# Exploring the Implementation of Domain-Driven Design in the Context of .NET and Azure: A Case Study

## Abstract

This article presents an empirical case study that examines the implementation of Domain-Driven Design (DDD) principles in a cloud-native environment, with the support of .NET and Azure. Building upon our previous research, which discussed the theoretical foundations of employing Domain-Driven Design (DDD) in cloud-native service architectures, the present study seeks to provide a pragmatic viewpoint. This study aims to provide a comprehensive understanding of the practical complexities, challenges, and advantages associated with the implementation of Domain-Driven Design (DDD) in a .NET and Azure environment, supported by Microsoft.

## Introduction

The preceding publication, titled "Domain-Driven Design Approaches in Cloud Native Service Architecture," established the conceptual foundation by examining fundamental aspects of DDD, the management of complexity through a stratified methodology, the implementation of command and query responsibility segregation (CQRS) and event sourcing, and the significance of test-driven development (TDD) in cloud-based services. Based on the aforementioned groundwork, the primary objective of this study is to conduct an empirical assessment of the aforementioned theoretical constructs through the utilization of Domain-Driven Design (DDD) within a .NET and Azure.

## Literature Review & Previews Work

Within the dynamic realm of cloud computing, there is a fundamental classifications, show on fig 1. Each category represents a discrete level of abstraction and service provision customized to meet unique operational requirements.



Fig 1. hierarchical distribution of responsibilities across the fundamental cloud service types

Among the presented, it is evident that Platform as a Service (PaaS) and, to some extent, Infrastructure as a Service (IaaS) have emerged as the primary areas of attention for DDD. PaaS and IaaS provide a framework that allows customers to create, build, and manage applications, therefore eliminating the challenges associated with the development and upkeep of the underlying infrastructure (or part of it). The IaaS has an innate capability to effectively handle a wide range of complex components, including networking, storage, servers, virtualization. In addition to these, PaaS also has the operating systems, middleware, and runtime environments. Therefore, the responsibility for developers is managing the applications and data components. The prominence of DDD concepts becomes evident inside this particular setting. The principles and principles of DDD are inherently aligned with the app and data components, enhancing the effectiveness of cloud-based solutions and strengthening the relationship between DDD and cloud paradigms.

In the year 2014, an article named "Microservices" was published by Martin Fowler and James Lewis, marking the beginning of one of the cloud native standard. Another core pillars of Cloud-native include containers, backing services, automation, and contemporary design. According to the Cloud Native Computing Foundation (CNCF), the official definition states that mentioned techniques „enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil." For instance, prominent companies such as Netflix (https://www.infoq.com/news/2013/06/netflix/) and Uber (https://www.uber.com/en-IT/blog/micro-deploy-code/) have developed goods that include a vast array of services, exceeding a count of 1,000 in their production. The deployment occurs on a frequent basis, with a frequency of several thousand instances each week.

Event Storming Approach

The primary objective of microservice architecture is to establish and delineate specific boundaries. One methodology that may be used is the event storming. It is a collaborative process used for design and scoping purposes in software development. The implementation of event storming and „ubiquitous language“ requires a comprehensive organizational framework, including the involvement of stackeholders, who have deep understanding of the business objectives. This approach is well-suited for teams who are using Lean and Agile methodologies, having hands-on approach to visually represent software solution (or part of it) in a understandable way.

Functional Programming:

In the scope of this work, we want to analyze and describe the practical aspects of the functional programming (FP). The paradigm is distinguished by two fundamental principles: integrity of method signatures and referential transparency. The concept of method signature honesty guarantees that a function's signature effectively represents all potential input and output values. On the other hand, referential transparency assures that the function's output stays constant for a specific input, without any accompanying side effects. In addition, FP contributes to the reduction of code complexity, facilitating improved comprehension and logical analysis of the code, streamlining the process of unit testing, and augmenting the modularity and composability of software components.

