Domain Driven Design Approaches in Cloud Native Service Architecture

Jordan Jordanov [1], Pavel Petrov [1]

[1] *University of Economics - Varna, Varna, Bulgaria*

Abstract **– With the proliferation of cloud native services, the need for efficient software design strategies has become of the utmost importance. The hypothesis of this article is that domain driven design approaches, when integrated into cloud native service architecture, provide a valuable methodology for building modular, scalable, and maintainable systems. The goal of the article is to analyse how these approaches can improve software design while also contributing to system availability, reliability, and resilience. The methodology employed in this study involves the analysis of domain-driven design approaches and their integration with cloud native technologies. The paper emphasizes the importance of clean domain models, well-defined bounded contexts, and the separation of concerns in enterprise-grade software. While focusing on foundational concepts, the paper suggests the potential for a future case study to illustrate the domain driven software development process in action. While the paper does not provide specific empirical results, it highlights the potential benefits of adopting domain-driven design and cloud native architectures. That is why the article examines the fundamental components of domain driven design, their integration with cloud native technologies, benefits, and challenges.**

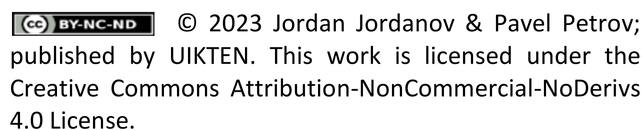
In addition, the study sets the stage for further research in this area to help software architects and developers.

Keywords **– Domain driven design, cloud native services, distributed systems, software architecture.**

DOI: 10.18421/TEM124-09
https://doi.org/10.18421/TEM124-09

Corresponding author: *Pavel Petrov,*
University of Economics - Varna, Varna, Bulgaria
Email: petrov@ue-varna.bg

Received: 03 July 2023.
Revised: 05 September 2023.
Accepted: 11 September 2023.
Published: 27 November 2023.



© 2023 Jordan Jordanov & Pavel Petrov; published by UIKTEN. This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 4.0 License.

The article is published with Open Access at https://www.temjournal.com/

1. Introduction

Cloud services have revolutionized the production and deployment of software systems. These services take advantage of the agility, adaptability, and fault tolerance that cloud platforms offer [16]. Even so, there are unique challenges associated with leading organizations to developing and operating applications in a dynamic environment.

The domain driven design (DDD) is a software development methodology that prioritizes the business domain as the driving force behind architecture design [27]. From a business perspective, a domain is defined as a "field or industry in which a business operates, composed of multiple subdomains". There are three categories of subdomains: generic, core, and supporting [17]. It is well known that businesses invest in software to meet specific requirements or address specific problems. For an in-depth understanding of the problem, architects must first grasp the domain. Core DDD principles include capturing valuable domain knowledge in code models, which can include both structural and behavioural aspects, in a collaborative mode between domain experts and software engineers [15]. DDD provides conditions and activities for constructing a domain model as the primary artifact [2]. In this context it is important to examine the potential of DDD as a guiding principle for designing cloud native services to optimize development processes.

A list of essential concepts for designing robust, scalable, and secure cloud-based systems is presented in Table 1. Each principle may be used as a solution to a commonly occurring problem.

Table 1. List of key design principles

Name	Description
Separation of Concerns	A design guideline for dividing distinct sections of a computer program. Each module and object must have its own purpose and context. This leads to more opportunities for module development, reuse, and autonomy.
Encapsulation	A way to restrict direct access to certain segments of an element so that people cannot view the state values of all of an object's variables. Encapsulation can be used to cover up both the data members and the data functions or methods.
Single Responsibility	The basic concept asserting that a "module should only be accountable to a single actor." [20]. To put it another way, each piece in the design must have a single purpose. Single responsibility is closely related to the concepts of coupling and cohesion.
Dependency Inversion	Research by R. C. Martin [19], [20] shows that this principle is a specific way to loosely connect software modules. It specifies that high-level modules should not rely on low-level modules. Each must rely on abstractions. In other words, the principle suggests that classes or modules should rely on abstractions (interfaces or abstract classes) instead of actual implementations.
"You Are Not Going to Need It" (YAGNI)	A fundamental principle of extreme programming [32]. YAGNI says, "Do not add functionality unless it is considered required." In other words, create the code required for the given circumstance. One must not add anything that is unneeded. When adding logic to the code, one should not take into account what may be required in the future.
"Keep It Short and Simple" (KISS)	This idea relates to the simplification of functionality implementation. Less complicated code is easier to read and hence easier to maintain.
Factory	This is one of the well-known Gang of Four design patterns. It offers an interface for constructing objects without specifying their classes. It encapsulates the logic of object construction within a distinct factory class.

All the patterns, techniques, and principles are geared toward the design and development of simple, intuitive, flexible, testable, and maintainable cloud software architectures. The architectures have a high level of abstraction and a long-term focus for solution components.

