# Applying the Domain-Driven Design in Cloud-Native Services

## Abstract

The number of cloud-based systems using domain-driven design has been increasing in recent years. This paper gives a brief overview of domain-driven, cloud-based software development activities and how they fit into a well-known software development process.

## I. Introduction

Domain-Driven Design (DDD) is an approach to software development that focuses on the application domain, its concepts, and their relationships as primary drivers for architecture design (Rademacher, F., 2017). Core principles of DDD include capturing relevant domain knowledge in domain models, which can include both structural and behavioral aspects, collaborative modeling between domain experts and software engineers. Also encouraging experimental design by strictly aligning model and implementation throughout the software development process, as well as continuous model refinement (Sachweh, S., Zündorf, A., 2017). Domain-driven design gives patterns, activities, and examples of how to build a domain model, which is its main artifact (Hippchen, B. et all, 2017). Also, putting the patterns and principles into software architecture concepts like architecture perspectives and their requirements helps software architects design cloud-native solutions (Giessler, P., Steinegger, R., 2017).

## II. Theoretical foundation

This section introduces the design and implementation of domain-driven design patterns.

### Cloud-Native Services

One of the first decisions to make when implementing a cloud solution is which service(s) to use to run the applications. Table 1 shows the choices for which cloud services are best for which types of applications.

**Table 1:** Which cloud services are best suited for which types of applications

(Rob Caron Sr. Product Marketing Manager, Barry Luijbregts, Microsoft Azure, 2022)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Web service | Mobile service | Serverless | Virtual Machine | Microservices |
| Monolithic and N-Tier app | **✓** |  |  | **✓** |  |
| Mobile app back end | **✓** | **✓** |  |  | **✓** |
| Distributed system |  |  | **✓** |  | **✓** |
| Business process workflow | **✓** |  | **✓** |  |  |

#### Web service

One of the easiest and powerful ways to host applications in the cloud environment is the HTTP-based service for hosting web applications. Some examples are Azure App Hosting Service, AWS Elastic Beanstalk, Google App Engine. They provide a set of services that host an application and hide the complexity of the operating system and infrastructure. They are highly available by default and will stay up and running for at least 99.95% of the time. They share powerful features like automatic scaling, zero-downtime deployments, and easy authentication and authorization (Martin Ekuan). Some of them enable debugging the application while it is in production, using tools such as Snapshot Debugger (Hannah Hunter, 2022).

#### Mobile service

When building a mobile application, there is a need for a back end that the mobile application connects to. Usually, this is some sort of API that the app can use to retrieve and store data. Azure Mobile Apps and AWS Amplify provide such solutions with unique capabilities. For instance, there is an offline sync that enables the mobile app to continue working when there is no connection to the back end, and the sync changes once the connection is restored. Another feature is sending push notifications to the mobile apps, regardless of the platform they run on (iOS, Android, or Windows).

#### Serverless compute

These apps are small pieces of code that were written without worrying about the underlying infrastructure or about scaling. Many refer to this deployment model as "Functions as a Service" (FaaS). A wide range of events, both internal and external to the cloud provider, can easily trigger function applications. A function app is able to respond to web requests thanks to HTTP triggers. These functions even handle the scaling. They transparently spin up more functions to deal with high loads, and they go away when the code is done executing. Because of this, companies only pay for the code that is executed, not for a service that runs all the time, waiting to be triggered.

#### Virtual Machines

This is an easy way to get started because it allows you to lift-and-shift existing applications from virtual machines that are currently running in a private datacenter to VMs that run in the cloud. There are many predefined VM images that are ready-to-use. However, running the application in a VM doesn’t provide features like zero-downtime deployments or easy authentication. The operation team is also responsible for patching the operating system and making sure that antivirus software is up-to-date. Azure Virtual Machines, Amazon EC2 and Google Compute Engine are such solutions.

All of the aforementioned types are created as monolithic large core applications that contain all of the domain logic. It has components that communicate to one another directly within a single server process (Vettor, 2022).