The relevance of immutability by FP is vital, considering the possibility for mutable operations to bring dishonesty into code. The absence of clarity affects our ability to fully engage in logical thinking, which sometimes complicating the process of debugging and perhaps posing obstacles to multi-threading. Moreover, the use of FB is enhancing by the set up of CQRS and the incorporation of the core domain logic. The notion of railway-oriented programming, which draws inspiration from Scott Wlaschin, presents a more efficient approach to organizing processes in contrast to standard methodologies characterized by lengthy and intricate code blocks including numerous "if" / "else" and "try" / "catch" lines. This functional approach uses extension methods to improve readability by minimizing the need for repetitive code and highlighting the primary logic flow.

## Methodology & Data Collection

The goal of this study is the exploration and in-depth understanding of the complex development of DDD, CQRS and ES via .NET and Azure. In order to reach this goal, the selection of an appropriate research approach is an important step. This section will present details with regard to research process, data collection and analysis procedure. The literature review and previews work have shown high uncertainty and a lag of research with regard to the implementation of the DDD concepts. The goal of this study is to fill this gap and show strong and reliable development processes. To approach this goal, case study research was deemed as an appropriate reseatch method. Case studies, representing qualitative research methods, are commonly used within the computer and social science. According to XXZ, the case study design may be chosen when the selected case represents a critical case in testing a well-formulated theory with clearly defined propsitions, which is going to be shown in the 3rd sub-section of this chapter. The nature of the current case study is confirmative (explanative). The purpose is testing the DDD theories that have been deducted from the previews literature and terminological foundations.

### 3.1. Tools & Technologies

The .NET is widely acknowledged as a prominent option for developing scalable and robust corporate applications. Based on statistics provided by Techempower [], it has been observed that ASP.NET exhibits superior efficiency and performance compared to several alternative web application platforms and full-stack frameworks. Microsoft has outlined a strategic plan [] for the future development and maintenance of .NET, guaranteeing regular upgrades and expanded library support until the year 2026. The framework of .NET is highly regarded due to its ability to seamlessly integrate with many programming languages, such as C#, F#, and VB, all of which have prominent positions on the Tiobe index []. According to research conducted by Statista [], C# has emerged as a prominent programming language used by developers for microservices. One of the factors contributing to this is the lightweight Minimal API [], which is a framework component specifically designed for microservices. Additional factors include the use supplemental libraries such as EntityFramework, MediatR, Optional, Marten, SignalR, AutoMapper, Serilog, Stylecop, Swagger, FluentValidation, xUnit, Autofixture, Moq and Shouldly. This interoperability further enhances the esteemed status of .NET. 

Microsoft Azure, a well-known provider of cloud services, offers extensive support for .NET applications via features like the Visual Studio Integrated Development Environment (IDE). This integration enhances the development experience and ensures interoperability within the broader Microsoft ecosystem. The figure 2 obtained from the "Flexera's 2023 State of the Cloud Report," [] showcases the use trends of different public cloud providers across enterprises.



Fig 2. cloud service providers used by organizations in the public sector. Source: Flexera 2023 State of the Cloud Report

The findings, derived from a sample of 750 participants, apparent that Azure has emerged as a prominent participant, as indicated by the fact that 41% of firms are using its platform to execute considerable workloads, namely 30%, are using it for certain tasks, and arround 13%, are now in the period of testing.

Based on the data obtained from Gather in 2023, Azure has shown a substantial growth rate of 47% in Cloud Infrastructure and Platform Services [], establishing its position as the leading public cloud platform. Azure has solidified its position as a dominant entity with its extensive network of more than 60 data centers. This surpasses the offerings of other cloud providers. Notably, Azure boasts major clients such as Samsung, Boeing, Ebay and BMW.

Based on the collected data, it can be deduced that the use of .NET and Azure is a favorable choice for performing a thorough analysis of the implementation of DDD within a particular technical stack.