Their design is comprehensive, and implementation focused. Clean architecture [20] is a philosophy of architectural essentialism and operates mainly according to a cost-benefit analysis. The clean architecture centers around ensuring that the system accurately mirrors the users' use cases and mental models. It builds only what is necessary when it is necessary and optimizes it for maintainability. The topic of clean architecture is also connected to the notion of "clean code" [19]. Clean code is straightforward, easy to understand and "reads like well-written prose". Clean code never hides the programmer's intent and is replete with clear abstractions and control flow [5].

When creating a cloud solution, one of the first decisions to make is which service(s) to utilize in order to operate the applications [7]. Table 2 shows the choices for which cloud services are best for which types of applications.

Table 2. Cloud services' suitability for various application types [7]

	Web service	Mobile service	Serverless	Virtual Machine	Microservices
Monolithic and N-Tier app	✓			✓	
Mobile app back end	✓	✓		✓	✓
Distributed system			✓		✓

One of the simplest and most effective solutions for managing cloud-based apps is the HTTP-based service for hosting web applications. Some examples of this service are Azure App Hosting Service, AWS Elastic Beanstalk, and Google App Engine. They provide a set of hosting services that cover the complexity of the operating system and infrastructure while hosting an application. They are highly available by default and are operational at least 99.95% of the time. They share potent characteristics such as automatic scaling, zero-downtime deployments, and straightforward authentication and authorization [7], [35]. Some of them enable debugging the application while it is in production, using tools such as Snapshot Debugger.

When developing a mobile application, a backend that the application can connect to is required. Typically, this is an application programming interface (API) that the application can utilize to access and store data [29]. Azure Mobile Apps and AWS Amplify provide such solutions with unique capabilities.

For example, there is an offline sync that enables a mobile app to keep functioning if there's no connection to the backend, and the sync is refreshed whenever the connection is re-established. Another feature is sending push notifications to the mobile apps, regardless of the platform they run on (iOS or Android), with services such as Firebase Cloud Messaging, Azure Notification Hubs, and Apple Push Notification Service.

Serverless functions, also known as “function as a service” (FaaS), are a cloud computing paradigm that allows developers to compose and deploy individual functions or code fragments without managing or provisioning servers [18]. In a serverless setup, the cloud provider handles server administration, scalability, and infrastructural duties, freeing developers to concentrate on writing and deploying code [35].

Existing applications could be lifted and relocated from virtual machines (VMs) operating in a local data centre to VMs running in the cloud, making this a simple approach to getting started. There are many predefined VM images that are ready to use. Even so, running the application in a VM does not offer any optimizations. The operation staff is also accountable for maintaining the operating system and anti-virus software [7]. Azure Virtual Machines, Amazon EC2, and Google Compute Engine are examples of such solutions.

All the aforementioned types are created individually as monolithic, large-core applications that contain all of the domain logic. They have components that communicate with one another directly within a single server process [35]. A monolithic application is a single, integrated unit, whereas microservices divide the application into several 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. These services communicate with one another via well-defined APIs [29]. Large, sophisticated applications may be delivered quickly, consistently, and reliably. 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 such as scalability, availability, and resilience that can be used to their fullest potential by microservices [35]. Such cloud solutions are Azure Kubernetes Service, Amazon EC2 & EKS, Google Kubernetes Engine, Red Hat OpenShift, DigitalOcean, and many more. Microservices architecture is a catalyst and enabler for continuous business transformation [14].

This paper begins with a detailed description of DDD, highlighting its fundamental principles and advantages. This will lay the groundwork for understanding how DDD can be implemented effectively in a cloud native service architecture. Next, we will investigate how Command Query Responsibility Segregation and Event Sourcing, which have strong ties to DDD, can enhance the application code further. In the end, we will examine how DDD and Test-Driven development can significantly benefit software development when used together.

2. The Features of Domain-Driven Design in the Context of Cloud Services

A web service, whether a monolith or part of a distributed system, has certain features such as the volume of information handled, efficiency, business logic, and technological upgrading [25], [33]. DDD strategies are beneficial for initiatives with a large number of complex business principles, because they can simplify the business logic. In other words, the primary objective of DDD concepts is to deal with the complexity of domain logic, which consists of business rules, validations, and calculations [17].

The classic approach incorporates the separation of services based on their technical and functional characteristics [10]. It focuses on core capabilities exposed as services. E. Evans [11], [12], on the other hand, states that DDD provides the key ideas needed to separate web services into different parts. The DDD methodology offers a way of expressing the actual world through a structured representation of a solution that meets the requirements in the problem space. These characteristics lead to improved software architecture quality.