#### Microservices

A monolithic application is a solitary, integrated unit, whereas a microservices architecture divides it into a number of smaller units. Microservices are an organizational and architectural approach to developing software. According to this approach, software is composed of loosely connected services that are organized around business capabilities and that can be independently deployed and tested (Wolff, 2016). These services communicate with one another via well-defined APIs. Large, sophisticated applications may be delivered quickly, consistently, and reliably thanks to the microservices design. Microservices are technology and language-agnostic, so it is quite possible for a single organization to utilize multiple runtime platforms. Modern cloud platforms have features like scalability, availability, and resilience that can be used to their fullest by microservices (Smith, 2022).

### Domain-Driven Design

Every web service, no matter if it is a monolith or microservices, has a set of characteristics, the most important of which are the amounts of data handled, 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 work with large amounts of data, get great performance, or write code for hardware systems. The only purpose DDD concepts serve is to tackle business logic complexity. The traditional approach, as discussed by T. Erl in his book “SOA Principles of Service Design”, suggests a technical and functional separation of services. E. Evans (2003), on the other hand, says that DDD gives the key ideas needed to separate web services into different parts. The DDD approach provides a means of representing the real world in the architecture, for instance, by using bounded contexts to represent organizational units and also identifying and focusing on the core domain. These characteristics lead to improved software architecture quality (E. Landre, 2016).

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, and delete operations doesn't carry a lot of complexity with it. This case can be managed with simpler approaches. At the same time, an order management 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 is the number of algorithms that need to be implemented to make the software work.

Domain-driven design (DDD) says that use cases should be modeled based on how the business actually works. In the context of building applications, DDD talks about problems as "domains" (César de la Torre, 2022). It calls separate problem areas "bounded contexts" and stresses the need to talk about these problems in the same way.

DDD suggests many technical ideas and patterns to help with the internal implementation. These include domain entities with rich models (no "anemic" domain models), value objects, aggregates, and aggregate root (or root entity) rules. Some people see these technical rules and patterns as hard to learn obstacles that make it hard to use DDD approaches. But the important part is not the patterns themselves, but organizing the code so it is aligned with the business problems (Bill Wagner, 2022).

In the book Patterns of Enterprise Application Architecture, Martin Fowler (2012) displays a diagram that has complexity on the X axis and time and costs on the Y axis. The curve shows that at some point beyond a certain level of complexity following data-centric design patterns, even a small increase in complexity results in a significant peak in costs.



On the other hand, the time and cost of a project designed from a domain-centric perspective tended to grow linearly with complexity but had to deal with quite high startup costs.

In the end, DDD is exactly what its name implies: design that is based on the characteristics of the domain.

#### Ubiquitous Language

Core principles of DDD make it easier for domain experts and software engineers to talk to each other by defining an explicit ubiquitous (universal) language together. This language is made up of relevant domain-specific terms and is used both in domain models and in implementation. It helps bring the domain expert, the designer, and the programmer together so they can work together to build the domain model(s) and then put them into action (Hippchen, Benjamin, 2017). 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. For the idea of a 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 lay the groundwork for efficient communication.

#### Bounded Contexts

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. The focus should always be on the core domain.

These three concepts: ubiquitous language, bounded context, and core domain— are the strategic elements and the most important parts of domain-driven designThe other notions, such as entities, value objects, and repositories, comprise the tactics of how a software project should be built. They are described in Table 2:

The author of the DDD has emphasized the importance of using design patterns to enrich the ubiquitous language since its inception.

subdomains and bounded contexts are different. A subdomain shows how the business or domain activity is broken up, while a bound context shows how the software and its development have been organized. Quite often, these will match up perfectly, but not always.

#### Entities

Even though a DDD app is driven by behavior, objects are still needed. DDD expresses two types of objects: those that are defined by an identity and those that are defined by their values.

An entity is something that can be tracked, located, retrieved, and stored by an identity key. Entities naturally accumulate a ton of functionality because they play such a crucial role in the system. Applying the single responsibility pattern to entities is indeed a nice idea. Anything that doesn't fit that description should be placed elsewhere.

Instead of being defined by their values, entities are things that were defined by a thread of continuity and identity.

