = f(read record time) + f(network latency)
>
> f(read record time) = time required to fetch a record from de-normalized read model
>
> f(network latency) = time required to transport data over network

As we can see, in the case of CQRS, the total time is the simple time needed to fetch a record from de-normalized read model and network latency. This is because, in CQRS, read models are stored in a de-normalized manner. So data need not be fetched from various sources.

Additionally, since there is a single de-normalized source of data that doesn't refresh often, caching can be applied and can further reduce the response time.

So, after the first request, subsequent requests are likely to be much faster as they will fetch data from the cache instead of a database.

So we can see that CQRS implementation is clearly significantly faster in terms of response time.

# Conclusion

CQRS is not a panacea for all scalability optimizations. Like any pattern, CQRS has its own strengths and weaknesses. It needs to be used judiciously to gain maximum benefits.

## Aspects of successful CQRS implementation

**1) Consistency Requirements**

CQRS assumes eventual consistency. In systems where high consistency is required, CQRS will not work. Also, eventual consistency implementation involves the use of other mechanisms like queue service, event sourcing patterns etc. This adds to the overall complexity of the design.

**2) The complexity of domain itself**

Implementing CQRS adds a certain level of complexity. If the domain itself is simple enough so that CRUD APIs are sufficient, then the complexity of CQRS will not outweigh its benefits.

**3) Ability to identify a bounded context for segregation**

In some systems, the domain itself can be so complex that it may not be possible to identify and single out a bounded context where read and write can be separated.

**4) CQRS is not a top-level architectural style for a system**

It is important to note that CQRS is not a general architectural style to be applied to system as a whole. But it needs to be applied to parts of the system where applicable. Failure to do so often leads to the failed implementation of CQRS.

CQRS has advantages as well as disadvantages. When used in the right manner, it has a potential to improve performance to a great extent.

## References:

Microservices:
https://aws.amazon.com/microservices/

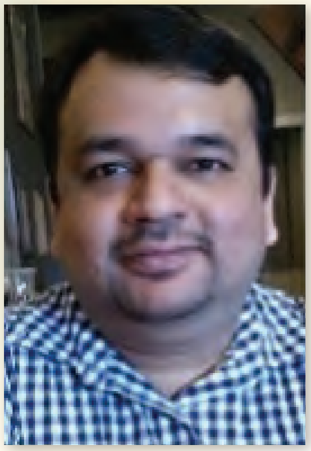
Domain-Driven Design:
https://learn.microsoft.com/en-us/archive/msdn-magazine/2009/february/best-practice-an-introduction-to-domain-driven-design

CQRS:
https://martinfowler.com/bliki/CQRS.html

Eventual consistency:
https://en.wikipedia.org/wiki/Eventual_consistency

Bounded Context:
https://codeburst.io/ddd-strategic-patterns-how-to-define-bounded-contexts-2dc70927976e

## Author Profile



**Dinar Senjit**
Technical Architect, LTIMindTree

Dinar Senjit works as Microsoft Technical Architect specializing in Azure, .NET, SharePoint, and Dynamics CRM. He has about 17+ years of hands-on experience in coding, design, and architecture. Equipped with extensive domain knowledge in the Insurance industry, HR-Careers, etc. He has worked on various solutions and applications across US, UK and Canada. He is a technology evangelist with an interest in ElasticSearch and other emerging technologies.