### 3.2. Case Selection & Data Collection

Explaining the case study approach, detailing how data was collected and analyzed.

|  |  |  |
| --- | --- | --- |
| Case | System | Description |
| A | Order Managment |  |
| B | E-Commerse |  |
| C | Supply Chain Managment |  |

### 3.3. Theoretical Framework

Figure X illustrates a theoretical model of the DDD Approaches in the Cloud-Native Services Architecture.



Figure X. theoretical model of the DDD Approaches in the Cloud Enviorment.

Within the theoretical framework, the use DDD is underscored as a major paradigm, placing emphasis on domain-centric constructs such as Bounded Context and Ubiquitous Language, as supported by prior research on the subject matter. The use of CQRS pattern enhances this approach by advocating for the separation of read and write processes. Moreover, Event Sourcing offers a means to record changes in state, while TDD guarantees functional dependability. The study expands on these approaches in the context of a cloud-native environment. It employs a case study approach to provide empirical observations on the feasibility and effectiveness of implementing DDD methodologies.

## Implementation of DDD principles in .NET

Recapitulate key principles of DDD outlined in your previous work that are pertinent to the current case study.

### 4.1. Applying Bounded Contexts to Microservice Architecture

The concept of a bounded context, refers to a well defined area of responsibility that is delineated by a distinct border, which strongly aligns with the fundamental principles of microservice design. This may be seen as a preliminary stage in the development. Within a business domain, each bounded context serves as a container for a fundamental business idea, connecting functionality and data models.



Fig. UML Component Diagram of bounded contexts

Core ideas, such as 'order command api' or 'order receiver api, consist of subconcepts, such as .., that are associated with relevant data or functions.

Showen bounded context are characterized by CQRS oriented borders, which guarantees that each subconcept is either fully contained inside the context or completely outside of it.

This barrier functions similarly to the interface in microservices, providing protection for the internal data models.

The establishment of a protective border, in conjunction with the use of contract models, enables the maintenance of backward compatibility by allowing internal adjustments to be made without necessitating any changes to the exposed interface.

Bounded context, together with other DDD concept, provides a strategic framework for the identification and definition of the assemblies of microservices. Suggested:



### 4.2. CQRS in Practice

The architectural pattern known as Command and Query Responsibility Segregation (CQRS) addresses the separation of command operations, which involve writing data, from query operations, which involve reading data. This pattern is designed to enhance the performance and efficiency of distributed data architectures in systems that are built on microservices.

As seen in Figure 1, the implementation of CQRS involves the division of a conceptual microservice into two distinct physical microservices, with one dedicated to write operations and the other focused on read operations. It also enables the use of different API styles for the different microservices. This architectural approach enables the achievement of efficient and scalable data reporting on a wide scale, while maintaining optimal performance levels. An essential component of this architecture is the synchronization mechanism, which guarantees the maintenance of data consistency across the two distinct datastores. This process is facilitated by using the advantages of cloud infrastructure.

This differentiation of commands and queries is based on the "separation of concerns" principle. Commands, which modify the data state of an application, are task-based operations such as "register a new order item". These write operations preserve ACID transactions and reliable information consistency. In contrast, queries concentrate on intricate join operations and return results without modifying the data state of the application. They retrieve information from highly denormalized materialized views, thereby avoiding expensive repetitive table joins and table locking. Notably, queries always return data in standardized format using Data Transfer Objects (DTO).

The Mediator design pattern is used to organize the architecture in accordance with the principles of CQRS. It functions as the only means via which communication is facilitated between the user interface and the data repository. The notion entails the use of an intermediary entity, known as the 'mediator', to promote communication across various objects, hence encapsulating their interactions. This specific approach facilitates the reduction of direct interactions between objects, hence supporting the principles of "loose coupling" and "Inversion of Control“.

The use of Mediator pattern in conjunction with the CQRS is characterized by its simplicity. In the system, a unique input class is created for each command or query, which is mapped to the corresponding user action and allinged with the ubiquitous language. In addition, an individual message handler is created for each, shown in the following diagram.