#### Value Objects

A value object has very specific characteristics. It is an item that is used to quantify, measure, or characterize a certain topic. Because the property values define it, it ought to be immutable. Value objects may have methods and behavior, but they should never have side effects. Vaughn Vernon says in his book Implementing Domain-Driven Design that value objects should be used instead of entities whenever possible. Even if a domain concept has to be modeled as an entity, the entity should be designed so that it is more likely to hold values than to hold other entities. Value objects are a really good place to put methods and logic because reasoning can be done without side effects, especially with the complications that identity brings along—all those things that make logic tricky.

#### Aggregates

Eric Evans writes in his book Domain-Driven Design that an aggregate is a group of related objects that are changed as a single unit. Aggregates consist of one or more entities and value objects that change together. Aggregates are treated as a unit for data changes. The entire aggregate's consistency needs to be considered before any changes are applied. Every aggregate must have an aggregate root, which is the parent object of all members of the aggregate. In some cases, the aggregate may have rules that make sure all of the objects' data is consistent. Data changes in aggregate should adhere to ACID, which means they should be atomic, consistent, isolated, and long-lasting. It's also up to the aggregate root to keep its invariants, like the number and type of its parts, the same. An invariant is a condition that should always be true for the system to be in a consistent state.

#### Repository

A repository is a set of all the objects of a certain type that can be queried like a collection, but with more options. Repositories provide a common abstraction for all of our persistence concerns. This makes it easy for clients to get model objects and manage their lifecycle. The public interface of a repository very clearly communicates the design decisions. Only certain objects should be accessed directly, so repositories provide and control this access. An important benefit is that repositories make the code easier to test. They reduce tight coupling with external resources like databases, which would normally make unit testing difficult. When the code for accessing data is wrapped up in one or more well-known classes, it is easier and safer to use.

#### Domain Events

Domain events are a critical part of a bounded context. They give a way to talk about important things that happen or change in the system. Then, loosely connected parts of the domain can respond to these events. In this way, the objects that are raising the events don't need to worry about the behavior that needs to occur when the event happens. And likewise, the event handling objects don't need to know where the event came from. Domain events are encapsulated as objects. Vaughn Vernon describes domain events, saying they should be used to capture an occurrence of something that happened in the domain. They should be part of our ubiquitous language. Events are helpful because they signal that a certain thing has happened, so other parts of the system can „listen“ and take action accordingly. A domain event is essentially a message, a record of something that happened in the past.

### Command and Query Responsibility Segregation

Bertrand Meyer first wrote about Command Query Separation (CQS) in his book Object-Oriented Software Construction (2000). The fundamental idea is that an 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;

Based on the CQS principle, Greg Young (2013) introduced the Command and Query Responsibility Segregation (). CQRS 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 (César de la Torre, 2022). The separation aspect of CQRS is achieved by grouping query operations into one layer and commands into 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 (Bill Wagner, 2022). CQRS means having two objects for a read/write operation, whereas in other contexts there is only one. The goal is to have more flexibility in the queries instead of limiting them with constraints from DDD patterns like aggregates (Mike Rousos, 2022). On the other hand, commands, which trigger transactions and data updates, change the state of the system.

..

CQRS stands for Command-Query Responsibility Segregation. There should be two models, one for reads and the other for writes, as opposed to one unified model.

CQRS was introduced by Greg Young back in 2010. Greg based this idea on the command-query separation principle coined by Bertrand Meyer. The command-query separation principle, or CQS for short, states that every method should either be a command that performs an action or a query that returns data to the caller, but not both. So, asking a question shouldn't affect the outcome of the response. Formally speaking, methods should only return a value if they are referentially transparent and don't have any side effects, like changing the state of an object or a file in the file system, etc. To follow this principle, if a method changes some piece of state, this method should always be of type void otherwise, it should return something. This increases the readability of the code base. is not always possible to follow the command-query separation principle, and there will almost always be situations where it would make more sense for a method to both have a side effect and return something. An example here is Stack. Its Pop method removes the element pushed into the stack last and returns it to the caller. This method violates the CQS principle, but at the same time, it doesn't make a lot of sense to separate those responsibilities into two different functions. The relationship between CQS and CQRS is that CQRS takes this same idea and extends it to a higher level. Instead of methods like in CQS, CQRS focuses on the model and classes in that model and then applies the same principles to them. CQRS supports detangling a single, unified domain model and establishing two models: one for managing commands, or writes, and the other for handling queries, or reads. This is similar to how CQS encourages breaking a method into two, a command and a query.

