# 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.

## I. Introduction

….

## II. Theoretical foundation

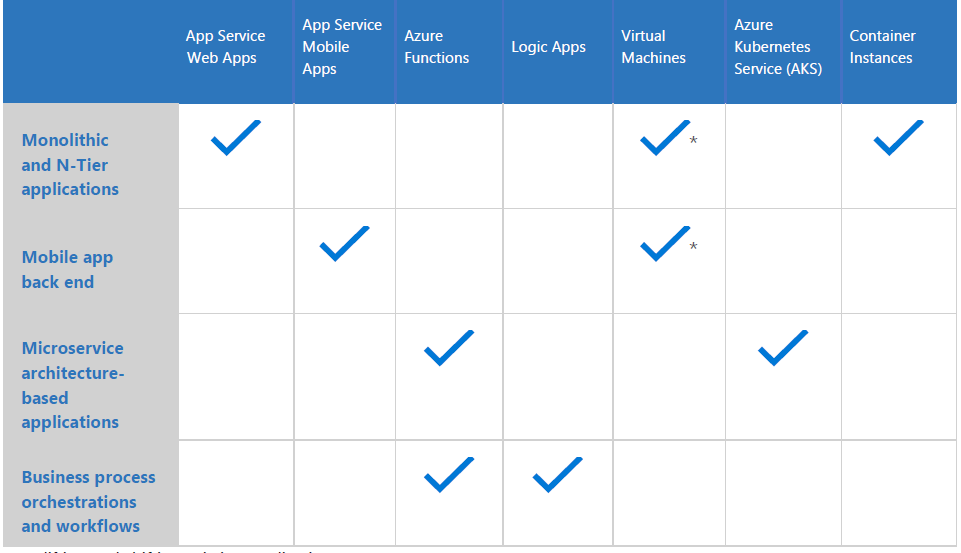
This section introduces the design and implementation of domain-driven design patterns.

### 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.

### Domain-Driven Design

Every software project 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. 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 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, described in Table 1. 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.

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). 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 (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

### Testing

## III. Dealing with the business complexity

Asdasdadas

## 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.