# Application of the domain driven design in cloud native computing

## Abstract

The number of cloud-based systems using domain-driven design has been increasing in recent years.

## Introduction

Интро

## Patterns –

DDD techniques that enable to find common patterns that can be reuse in cloud applications.

Command Query Separation (CQS) was originally defined by Bertrand Meyer in his book Object-Oriented Software Construction. The fundamental idea is that object's methods should be divided into two categories:  
- Queries: free of side effects, which return a result and do not change the state of the system;  
- Commands: Change the state of a system but do not return a value;

Command and Query Responsibility Segregation (CQRS) was introduced by Greg Young. It is based on the CQS principle. It can be considered an architectural pattern that separates the models for reading and writing data based on commands and events plus optionally on asynchronous messages. In many cases, CQRS is related to more advanced scenarios. A more advanced CQRS system might also use Event-Sourcing (ES), which stores events in the domain model instead of the current-state data (César de la Torre, Bill Wagner, Mike Rousos).

The separation aspect of CQRS is achieved by grouping query operations in one layer and commands in another layer. Each layer has its own data model. More importantly, the two layers can be within the same tier or microservice, or they could be implemented on different microservices or processes so they can be optimized and scaled out separately without affecting one another. CQRS means having two objects for a read/write operation where in other contexts there is one. The goal is to have more flexibility in the queries instead of limiting the queries with constraints from DDD patterns like aggregates.

An example of this kind of service is the ordering microservice from the eShopOnContainers reference application. This service implements a microservice based on a simplified CQRS approach. It uses a single data source or database, but two logical models plus DDD patterns for the transactional domain, as shown in Figure 7-2.

Diagram showing a high level Simplified CQRS and DDD microservice.



Figure 7-2. Simplified CQRS- and DDD-based microservice

The Logical "Ordering" Microservice includes its Ordering database, which can be, but doesn't have to be, the same Docker host. Having the database in the same Docker host is good for development, but not for production.

The application layer can be the Web API itself. The important design aspect here is that the microservice has split the queries and ViewModels (data models especially created for the client applications) from the commands, domain model, and transactions following the CQRS pattern. This approach keeps the queries independent from restrictions and constraints coming from DDD patterns that only make sense for transactions and updates, as explained in later sections.

The essence of those patterns, and the important point here, is that queries are idempotent: no matter how many times you query a system, the state of that system won't change. In other words, queries are side-effect free.

Therefore, you could use a different "reads" data model than the transactional logic "writes" domain model, even though the ordering microservices are using the same database. Hence, this is a simplified CQRS approach.

On the other hand, commands, which trigger transactions and data updates, change state in the system. With commands, you need to be careful when dealing with complexity and ever-changing business rules. This is where you want to apply DDD techniques to have a better modeled system.

The DDD patterns presented in this guide should not be applied universally. They introduce constraints on your design. Those constraints provide benefits such as higher quality over time, especially in commands and other code that modifies system state. However, those constraints add complexity with fewer benefits for reading and querying data.

One such pattern is the Aggregate pattern, which we examine more in later sections. Briefly, in the Aggregate pattern, you treat many domain objects as a single unit as a result of their relationship in the domain. You might not always gain advantages from this pattern in queries; it can increase the complexity of query logic. For read-only queries, you do not get the advantages of treating multiple objects as a single Aggregate. You only get the complexity.

As shown in Figure 7-2 in the previous section, this guide suggests using DDD patterns only in the transactional/updates area of your microservice (that is, as triggered by commands). Queries can follow a simpler approach and should be separated from commands, following a CQRS approach.

It's important to understand that CQRS and most DDD patterns (like DDD layers or a domain model with aggregates) are not architectural styles, but only architecture patterns. Microservices, SOA, and event-driven architecture (EDA) are examples of architectural styles. They describe a system of many components, such as many microservices. CQRS and DDD patterns describe something inside a single system or component; in this case, something inside a microservice.