The mediator message handlers demonstrate a diverse range of operating capabilities inside a single application.

The handlers demonstrate proficient management of validation, utilizing tools like Fluent Validation which allows the creation of validator handlers specific to message types, as exemplified by the IMessageValidator<T> interface. An intriguing aspect of this interface is its contravariant nature, enabling the adoption of a base type validator. In addition, they effectively mitigate risks pertaining to data redundancy via efficient duplicate management.

The primary objective of these algorithms is to effectively tackling the business complexity, by providing expertise in managing domain models like value objects, entities and event-sourced aggregates, which is will be examined in the subsequent section.

### 4.3. Ubiquitous Language via Functional Programming

The notion of Ubiquitous Language (UL) is a linguistic framework used in the practice of DDD to facilitate cohesive communication among team members in relation to high quality software code. It supports the process of defining and determining the dimensions of event handlers. The use of UL improves the process of building specialized software by describing it via core ideas and their associated subprocess. The successful execution necessitates a cooperative effort between software development teams and individuals with specialized knowledge in the relevant field. In an ideal situation, it is expected that all stakeholders possess a comprehensive understanding of the source code, enabling them to propose or endorse improvements, as well as detect possible issues or edge cases. Within the domain of C# and F# programming, the functional 'Either' monad arises as a sophisticated instrument for expressing complex business logic in a manner that corresponds to sequential operation descriptions. This approach allows for the representation of challenging scenarios in a pseudocode structure and promotes a smooth transition into executable code suitable for production.

In accordance with the specifications set out by UL, the generic structure of the Either type is established as follows:

**public** **struct** Either<T, TException>

{

**public** **bool** IsSuccessful { **get**; **private** **set**; }

**public** Either<TResult, TException> Map<TResult>(Func<T, TResult> mapping) =>

      Match(success: value => Either<TResult,TException>.Success(mapping(value)),

           error: exception =>

Either<TResult, TException>.Error(exception));

}

Essential part of the structure is the behavior of the map function. For a type container C<T> , Map behaves as follows: (C<T>, (T => T2)) => C<T2>.

When an Either object has a value representing a successful outcome, the map function applies a given transformation function to that value. Alternatively, in the event that an exception occurs, it will be returned as the value of a new "transformed" Either. An enhancemt is the FlatMap operation, which is related to the Map operation, designed to accept functions that return another Either, rather than accepting any general transformation function, e.g. (C<T>, (T => C<T2>)) => C<T2>.

Monads, in the realm of functional programming, refer to types that include a FlatMap function, among other features.

The functional approach, characterized by the Map function, offers a streamlined method of chaining operations, making the code more readable and maintainable.

The article particularly in scenarios involving order requests and results.

OrderRequest(...).Map(response =>

PublishEvent(... response.Data).Map(anotherResponse =>

StoreMetadataIntoTheCloud (... anotherResponse.Data).Map(thirdResponse =>

SendToExternalService (... thirdResponse.Data))));

The use of the 'Either' monad in the code facilitates the decomposition of intricate logic into separate, independent processes, aligning with the previously presented idea of UL. While imperative coding involves programmers working with complex code structures, the chained approach offers a simplified method for adding functionality and improving the ease of maintenance. This approach not only mitigates the likelihood of errors but also facilitates a smooth progression from pseudocode, which is first proposed by business analyst, to code that is suitable for production crafted by the software engineers. The use of the Mediator pattern in conjunction with the 'Either' monad yields code that exhibits qualities of readability, maintainability, and resilience.

### 4.4. Referencing the Event Sourcing

One notable aspect is the shift from the "Models to persist" approach in CQRS to the "Events to log" approach. This highlights the event-driven characteristic of CQRS, whereby data changes are not only stored in models but also logged as events.