The focus should always be on the core domain. Business logic complexity is the first indicator of how complicated the problem domain in which a software works is. A simple application that needs to perform fundamental create, read, update and delete operations (CRUD), is not particularly complex [8]. This situation can be handled with less complicated methods. Simultaneously, an order management system, which automates a significant portion of a company's activity, must model all the processes upon which the company acts and therefore manage a large number of complex business responsibilities. This system's business logic complexity may be extremely high. Another attribute is its technical complexity, a term that refers to the number of algorithms that need to be implemented to make the software work.

Martin Fowler [13] presents a diagram (Figure 1) with time and cost on the Y axis and complexity on the X axis.

In accordance with data-centric design patterns, the curve indicates that beyond a certain level of complexity, even a small increase in complexity results in a significant cost peak.

On the other hand, the time and cost of a project designed from a domain-centric perspective tended to increase linearly with complexity, whereas the start-up costs were quite high. According to DDD, use cases should be modeled based on the way the business actually operates, which is always evolving.

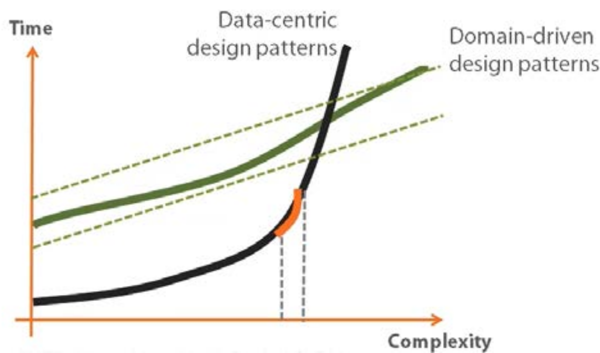


Figure 1. Domain-centric versus data-centric in the context of a software development diagram depicting time and complexity [13]

DDD offers a variety of technical concepts and patterns to assist in the internal implementation [38]. Ubiquitous (universal) language (UL), bounded context (BC), core domain, entities, value objects, aggregates, and repositories, are the steps for building a software project. Some individuals view these technical rules and patterns as difficult-to-learn obstacles that make it challenging to employ DDD methodologies. However, the most critical aspect is arranging the code so that it matches the business problems [23].

Each industry and profession have its own terminology. To build complex systems, IT teams must learn the business terminology used by the relevant stakeholders. A core principle of DDD is to make it easier for domain experts and software engineers to talk to each other by defining an explicit UL. This language assists in bringing together the stakeholder, the designer, and the programmer so that they may construct the domain model(s) and then put them into action [3]. Code written in the UL can provide a hint for some edge cases that were not clear enough at the start. For the idea of a UL 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 UL. Common nomenclature facilitates the understanding of user requirements. Batista's research [3] indicates that this lays the groundwork for productive interaction, so he seeks to develop a standard, business-oriented language, with the primary objective of preventing misunderstandings and incorrect assumptions.

UL is utilized in documentation, conversations, app code and testing code and is used by domain experts and, delivery teams. UL evolves over time and may be managed on any knowledge collaboration platform. It helps in identify focus areas for knowledge crunching, which is the process of "coping" the knowledge received from the experts into domain models [27].

The BC is a small area within the domain that gives each element of the UL its own meaning [36]. Quite often, an application's code base becomes unmanageable as its volume increases. A BC illustrates how the program, and its development were structured. Frequently, it corresponds to a subdomain, which indicates how the business or domain activity is divided [23]. Each BC is developed independently. The domain model built for a BC is applicable only within its boundaries.

A context map facilitates the identification and management of interdependencies and collaborations among BC [2]. It enables teams to comprehend the structure of the larger system and understand how their individual contexts integrate into the bigger picture.

Even though a DDD application is governed by behaviour [15], objects are still required. DDD conveys distinct types of objects, characterized by their identities or values.

An **entity** represents a uniquely identifiable business object that encapsulates attributes and a well-defined domain behaviour [2]. The definition of an entity consists of attributes and behaviour. An entity is something that can be tracked, located, retrieved, and kept in long-term storage.

Value objects are small, simple objects whose equality is not based on identity [2]. They are items used to quantify, measure, or characterize a certain topic. Value objects may have methods and behaviours, but they should never have side effects. Vernon [34] says that value objects should be used instead of entities if possible.

An **aggregate** is a collection of connected items that are modified as a single entity. Aggregates are treated as a unit for data changes. They consist of one or more entities and value objects that change together. Before making modifications, it is necessary to evaluate the consistency of the whole aggregate [36]. Every aggregate must have an aggregate root, which is the parent object of all members. In some cases, the aggregate may have rules that ensure all of the objects' data are consistent. Data changes in aggregates should adhere to ACID, which means they should be atomic, consistent, isolated, and long-lasting. The factory pattern can be used for creating complex aggregates [2].