Different Bounded Contexts (BCs) will employ different patterns. They have different responsibilities, and that leads to different solutions. It is worth emphasizing that forcing the same pattern everywhere leads to failure. Do not use CQRS and DDD patterns everywhere. Many subsystems, BCs, or microservices are simpler and can be implemented more easily using simple CRUD services or using another approach.

There is only one application architecture: the architecture of the system or end-to-end application you are designing (for example, the microservices architecture). However, the design of each Bounded Context or microservice within that application reflects its own tradeoffs and internal design decisions at an architecture patterns level. Do not try to apply the same architectural patterns as CQRS or DDD everywhere.

## Design a DDD-oriented microservice

Domain-driven design (DDD) advocates modeling based on the reality of business as relevant to your use cases. In the context of building applications, DDD talks about problems as domains. It describes independent problem areas as Bounded Contexts (each Bounded Context correlates to a microservice), and emphasizes a common language to talk about these problems. It also suggests many technical concepts and patterns, like domain entities with rich models (no [anemic-domain model](https://martinfowler.com/bliki/AnemicDomainModel.html)), value objects, aggregates, and aggregate root (or root entity) rules to support the internal implementation. This section introduces the design and implementation of those internal patterns.

Sometimes these DDD technical rules and patterns are perceived as obstacles that have a steep learning curve for implementing DDD approaches. But the important part is not the patterns themselves, but organizing the code so it is aligned to the business problems, and using the same business terms (ubiquitous language). In addition, DDD approaches should be applied only if you are implementing complex microservices with significant business rules. Simpler responsibilities, like a CRUD service, can be managed with simpler approaches.

Where to draw the boundaries is the key task when designing and defining a microservice. DDD patterns help you understand the complexity in the domain. For the domain model for each Bounded Context, you identify and define the entities, value objects, and aggregates that model your domain. You build and refine a domain model that is contained within a boundary that defines your context. And that is explicit in the form of a microservice. The components within those boundaries end up being your microservices, although in some cases a BC or business microservices can be composed of several physical services. DDD is about boundaries and so are microservices.

Determining where to place boundaries between Bounded Contexts balances two competing goals. First, you want to initially create the smallest possible microservices, although that should not be the main driver; you should create a boundary around things that need cohesion. Second, you want to avoid chatty communications between microservices. These goals can contradict one another. You should balance them by decomposing the system into as many small microservices as you can until you see communication boundaries growing quickly with each additional attempt to separate a new Bounded Context. Cohesion is key within a single bounded context.

It is similar to the [Inappropriate Intimacy code smell](https://sourcemaking.com/refactoring/smells/inappropriate-intimacy) when implementing classes. If two microservices need to collaborate a lot with each other, they should probably be the same microservice.

Another way to look at this aspect is autonomy. If a microservice must rely on another service to directly service a request, it is not truly autonomous.

## Layers in DDD microservices

Most enterprise applications with significant business and technical complexity are defined by multiple layers. The layers are a logical artifact, and are not related to the deployment of the service. They exist to help developers manage the complexity in the code. Different layers (like the domain model layer versus the presentation layer, etc.) might have different types, which mandate translations between those types.

For example, an entity could be loaded from the database. Then part of that information, or an aggregation of information including additional data from other entities, can be sent to the client UI through a REST Web API. The point here is that the domain entity is contained within the domain model layer and should not be propagated to other areas that it does not belong to, like to the presentation layer.

Additionally, you need to have always-valid entities (see the [Designing validations in the domain model layer](https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/domain-model-layer-validations) section) controlled by aggregate roots (root entities). Therefore, entities should not be bound to client views, because at the UI level some data might still not be validated. This reason is what the ViewModel is for. The ViewModel is a data model exclusively for presentation layer needs. The domain entities do not belong directly to the ViewModel. Instead, you need to translate between ViewModels and domain entities and vice versa.

When tackling complexity, it is important to have a domain model controlled by aggregate roots that make sure that all the invariants and rules related to that group of entities (aggregate) are performed through a single entry-point or gate, the aggregate root.

## The domain model layer

Eric Evans's excellent book [Domain Driven Design](https://domainlanguage.com/ddd/) says the following about the domain model layer and the application layer.