The implementation of the Command Query Responsibility Segregation (CQRS) pattern has the potential to impact several facets of a system, including storage techniques, data flow

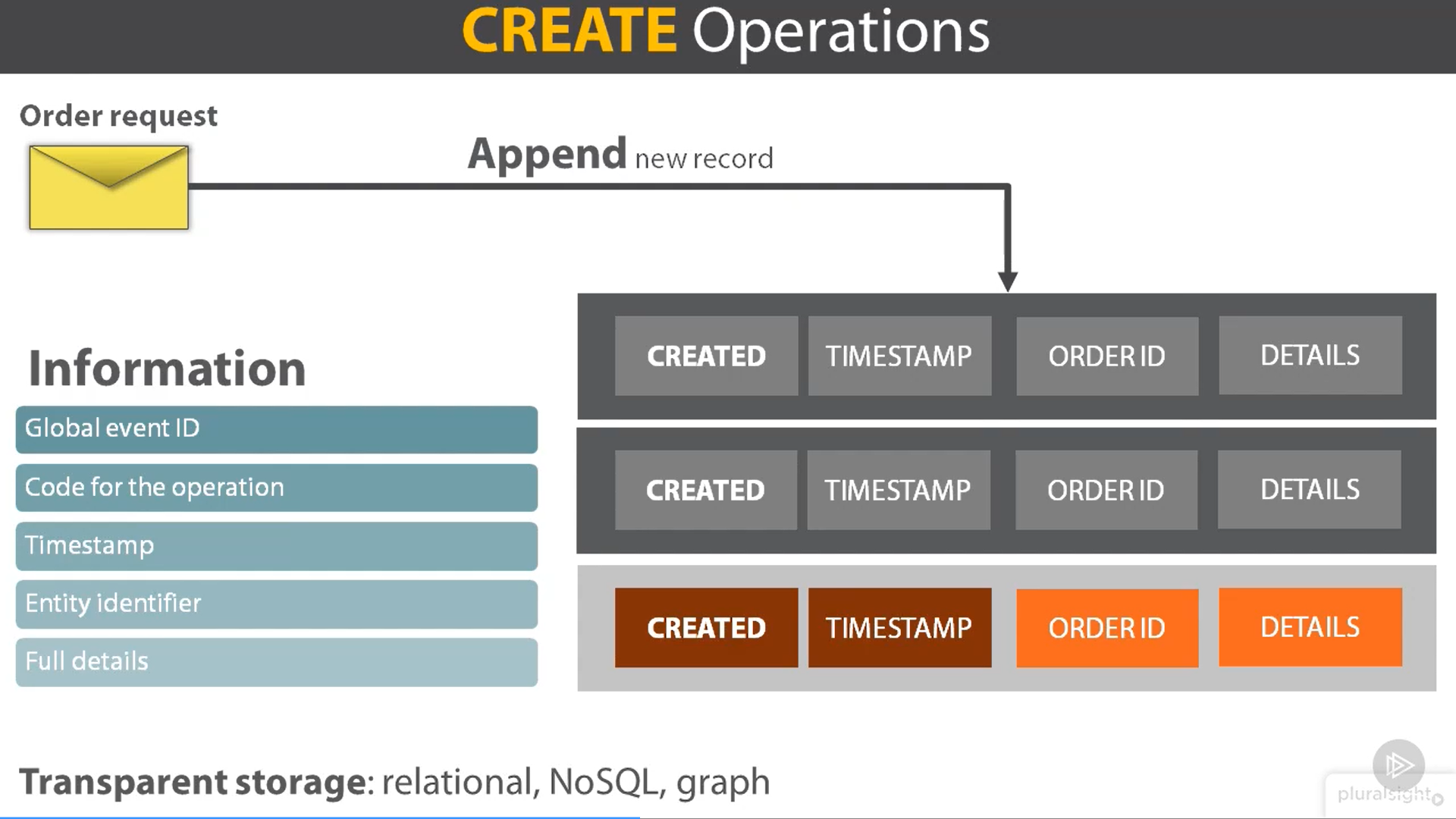
Event sourcing is a pattern that deviates from traditional data storage methods by encapsulating data as a series of events. This approach offers a systematic method for tracking data modifications, particularly in distributed database systems, by providing a comprehensive audit trail detailing when, by whom, and what specific data alterations were made.