A **repository** is a collection of items of a particular type. Repositories offer a unified abstraction for all persistence-related problems. This makes it easy for clients to obtain and manage model objects. The public interface of a repository communicates design decisions very clearly. Few objects ought to be directly accessible; consequently, repositories provide and regulate this access. An important benefit of repositories is that they make the code easier to test. They reduce the tight coupling with external resources such as databases and data providers, which would traditionally make unit testing challenging. When code for data access is wrapped in one or more well-known classes, it is easier and safer to use.

Vernon describes **domain events**, saying they should be used to capture an occurrence of something that happened in the domain, and should be part of the UL [34]. Events are helpful because they signal that a certain thing has happened. A domain event is essentially a message, a record of something that happened in the past.

Model-driven design (MDD) provides a framework for the implementation of modeled systems. The previously listed elements of construction have relationships. MDD expresses state and computation through value objects, identity through entities, and change through domain events [28]. Repositories permit access to entities and aggregates. Except for the events, they can all be encapsulated in a factory.

3. Managing the Complexity Issues in Cloud Services Through Layered Approach

DDD concepts create a structure known as onion architecture [24]. The word onion is used because the architecture has numerous layers and a central core. The top layers are dependent on the bottom layers, yet the bottom layers have no knowledge of the top ones. Onion architecture illustrates that the fundamental elements of the DDD should operate independently of one another.

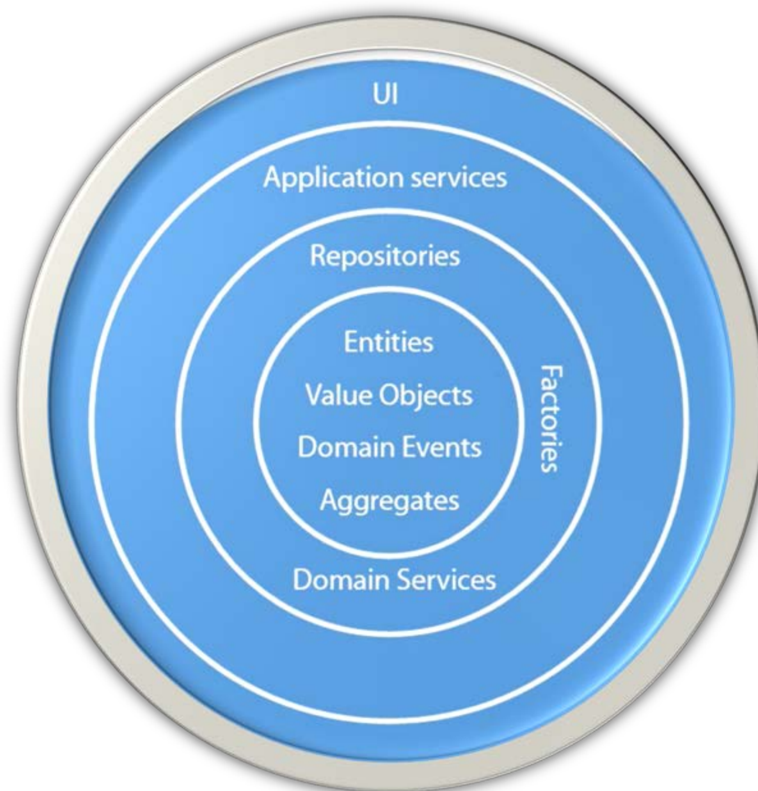


Figure 2. The fundamentals of DDD in onion architecture [24]

The middle section consists of notions including entity, value object, domain event, and aggregate that are connectable but unable to interact with the elements from the upper level. The following layer includes repositories, factories, and domain services; they may be aware of one another and the four fundamental components, but they should not mention the application services [15]. User interface and application services are on top.

The main reason for this isolation is to allow the separation of concerns.

The most important aspect of designing and establishing a service is setting its boundaries. Each BC identifies the entities and value objects, characterizes them, and combines them. Choosing where to draw the border between BCs requires balancing two competing objectives.

Creating a barrier around items that need cohesion is the first step. The second goal is to avoid "chatty" inter-unit communications. These objectives may conflict with each other. Balance should be accomplished by decomposing the system into the smallest units feasible. In a single-bound context, cohesion is crucial. Another way to look at this aspect is to view it as autonomy. A unit is not completely autonomous if it relies on another unit to fulfil a request directly.

The majority of enterprise applications have distinct tiers [8]. They help developers manage the complexity of the code [17]. MDD isolates domain expression using layers. Those layers have nothing to do with the deployment of the service. When DDD principles are employed, the elements may be organized differently depending on the specific implementation. Nonetheless, as shown in Figure 3, there are a few common layers.