**Domain Model Layer**: Responsible for representing concepts of the business, information about the business situation, and business rules. State that reflects the business situation is controlled and used here, even though the technical details of storing it are delegated to the infrastructure. This layer is the heart of business software.

The domain model layer is where the business is expressed. When you implement a microservice domain model layer in .NET, that layer is coded as a class library with the domain entities that capture data plus behavior (methods with logic).

Following the [Persistence Ignorance](https://deviq.com/persistence-ignorance/) and the [Infrastructure Ignorance](https://ayende.com/blog/3137/infrastructure-ignorance) principles, this layer must completely ignore data persistence details. These persistence tasks should be performed by the infrastructure layer. Therefore, this layer should not take direct dependencies on the infrastructure, which means that an important rule is that your domain model entity classes should be POCOs.

Domain entities should not have any direct dependency (like deriving from a base class) on any data access infrastructure framework like Entity Framework or NHibernate. Ideally, your domain entities should not derive from or implement any type defined in any infrastructure framework.

Most modern ORM frameworks like Entity Framework Core allow this approach, so that your domain model classes are not coupled to the infrastructure. However, having POCO entities is not always possible when using certain NoSQL databases and frameworks, like Actors and Reliable Collections in Azure Service Fabric.

Even when it is important to follow the Persistence Ignorance principle for your Domain model, you should not ignore persistence concerns. It is still important to understand the physical data model and how it maps to your entity object model. Otherwise you can create impossible designs.

Also, this aspect does not mean you can take a model designed for a relational database and directly move it to a NoSQL or document-oriented database. In some entity models, the model might fit, but usually it does not. There are still constraints that your entity model must adhere to, based both on the storage technology and ORM technology.

### The application layer

Moving on to the application layer, we can again cite Eric Evans's book [Domain Driven Design](https://domainlanguage.com/ddd/):

**Application Layer:** Defines the jobs the software is supposed to do and directs the expressive domain objects to work out problems. The tasks this layer is responsible for are meaningful to the business or necessary for interaction with the application layers of other systems. This layer is kept thin. It does not contain business rules or knowledge, but only coordinates tasks and delegates work to collaborations of domain objects in the next layer down. It does not have state reflecting the business situation, but it can have state that reflects the progress of a task for the user or the program.

A microservice's application layer in .NET is commonly coded as an ASP.NET Core Web API project. The project implements the microservice's interaction, remote network access, and the external Web APIs used from the UI or client apps. It includes queries if using a CQRS approach, commands accepted by the microservice, and even the event-driven communication between microservices (integration events). The ASP.NET Core Web API that represents the application layer must not contain business rules or domain knowledge (especially domain rules for transactions or updates); these should be owned by the domain model class library. The application layer must only coordinate tasks and must not hold or define any domain state (domain model). It delegates the execution of business rules to the domain model classes themselves (aggregate roots and domain entities), which will ultimately update the data within those domain entities.

Basically, the application logic is where you implement all use cases that depend on a given front end. For example, the implementation related to a Web API service.

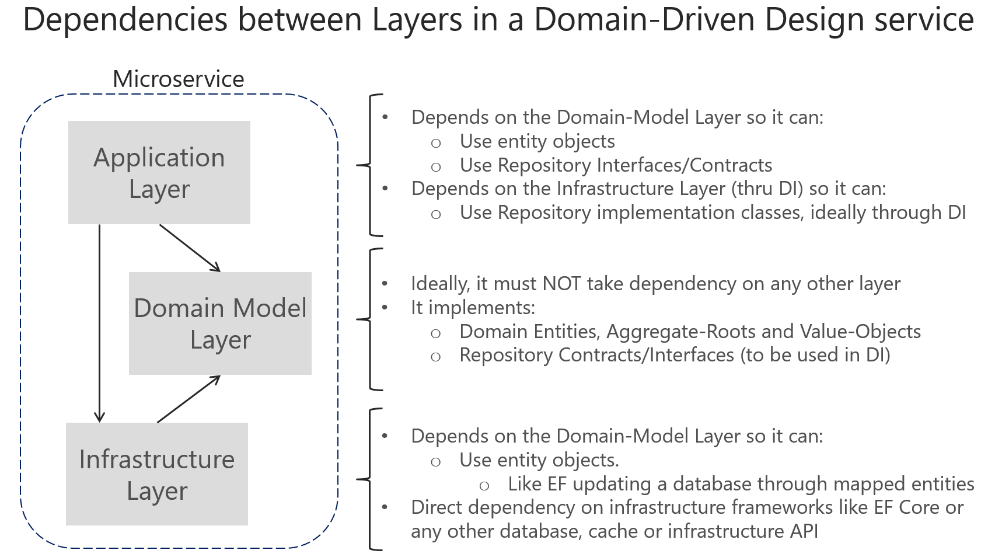
The goal is that the domain logic in the domain model layer, its invariants, the data model, and related business rules must be completely independent from the presentation and application layers. Most of all, the domain model layer must not directly depend on any infrastructure framework.