The CQRS pattern has several advantages:

* 1. Scalability - in a typical system, reads outnumber writes, making it crucial to be able to scale each separately.
  2. Performance - In contrast to a single unified model, optimization methods could be used.

The command side and the query side have drastically different needs, and trying to come up with a unified model for these needs is like trying to fit a square peg in a round hole. A distinct approach that makes the most sense in each specific situation could be introduced by focusing on each of them separately. This can be seen as the single responsibility principle being used at the architectural level. In the end, there are two models, each of which does only one thing well.

In summary, CQRS focuses on making decisions that are optimal for various circumstances. For the command and query sides, multiple levels of consistency and different database normal forms could be selected.

## The CAP theorem

The CAP theorem and CQRS have a close relationship. The CAP theorem says that a distributed data store can't guarantee more than two of the following three things at the same time: consistency, availability, and partition tolerance. Every read gets the most recent write or an error if consistency is maintained. Availability means that every request gets a response, even when all nodes in the system are down. Partition tolerance means that the system keeps running even when messages are dropped or delayed between network nodes. Because it is impossible to choose all three, it is necessary to strike the right balance. The fact that CQRS offers a variety of options makes it effective.

CQRS is frequently referred to as an intermediate phase before event sourcing.

### Event-sourcing

Event sourcing is a strategy wherein the applications store the transactions but not the states. All the transactions since the beginning of time are applied when state is required (Martin, R. C., 2017). Nothing gets deleted or updated from the data store. Because neither updates nor deletions occur in the data store, there cannot be any concurrent update issues.

Event sourcing is a way of designing based on the idea that all changes to the state of an application over the course of its lifetime are stored as a series of events.  It ends up having serialized events as the building blocks of the application. The application's data source is essentially a stream of serialized events.

Most applications today work by storing the current state of domain entities and using that stored state as a starting point for business transactions. Rather than storing all of the information in the columns of a single record or in the properties of a single object, the state of the entities is described by the sequence of events that led to it containing a given list of items. This is actually an event-based representation of an entity.

An event is something that has happened in the past. Events are an expression of the ubiquitous language. Events could be kept in a NoSQL database, an ad-hoc relational table, or with a particular product. An event store of any kind is an append-only store and doesn't support deletions. Events express the entire state of a domain entity. To get the full state, the application timeline should be replayed from the beginning. This could sometimes involve handling excessive amounts of data. In this case, define snapshots, which are the state of the entity at a given time. Once stored, events are immutable. For scalability purposes, duplicate and replicate events are possible. With events, everything that occurred at the specific moment it occurred is recorded, regardless of the results it had. A lower abstraction level is used to store any system data.

Storage can be relational, document-based, graph-based.

The persistence of messages, which records all alterations to the application's state, is the key component of event sourcing.

The state of an aggregate can be recreated using recorded events.

Replay is just about looking into this data and performing logic to extract information from it.

Ad-hoc projections can address other, more interesting scenarios, like business intelligence, statistical analysis, what if, and why not simulation.

Yes, it might be hard and impractical to project state from logged events if there are a lot of them, and the number of logged events in many applications can only grow over time because it's an append-only store.

Instead of processing the entire stream of events, you serialize the state of aggregates at a given point in time and save that as a value. Next, you keep track of the snapshot point and replay events for an aggregate from the latest snapshot to the point of interest.

## III. Dealing with the business complexity

### Designing a DDD-oriented service

The key task when designing and defining a service is determining where to draw the boundaries. DDD patterns help with understanding the complexity of the domain. Each bound context finds the entities and value objects, describes them, and puts them together to make a model of the domains. Choosing where to draw the line between bound contexts requires balancing two competing goals. The first one is creating a boundary around things that need cohesion. The second goal is to avoid chatty communications between units. These goals can contradict each other. Balance should be achieved by breaking the system down into as many small units as possible. Cohesion is key within a single-bound context. It is similar to the inappropriate intimacy code smell (Alexander Shvets, 2018) when implementing classes in the object orient programming. Another way to look at this aspect is autonomy. A unit is not truly autonomous if it must rely on another unit to directly service a request.