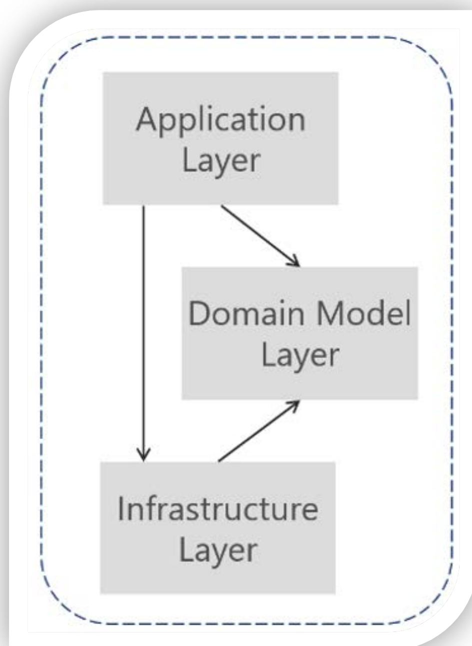


Figure 3. Dependencies between layers in DDD [8]

The **application layer** coordinates the execution flow between various domain objects/entities to solve problems. It also specifies the use cases and operations that can be carried out within the service and orchestrates interaction between the UI and the core elements. Commonly, the application layer is implemented as a web API or an MVC project. The application layer depends on domains and infrastructure.

The **domain model layer** encapsulates the business logic and principles and constitutes the core of the service. It contains domain objects/entities, aggregates, value objects, and domain services.

The domain layer concentrates on solving business problems and expresses the business domain's concepts and behaviours. This layer should have entirely decoupled and simple class objects to implement "the heart of the software" from a code perspective. The domain layer does not depend on any other.

The **infrastructure layer** is responsible for providing the domain layer with the necessary technical facilities and support. The infrastructure layer's primary function is to abstract and encapsulate technical details and complexities. It provides implementations for multiple concerns, including data persistence, messaging, network communication, integration with external services, caching, and performance optimization.

4. Using Command and Query Responsibility Segregation and Event Sourcing in Cloud Services

Greg Young [8] introduced command and query responsibility segregation (CQRS) in 2010 as an extension of the DDD principles. Young based this idea on Bertrand Meyer's command-query separation principle [21]. Command-query separation (CQS) states that every method must either be a command that executes an operation that modifies the state of the system or a query that provides data to the caller, but not both [23]. Therefore, asking a question should not affect the outcome of the response. Methods should only return a value if they are referentially transparent and do not have any side effects, such as changing the state of an object or a file in the file system. To follow this principle, if a method changes some piece of state, it should always be of type void. This increases the readability of the code base. However, it is not always practical to stick to the CQS paradigm. There are occasions when it makes more sense for a method to have both a side effect and a return value. One example of this is the linear data structure stack. Its pop method removes the element last pushed into the stack and returns it to the caller. This solution violates the CQS concept yet separating these duties into two distinct functions is illogical.

The relationship between CQS and CQRS is that the latter extends the same notion as the former to a higher level. CQRS is seen as an architectural pattern. Instead of focusing on methods such as CQS, CQRS applies the same principles by facilitating the separation of operations [28]: one for command management, or writes, and the other for query processing, or reads. CQRS is an object-oriented expression of the domain and is frequently associated with more complex business contexts.

Typically, it is difficult to create one specific unified model since retrieving and persisting data have very distinct needs. By concentrating on each command and query case individually, one can develop a different strategy that makes the most sense. In the end, there are two models, each of which specializes in a certain purpose. Separation is accomplished through clustering query activities into one composition and commands into another. Each one has a unique data model [8]. The application layer turns any input into a command or a query and sends it to a shared communication channel (message handler). The three main categories of messages in an application are commands, queries, and events. They are all part of the core domain model, located in the centre of the onion architecture. Commands tell the application to do something; queries ask it about something; and events are informational messages. Commands trigger a reaction in the domain model, while events are the result of that reaction. Naming guidelines are associated with UL and all three types of messages, with commands always being in the imperative tense, queries usually starting with the word GET, and events always being in the past tense.

In addition, the query and command handlers can be implemented within the same tier or on distinct services so that they can be autonomously tuned and developed by not harming each other, offloading, the complexity from the code base [8]. This can be seen as the single responsibility principle being used at the architectural level.

The CAP theorem and CQRS have a close relationship. The CAP theorem, also known as Brewer's theorem, is a fundamental principle in distributed computing that asserts that it is not feasible for a distributed system to guarantee all three of the following capabilities simultaneously: consistency, availability, and performance [6]. If consistency is maintained, every read operation returns the most recent write or an error. Availability, on the other hand, implies that every request receives a response, even if all system nodes are down. With partition tolerance, the system continues to function even when communications are lost or delayed across network nodes. Due to the impossibility of choosing all three options, it is necessary to reach a compromise. CQRS is effective because it provides numerous opportunities by emphasizing optimal decision-making in various circumstances.

By adopting CQRS, developers can design cloud native services that efficiently handle high query loads while ensuring data consistency through strict command processing. CQRS is commonly referred to as an interim stage preceding event sourcing. Event sourcing complements CQRS by collecting all system state changes as a series of events.