### The infrastructure layer

The infrastructure layer is how the data that is initially held in domain entities (in memory) is persisted in databases or another persistent store. An example is using Entity Framework Core code to implement the Repository pattern classes that use a DBContext to persist data in a relational database.

In accordance with the previously mentioned [Persistence Ignorance](https://deviq.com/persistence-ignorance/) and [Infrastructure Ignorance](https://ayende.com/blog/3137/infrastructure-ignorance) principles, the infrastructure layer must not "contaminate" the domain model layer. You must keep the domain model entity classes agnostic from the infrastructure that you use to persist data (EF or any other framework) by not taking hard dependencies on frameworks. Your domain model layer class library should have only your domain code, just POCO entity classes implementing the heart of your software and completely decoupled from infrastructure technologies.

Thus, your layers or class libraries and projects should ultimately depend on your domain model layer (library), not vice versa, as shown in Figure 7-7.



**Figure 7-7**. Dependencies between layers in DDD

Dependencies in a DDD Service, the Application layer depends on Domain and Infrastructure, and Infrastructure depends on Domain, but Domain doesn't depend on any layer. This layer design should be independent for each microservice. As noted earlier, you can implement the most complex microservices following DDD patterns, while implementing simpler data-driven microservices (simple CRUD in a single layer) in a simpler way.



## In practice –

Every software project has a set of attributes, the most important of which are the amounts of data it operates, performance requirements, business logic and technical complexity. The techniques DDD proposes are useful for projects that have a lot of complex business rules. DDD won't help you if you work with big data, need to achieve outstanding performance, or program against hardware systems. The only purpose DDD concepts serve is to tackle business logic complexity.

Every software project has a set of attributes, the most important of which are the amounts of data it operates on, performance requirements, business logic complexity, and technical complexity. Business logic complexity is the first indicator of how complicated the problem domain in which a software works is. For example, a CRUD application that performs basic create, read, update, delete operations doesn't carry a lot of complexity with it. At the same time, an ERP system, which automates a big chunk of the company's activity, must model all the processes the company acts upon and thus handle a lot of complex business roles. The business logic complexity of such a system may be extremely high. Another attribute is the technical complexity. which are the algorithms that need to be implemented to make the software work.

The techniques DDD proposes are useful for projects that have a lot of complex business rules. DDD won't help if a project needs to achieve outstanding performance or program against hardware systems. The only purpose DDD concepts serve is to tackle business logic complexity. A typical example of software with complicated business logic is enterprise-level applications. Most enterprise projects don't have outstanding performance requirements. Developers working on them usually don't have to deal with technical complexity on their own because there are plenty of tools that abstract out this kind of complexity for them. The most difficult challenge in such projects is dealing with the complexity of business logic in such a way that the solution can be extended and maintained in the long run. That is precisely the task that domain-driven design practices are intended to address. They help with creating code, which not only fully powers the problem but also does it in the simplest and thus most maintainable way possible.