### Onion Architecture

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

### Layers in DDD

Most enterprise applications with significant business and technical complexity are defined by multiple layers. (Mike Rousos, 2022). The layers are a logical artifact that has nothing to do with how the service is deployed. They exist to help developers manage the complexity of the code. Different layers may have different types, necessitating translations between them. A domain model must be controlled by aggregate roots that ensure that all invariants and rules related to that group of entities are performed through a single entry-point or gate (Bill Wagner, 2022).

#### Domain model layer

This layer is in charge of representing business concepts, business situation information, and business rules (Eric Evans). The domain model layer is where the business is expressed; it is the heart of business software. According to the Persistence Ignorance (todo) and Infrastructure Ignorance (OREN EINI, 2008) principles, this layer must not know anything about how data is stored. The infrastructure layer should be in charge of these persistence tasks. Domain entities shouldn't directly depend on any data access infrastructure framework, like by inheriting from a base class. Even though the persistence ignorance principle is important for the domain model, concerns about persistence should not be ignored. It is still important to understand the physical data model and how it maps to the entity object model; otherwise, impossible designs would be created (César de la Torre, 2022).

#### Application layer

The application layer defines the functions of the software and directs the expressive domain objects to solve problems. This layer is in charge of tasks that are important to the business or are needed to work with the application layers of other systems. This layer is kept thin. It contains no business rules or knowledge, but only coordinates tasks and delegated work to domain object collaborations in the next layer down. It does not have a state reflecting the business situation, but it can have a state that reflects the progress of a task for the user or the program (Eric Evans). It gives the task of executing business rules to the domain model classes (aggregate roots and domain entities), which will then update the data in those domain entities. The goal is for the presentation and application layers to have nothing to do with the domain logic in the domain model layer, its invariants, the data model, or any business rules that go with it (Bill Wagner, 2022).

#### Infrastructure layer

The infrastructure layer is where data from domain entities is stored in databases or another persistent store. An example is using object-relational mapping framework code to implement the repository pattern classes. According to the persistence and infrastructure ignorance principles, the infrastructure layer must not "contaminate" the domain model layer. The entity classes in the domain model must be isolated from the infrastructure.

#### Dependencies between layers

The following diagram depicts the interdependence of the three layers mentioned above:



Figure 7-7. Dependencies between layers in DDD

The application layer depends on domain and infrastructure, and infrastructure depends on domain, but domain does not depend on any layer.

### Principles

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.

### Test-Driven Development

Test-Driven Development is a software practice in which a failing test is created first before any production code is written and used to drive the design of the architecture. This three-step process of Test-Driven Development is referred to as "red, green, and refactor.":

red step - creating a failing test for the simplest piece of functionality

green step - Implementing enough production code to get that failing test to pass.

refactor step - improve both the test and production code to keep the quality high.

This cycle is repeated for each piece of functionality in order of increasing complexity in each method and class until the whole feature is finished.

A comprehensive collection of tests is produced by using test-driven development, which includes all of the application's code that is only crucial in a passive sense.

Additionally, by using TDD, the testing process is what guides the design of each of these classes and methods. Testable code is what produces maintainable code.

There are a variety of types of tests that exist in the world of software testing.

Some tests are based on the subject matter. Unit, integration, component service, and user interface tests, for instance. Some are determined by the purpose of the test. For example, functional tests, acceptance tests, smoke tests, and exploratory testing. Others, though, are determined by how they are being tested: automated, semi-automated and manual tests.

Mike Cohn talks about what he calls the "test automation pyramid" in his book Succeeding with Agile. Four different test kinds are identified:

1) unit tests - automated tests that check how well a single piece of code works on its own;

2) service tests - automated tests that check how well a group of classes and methods that provide a service to users work;

3) UI tests - automated tests that check that the whole application works (from the user interface to the database);

4) Manual tests - tests done by a person, also check the full application's functionality;

The test automation pyramid captures the essence of how each type of test becomes more expensive. As a result, the system should have a large number of low-cost tests and a small number of high-cost tests.

### Clean Architecture

A simple, intuitive, flexible, testable, and maintainable software architecture should be created and designed using the contemporary patterns, techniques, and principles mentioned above.