Event sourcing is a design technique based on the concept that all changes to the state of an application throughout its lifetime are recorded as a series of events. As a result, serialized events become the fundamental building blocks of the application. In the event sourcing approach, the programs store transactions but not their respective states. When a state is needed, all transactions from the beginning of time are applied. Nothing is deleted or updated from the data repository. Because of this, there cannot be any concurrent updating issues. Most applications work by storing the current state of domain entities and starting business transactions. Instead of storing all 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. This is an event-based representation of an entity. As described above, an event is something that occurred in the past and is an expression of the UL.

Event storage may be relational, document-based, or graph-based [9]; therefore, events may be stored in an SQL or NoSQL database [26], [30] such as PostgreSQL, MySQL, MongoDB, or Apache Cassandra, or they may be stored using a specific solution such as RavenDB or FaunaDB. Table 3 presents some examples of cloud-based options.

Table 3. Suitability of cloud-based storage options for various business cases [7], [16]

	Relational	Unstructured	Semi-Structured	Tuneable Consistency	Geo-Replication	Large Data
Azure SQL	✓				✓	
Azure Cosmos		✓	✓	✓	✓	✓
Azure Blob		✓			✓	✓
Amazon RDS	✓				✓	✓
Amazon Dynamo		✓	✓	✓	✓	✓
Amazon S3		✓			✓	✓
Google SQL	✓				✓	✓
Google Firestore		✓	✓	✓	✓	✓

As objects, domain events are an integral component of a BC. They provide a way to talk about important things that happen or change in the system, and then, loosely connected parts of the domain can respond to these events. In this manner, the objects that raise the events do not need to consider the action that must occur when the event occurs. Similarly, event-handling objects do not need to know where the event originated.

To obtain the entire state, it is necessary to replay the program timeline from the beginning.

Using recorded events, it is possible to reconstruct the state of an aggregate. This may sometimes require the management of huge volumes of data. In this case, snapshots, which represent the state of the entity at a certain point in time, may be specified [37]. Once stored, events are immutable. It is possible to duplicate and repeat events for scalability reasons.

The replay algorithm involves examining the data and using logic to retrieve the relevant information. Other, more intriguing situations, such as business intelligence, statistical analysis and tracking the history of a resource, may be addressed by ad hoc projections. Events also, provide a powerful and efficient approach to data warehousing, supported by cloud services such as Amazon Redshift, Google BigQuery, and Azure Synapse Analytics.

5. Applying Test-Driven Development Practice in Cloud Services

Test-driven development (TDD) and DDD are two potent methodologies that, when combined, can increase the quality of cloud services and the development process. By employing these practices, developers and quality assurance engineers can create a system that is more robust and reliable. TDD encourages a rigorous testing process in which tests are written prior to the implementation code; this process follows best practices, ensuring that the intended functionality is met. There is a three-step procedure known as red, green, and refactor [22]. Creating a failing test for a piece of functionality is the initial step. The second phase is the green step, during which sufficient production code is created to make the failed test pass. Refactoring is the last phase in which both test and production code are enhanced to maintain high quality. 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. The use of TDD ensures that the testing process is what guides the design. Testable code is what produces maintainable code [4].

In the field of software testing, there are several different sorts of tests. Some tests are subject matter based – e.g., unit, integration, component service, and user interface testing. Meanwhile, others are determined by the purpose of the test – e.g., functional tests, acceptance tests, smoke tests, and exploratory testing. Still others, are determined by how they are being tested – e.g., automated, semi-automated, and manual tests.

The test automation pyramid (Figure 4) depicts the types of tests that should be performed at various stages of the software development lifecycle and how often they should occur in a testing suite to ensure the quality of the program [17].

The notion behind the pyramid is that testers should devote more effort to basic tests before moving on to more complicated ones.

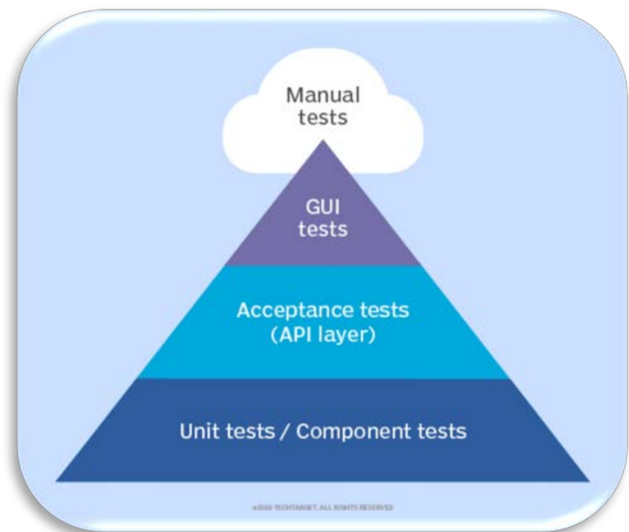


Figure 4. The agile test automation pyramid by Mike Cohn [1]

In Figure 4, four different kinds of test 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 works;

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

4) Manual tests - tests performed by a person which 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 many low-cost tests and a small number of high-cost tests.