There are several core principles in software development; two of them are YAGNI and KISS. YAGNI stands for "you are not going to need it" and basically means the implementation should include only the functionality needed in this particular moment. Future needs shouldn't be anticipated because most of the functionality sometimes turns out to be unused and thus a waste of time. KISS stands for "keep it short and simple." This principle is about making the implementation of the remaining functionality as simple as possible. The point here is that the simpler the code is, the more readable and thus more maintainable it becomes. These principles are important because they help solve two major problems: shortening the time needed for development and keeping the code base maintainable in the long run. The beauty of domain-driven design is that its practices complement these two software development principles. It allows for the extraction of the central part of the problem domain and its simplification, removing most of the necessary complexity. The ability to express business logic in the clearest way possible is a single trait that makes domain-driven design so appealing in enterprise-level applications.

It is hard to estimate how important that is. Keeping that complexity under control is the most difficult task in modern business line software. There is only so much complexity we can deal with at once. If the code exceeds it, it becomes really hard, and at some point even impossible, to change anything in the software without introducing some unexpected side effects. Extending such a project becomes a pain and usually results in a lot of bugs. This, in turn, slows down the development and may eventually lead to the failure of the project. One of the most common reasons for software project failure is uncontrollable complexity growth. Domain-driven design helps prevent it.

## Main Concepts of Domain-Driven Design

The first important concept is the notion of ubiquitous language, that is, the language structured around the domain model and used by all team members to refer to the elements of that domain. In many projects, domain experts and developers use different sets of terms when they talk about the domain. This difference leads to misunderstandings and slows down the overall development process. The notion of ubiquitous language helps eliminate the barrier.

Domain-driven design suggests explicitly highlighting those differences and adjusting terminology to conform to a single universal language. For the idea of ubiquitous language to work, the code base needs to be in sync with the terminology, or, more specifically, classes and tables in the database need to be named after the terms in the ubiquitous language. All this helps bridge the gap and set the groundwork for efficient communication.

Another important part of domain-driven design is the concept of bounded contexts. Often, an application grows so much that it becomes hard to maintain its code base as a whole. Code elements that make sense in one part of the system may seem completely irrelevant in another. In this case, the best solution would be to explicitly separate these parts from each other. That is where this concept helps. It allows for a clear definition of the boundaries of these parts, hence the name "bounded context."

The third concept is the notion of "core domain." Domain-driven design states that the main part of any system is its business logic, and not all of it but the most intrinsic piece of it, that is, the problem the software is meant to solve.

Domain-driven design proposes that the focus should always be on the core domain. These three concepts—ubiquitous language, bounded context, and core domain—are the most important parts of domain-driven design. They are the strategic elements of DDD. The other notions, such as entities, value objects, and repositories, comprise the tactics of how a software project should be built.

Domain-driven design is not only about writing code. Adhering to DDD practices also implies a heavy communication process between developers and domain experts. It's important to have direct access to the experts in the problem domain, because that's the only way to get complete information about the problem you are solving. To get the most out of domain-driven design, the domain knowledge should constantly be refined with the experts in the company; it shouldn't be a one-way process. Code written in the ubiquitous language can provide a hint for some edge cases that weren't clear enough at the start, or it can rewrite the problem statement in a much cleaner and more concise manner.

Software developers are often enthusiastic about coding tasks that regard building an infrastructure for a future project. Such tasks are often appealing because they present a technical challenge. Also, because of their technical nature, the knowledge acquired when solving them can be reused in other projects. All these make such activities compelling to many developers. But having in-depth domain knowledge will enable a programmer to do the best job possible. This knowledge will guide through the code. It will help to look at it from the domain expert's point of view. This skill is indispensable, as it allows you to combine the best of the two worlds: write technically correct code on the one hand and express domain knowledge with it on the other. Although problem domains are different from project to project, the skill of systematizing them with code is reusable.

