# 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

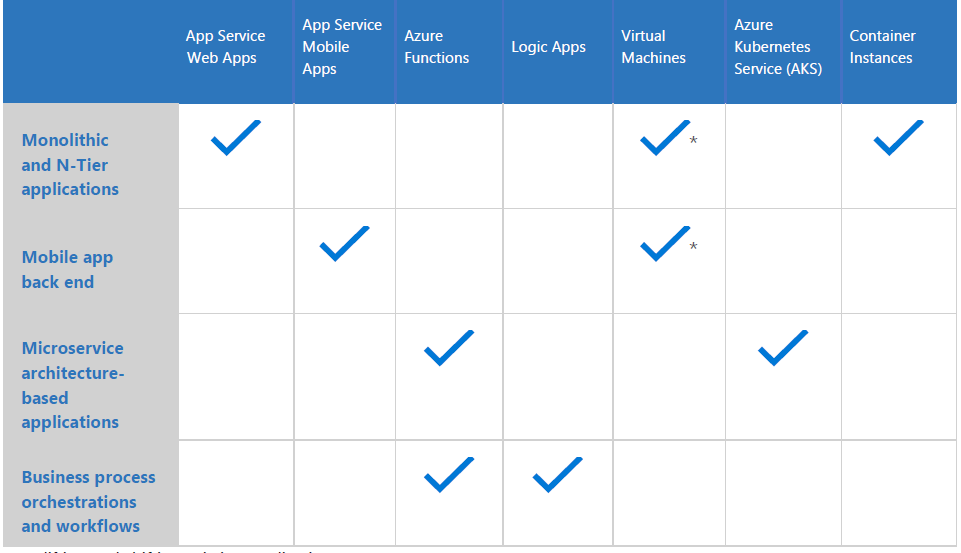
This section introduces the design and implementation of domain-driven design patterns.

### 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

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.

#### Microservices

.....

### Domain-Driven Design

Every web service, no matter if it is 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.

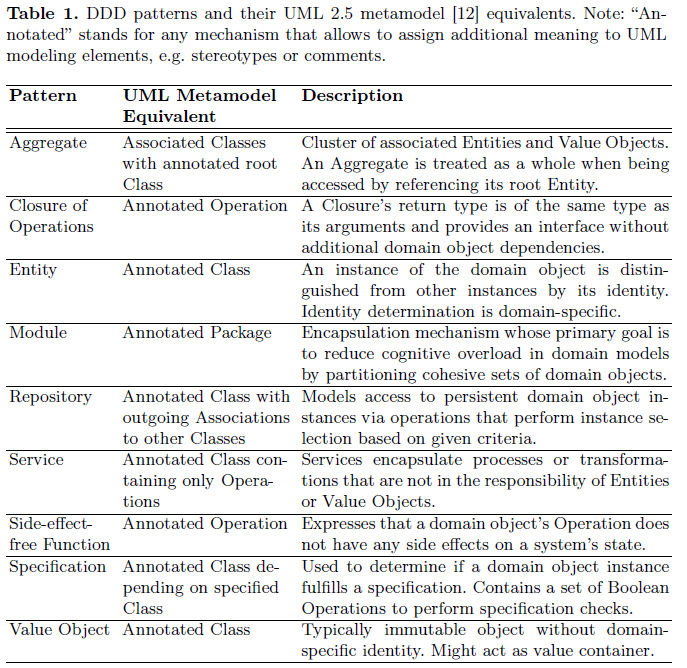
Domain-driven design (DDD) says that use cases should be modeled based on how the business actually works. In the context of building applications, DDD talks about problems as "domains" (César de la Torre, 2022). It calls separate problem areas "bounded contexts" and stresses the need to talk about these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

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.

### 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.

### Event-sourcing

## 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.

### 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.

## 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>