Clean code reads like well-written prose. Clean code never obscures the designer’s intent but rather is full of crisp abstractions and straightforward lines of control (Booch et al., 2007).

"Clean architecture" is a philosophy of architectural essentialism and mainly a cost-benefit argument.

Users' use cases and mental models need to be reflected in the system. And that is what clean architecture focuses on.

Clean architecture builds only what is necessary, when it is necessary and optimizes for maintainability.

### Limitations

The DDD patterns presented in this article should not be applied universally. They introduce constraints, which 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 (César de la Torre at all, 2022). Example: The aggregate pattern treats many domain objects as a single unit as a result of their relationship in the domain. If you treat multiple objects as a single aggregate for read-only queries, there is no benefit, but it can increase the complexity of query logic.

It's important to emphasize that CQRS and most DDD patterns (like DDD layers or a domain model with aggregates) are not architectural styles but only architecture patterns. Microservices and SOA 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 (Bill Wagner, 2022). At an architecture pattern level, the design of each bound context in that application shows its own trade-offs and internal design decisions.

## IV. Conclusion and Future Work

….

## REFERENCES

[1] R. Steinegger, P. Giessler, B. Hippchen, and S. Abeck, “Overview of a Domain-Driven Design Approach to Build Microservice-Based Applications,” in SOFTENG: The Third International Conference on Advances and Trends in Software Engineering, April 2017.

[2] S. Newman, Building Microservices, 1st ed. O’Reilly Media, Inc.,2015.

[3] M. Richards, Microservices vs. Service-Oriented Architecture. O’Reilly Media, Inc., 2015.

[4] T. Erl, SOA Principles of Service Design (The Prentice Hall Service-Oriented Computing Series from Thomas Erl). Upper Saddle River,NJ, USA: Prentice Hall PTR, 2007.

[5] E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional, 2003.

[6] Dive Into Design Patterns by Alexander Shvets Published December 5th 2018 by Refactoring.Guru

[7] <https://ayende.com/blog/3137/infrastructure-ignorance>

A[8] Mukherjee, Momin. (2016). Object-Oriented Analysis and Design. International Journal of Advanced Engineering and Management. 1. 18-24. 10.24999/IJOAEM/01010003.

[9] Object-Oriented Analysis and Design with Applications by Grady Booch (Author), Robert A. Maksimchuk (Author), Michael W. Engle (Author), Bobbi J. Young / Publisher Addison-Wesley Professional

[10] Martin, Robert C.. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Boston, MA: Prentice Hall, 2017.

Booch, G., Maksimchuk, R. A., Engle, M. W., Young, B. J., Conallen, J., & Houston, K. A. (2007). Object-Oriented Analysis and Design with Applications (3rd ed.). Addison-Wesley Professional.

Wolff, E. (2016). Microservices: Flexible Software Architecture. Microservices: Flexible Software Architecture. Addison-Wesley Professional.

Architecting Cloud Native .NET Applications for Azure / Steve "ardalis" Smith, Rob Vettor, / Redmond, Washington 2022 by Microsoft Corporation

Contents

[Applying the Domain-Driven Design in Cloud-Native Services 1](#_Toc124074278)

[Abstract 1](#_Toc124074279)

[I. Introduction 1](#_Toc124074280)

[II. Theoretical foundation 2](#_Toc124074281)

[A. Cloud-Native Services 2](#_Toc124074282)

[B. Domain-Driven Design 4](#_Toc124074283)

[C. Command and Query Responsibility Segregation 7](#_Toc124074284)

[The CAP theorem 8](#_Toc124074285)

[D. Event-sourcing 8](#_Toc124074286)

[III. Dealing with the business complexity 9](#_Toc124074287)

[A. Designing a DDD-oriented service 9](#_Toc124074288)

[B. Onion Architecture 10](#_Toc124074289)

[C. Layers in DDD 11](#_Toc124074290)

[D. Principles 12](#_Toc124074291)

[E. Test-Driven Development 13](#_Toc124074292)

[F. Clean Architecture 13](#_Toc124074293)

[G. Limitations 14](#_Toc124074294)

[IV. Conclusion and Future Work 15](#_Toc124074295)

[REFERENCES 16](#_Toc124074296)