By implementing TDD, programmers have the ability to identify potential problems early on and validate the veracity of the domain models. In addition, the iterative nature of TDD enables frequent feedback, which facilitates continuous refinement and adaptability in cloud service development [31].

The techniques laid out in this article are not suited to all situations and therefore have some limitations. They set constraints that provide long-term benefits, such as higher standards of craftsmanship. Time and effort are required to properly comprehend and implement the numerous DDD layers, patterns, and concepts, which can be overwhelming. The learning curve for DDD is steep, particularly for inexperienced coders [8]. It is important to emphasize that CQRS and most DDD patterns are not architectural styles but merely architectural patterns.

Microservices and service-oriented architecture (SOA) are examples of architectural styles, while CQRS and DDD paradigms characterize something contained within an individual unit of work [8]. At an architectural level, the design of each element in system shows its own trade-offs and internal design decisions.

6. Conclusion

The domain driven design approaches have emerged as a valuable methodology for building cloud native service architectures. By focusing on the core business domain and encapsulating it in a well-defined, bounded contexts, they help to create modular, scalable, and maintainable systems. By combining mentioned approaches, organizations can build systems that are not only technically robust but also aligned with their business goals, requirements, and objectives. Ultimately, the adoption of domain driven design and cloud native architectures can help organizations innovate faster, reduce costs, deliver better value to their customers, and stay competitive in a rapidly changing digital landscape.

Modification of the domain model is facilitated by its cleanliness. The incapability to maintain an adequate separation of concerns in enterprise grade software is the primary cause of overwhelmed code bases, leading to delays and even project failure. As this article focuses mostly on the relevant foundations, a case study on the domain driven software development process could be presented as a continuation.

Acknowledgements

This research is financially supported by NPD-331/2023 from University of Economics - Varna Science Fund.

References:

[1]. Ashbacher, C. (2010). Succeeding with agile: Software development using Scrum, by Mike Cohn. *The Journal of Object Technology*, *9*(4). Doi: 10.5381/jot.2010.9.4.r1

[2]. Avram, A. (2006). *Domain-Driven Design Quickly (first edit ed.). InfoQ*

[3]. Batista, F. (2019). *Developing the ubiquitous language*. The Domain Driven Design. Retrieved from: https://thedomaindrivendesign.io/developing-the-ubiquitous-language [accessed: 19 June 2023].

[4]. Bissi, W., Neto, A. T., & Emer, M. (2016). The effects of test-driven development on internal quality, external quality and productivity: A systematic review. *Information & Software Technology*, *74*, 45–54. Doi: 10.1016/j.infsof.2016.02.004.

[5]. Booch, G., Maksimchuk, R. A., Engle, M. W., Young, B. J., Conallen, J., & Houston, K. (2007). *Object-Oriented Analysis and Design with Applications*. Pearson Education.

[6]. Brewer, E. (2012). Pushing the cap: Strategies for consistency and availability. Computer, 45(2), 23-29.

[7]. Caron, R. (2018). *Get the Azure quick start guide for .NET developers*. Microsoft. Retrieved from: https://azure.microsoft.com/en-us/blog/get-the-azure-quick-start-guide-for-net-developers/ [accessed: 22 June 2023].

[8]. De La Torre, C., Wagner, B. & Rousos, M. (2023). *.NET microservices. architecture for containerized .NET applications*. Microsoft Learn. Retrieved from: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/ [accessed: 22 June 2023].

[9]. Debski, A., Szczepanik, B., Malawski, M., Spahr, S., & Muthig, D. (2018). A scalable & reactive architecture of a cloud application: CQRS and event sourcing case study. *IEEE Software*, *35*(2), 62–71. Doi: 10.1109/ms.2017.265095722

[10]. Erl, T. (2007). *SOA principles of service design*. Prentice Hall.

[11]. Evans, E. (2014). *Domain-driven design reference: Definitions and pattern summaries.* Dog Ear Publishing.

[12]. Evans, E. (2003). *Domain-driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional.

[13]. Fowler, M. (2012). *Pattern Enterprise Application Architecture*. Addison-Wesley.

[14]. Garverick, J., & McIver, O. (2023). *Implementing event-driven microservices architecture in .NET 7: Develop event-based distributed apps that can scale with ever-changing business demands using C# 11 and .NET 7*. Packt Publishing Ltd.

[15]. Hippchen, B., Giessler, P., Steinegger, R. H., Schneider, M., & Abeck, S. (2017). Designing microservice-based applications by using a domain-driven design approach. *International Journal on Advances in Software, 10*(3), 432–445.