## Onion Architecture and Domain Isolation

DDD notions form a construction named "onion architecture." It is so named because it resembles an onion with multiple layers and a core inside. The upper layers depend on the lower ones, but the lower layers don't know about the upper. It might seem similar to a classic onion-layer architecture with entities. The difference here is that onion architecture emphasizes the fact that the core part of this structure cannot depend on anything else except itself. It means that the core elements of our domain model should act in isolation from each other. The core part of this so-called onion is the notion of entity, value object, domain event, and aggregate. The next layer consists of repositories, factories, and domain services. Application services go beyond that, and finally, UI is the outermost layer, if, of course, the application contains a user interface. All work with a database should be encapsulated in repositories. They can refer to it either directly or through an ORM, but the general rule should remain. The code working with the data storage must be gathered under the repositories in the domain model. These four elements: entities, value objects, domain events, and aggregates, are the most basic. They can refer to each other, for example, and then they can contain a value object, or a value object can keep a reference to an aggregate root, but they cannot work with other DDD notions, such as repositories and factories. Similarly, repositories, factories, and domain services can know about each other and the four basic elements, but they should not refer to the application services. The main reason for the isolation of the four core elements of the domain model is the separation of concerns. Entities, value objects, domain events, and aggregates carry the most important part of the application, its business logic. They don't contain all of it, of course. Repositories and factories can keep some of the business logic as well, but these four elements include most of it. In the situation where you have some elements so deeply involved in the problem domain representation, it is vital to keep them as free as possible from other duties. It is crucial to leave entities and value objects to do only one thing: represent the domain logic in the application. In practice, it means they shouldn't contain any knowledge about how they are preserved or created. These two operations must be up to the standards of repositories and factories. They also shouldn't contain any knowledge about the tables and columns in the database where they are stored. This must be given away to database members. All they should know is the domain they represent. The cleaner the domain model is kept, the easier it is to reason about it and to extend it later on. Inability to maintain proper separation of concerns in enterprise-level applications is one of the biggest reasons why code bases become a mess, which leads to delays and even failure of the project. It is not always possible to separate them completely, though, and there will always be some elements not related to the domain. Nevertheless, it is possible to keep those elements under control so that they introduce almost no overhead to the domain clauses.



building blocks of domain-driven design

## Modeling Best Practices

DDD principles dictate that the main model becomes the heart of the software; this fact entails a guideline for how to work with the application code base. In practice, it means that the development process should always start with modeling the core domain, even if there is no UI or database structure yet. The user interface and the database are important elements of the system as well, but the core domain is the most important part. The investments in the domain model pay off greatly over time. It also means that in a typical enterprise application, an infrastructure code is less important than the core domain. Make certain that the business logic remains intact and does not dissolve in the infrastructure code.

## Domain-Driven Design and Unit Testing

When it comes to unit testing, it's important to keep a balance between the amount of coverage and the amount of effort. 100% coverage is an expensive mark to reach, and it doesn't necessarily provide proportional value to the quality of your software. In most enterprise-level applications, the value distribution corresponds to the number of unit tests in this way. At some point, the value gained from the additional tests doesn't justify the resources invested. In practice, unit tests should cover only those parts of the code base that are the most significant to the application, and this is the innermost layer in the onion architecture: entities, value objects, aggregates, and domain events, the elements that contain most of the domain knowledge of your application. It's a good idea to get 100% or close to 100% as coverage of them. That is another reason why the core layer of the domain model should remain isolated from other parts of the application, such as the database, email service, and so on. A good separation of concerns helps create testable code, which doesn't require any mocks or other test doubles to be tested. If the unit tests are hidden in the database or some other external dependency, it's a strong sign the domain model is not properly isolated.

Integration testing should cover the other parts of the code base. They are automated tests which cover several pieces of the application at once. Implementing them should be an easy task due to the great isolation achieved for the domain model.

Summary

This article explains how DDD principles apply to specific types of cloud services as well as provides an overview of software design principles such as YAGNI, which stands for implementing only the functionality you need right now, and KISS, which proposes the use of the simplest solution possible. These principles can help greatly when going along with a project. The beauty of domain-driven design is that its principles perfectly align with breaking a problem into consumable chunks and reducing its complexity to a level where it's no longer hard to understand and implement. The article emphasizes the importance of communication with domain experts, which needs to be two-sided, and the importance of the domain knowledge itself. As well as the notion of onion architecture, unit testing and why the domain model should remain isolated.