[16]. Indrasiri, K., & Suhothayan, S. (2021). *Design patterns for cloud native applications: Patterns in practice using APIs, data, events, and streams*. O'Reilly Media.

[17]. Khononov, V. (2021). *Learning domain-driven design: aligning software architecture and business strategy*. O'Reilly Media.

[18]. Kumar, V., & Agnihotri, K. (2021). *Serverless computing using Azure Functions: Build, deploy, automate, and secure serverless application development with Azure Functions (English Edition)*. BPB Publications.

[19]. Martin, R.C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education.

[20]. Martin, R. C. (2017). *Clean Architecture: A craftsman's guide to software structure and design*. Prentice Hall.

[21]. Meyer, B. (1997). *Object-oriented software construction*. Prentice Hall.

[22]. Myers, B. (2022). *Red, green, refactor. What is test-driven development.* Medium. Retrieved from:. https://medium.com/codecastpublication/red-green-refactor-what-is-test-driven-development-302794e06c [accessed: 27 June 2023].

[23]. Oukes, P., Van Andel, M., Folmer, E., Bennett, R., & Lemmen, C. (2021). Domain-driven design applied to land administration system development: Lessons from the Netherlands. *Land Use Policy, 104*, 105379. Doi: 10.1016/j.landusepol.2021.105379

[24]. Palermo, J. (2013). *The onion architecture : Part 4 – After Four Years.* Jeffrey Palermo. Retrieved from: https://jeffreypalermo.com/2013/08/onion-architecture-part-4-after-four-years/ [accessed: 01 July 2023].

[25]. Petrov, P., Krumovich, S., Nikolov, N., Dimitrov, G., & Sulov, V. (2018). Web technologies used in the commercial banks in Finland. In *Proceedings of the 19th International Conference on Computer Systems and Technologies*, 94-98.

[26]. Petrov, P., Kuyumdzhiev, I., Malkawi, R., Dimitrov, G., & Bychkov, O. (2022). Database Administration Practical Aspects in Providing Digitalization of Educational Services. *International Journal of Emerging Technologies in Learning, 17*(20), 274-282. Doi: 10.3991/ijet.v17i20.32785

[27]. Rademacher, F., Sachweh, S., & Zündorf, A. (2017). Towards a UML profile for domain-driven design of microservice architectures. In Cerone, A., Roveri, M. (eds) *Software engineering and formal methods*: *SEFM 2017 collocated workshops: DataMod, FAACS, MSE, CoSim-CPS, and FOCLASA*, Trento, Italy, September 4-5, 2017, 230–245. Springer International Publishing. DOI: 10.1007/978-3-319-74781-1_17

[28]. Rademacher, F., Sorgalla, J., & Sachweh, S. (2018). Challenges of domain-driven microservice design: A model-driven perspective. *IEEE Software*, *35*(3), 36–43. Doi: 10.1109/ms.2018.2141028

[29]. Steinegger, R. H., Giessler, P., Hippchen, B., & Abeck, S. (2017). Overview of a Domain-driven design approach to build microservice-based Applications. In *Third International Conference on Advances and Trends in Software Engineering (SOFTENG 2017)*, 79–87.

[30]. Stoyanova, M., Vasilev, J., & Cristescu, M. (2021). Big data in property management. In *AIP Conference Proceedings, 2333*(1). AIP Publishing LLC.

[31]. Stuckenberg, S. (2014). *Exploring the organizational impact of software-as-a-service on software vendors. The role of organizational integration in software-as-a-service development and operation*. Peter Lang

[32]. Uludağ, Ö., Hauder, M., Kleehaus, M., Schimpfle, C., & Matthes, F. (2018). Supporting large-scale agile development with Domain-driven design. In Garbajosa, J., Wang, X., Aguiar, A. (eds) *Agile processes in software engineering and extreme programming 19th international conference*, 232–247. Springer. Doi: 10.1007/978-3-319-91602-6_16

[33]. Vasilev, J., & Stoyanova, M. (2019). Information sharing with upstream partners of supply chains. In *International multidisciplinary scientific geoconference: SGEM, 19*, 329-336.

[34]. Vernon, V. (2016). *Domain-driven design distilled*. Addison-Wesley Professional.

[35]. Vettor, R., & Smith, S. (2023). *Architecting cloud native .NET applications for Azure*. Microsoft Learn. Retrieved from: https://learn.microsoft.com/en-us/dotnet/architecture/cloud-native/ [accessed: 02 July 2023].

[36]. Wlaschin, S. (2018). *Domain modeling made functional: tackle software complexity with domain-driven design and F#*. Pragmatic Bookshelf.

[37]. Young, G. (2019). *Event centric: Finding simplicity in complex systems*. Addison-Wesley Professional.

[38]. Zimarev, A. (2019). *Hands-on domain-driven design with .NET Core: Tackling complexity in the heart of software by putting Domain-driven design principles into practice*. Packt Publishing.