The approach of event sourcing brings about a fundamental shift in the storage of individual entity data, whereby the state of a record is represented as a collection of events that have occurred.

Nevertheless, event sourcing has difficulties related to the efficiency of data retrieval. In order to address the problem, event sourcing incorporates the notion of "snapshots," which represents the aggrates from the DDD.

The use of event source is intrinsically aligned with event-driven architectures, facilitating the dissemination of targeted event notifications. This pattern serves to guarantee the integrity of data, facilitate traceability, and improve data exchange techniques inside distributed systems.

Events offer key architectural advantages due to their immutable nature, ensuring comprehensive tracking of all domain-related activities. The capability to replay events offers flexibility in processing and deriving various data projections. This discussion aims to provide architectural insight into the significance of events, their potential as a primary data source, and the implications of replaying events for data projection.



A key principle of CQRS-inspired software is that asking a question should not change the answer. When a new relevant entity is created and persisted in an event-based system, the existing data store is extended with a new item that contains all the information needed to be immutable. Each event must be uniquely identified, given an app-specific code, and stored in a transparent technology such as relational or document-based.

UPDATE operations store the same information as the creation operations, including a unique event ID, timestamp, code, and changes applied. The aggregate ID and delta are also stored. DELETE operations are logical and logical, stating that the entity with a given ID is no longer valid. Storage is transparent and can be any technology that works for the system.

Understanding queries is crucial for implementing persistence in software. The simplest way to do this is by physically deleting the last record as if it never happened, but it is important not to delete events in the middle of the stream to avoid inconsistent data.

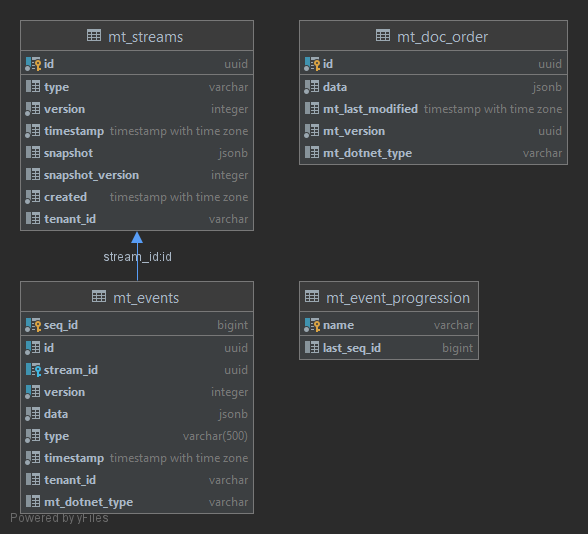
Example:



In many contemporary applications, data is primarily stored in databases by capturing the immediate state of an entity. For instance, when a user modifies their email address, the database table, specifically the email field, is updated to reflect this most recent change, thereby always ensuring that the latest data status is known. However, in expansive architectural frameworks, continuous database updates can detrimentally affect database performance, impede responsiveness, and pose challenges to scalability.

The Event Sourcing design pattern proposes an alternative approach. Rather than merely updating the current state, each action that influences the data is persisted in a specialized database termed the "Event Store." Each of these actions is designated as an "event." Contrary to traditional methods that save the latest data status, the Event Sourcing pattern suggests recording all events in a database in a sequentially ordered manner, thereby crafting a chronicle of data events.

This specialized database, the Event Store, refrains from overwriting existing data. Instead, every modification to the data is logged as a new record, culminating in a sequential compilation of historical events. Consequently, the Event Store becomes the authoritative source of data. This structured list of events is instrumental in generating Materialized Views, which represent the ultimate data state and facilitate query execution. Transitioning the Event Store to a readable database is achieved by adhering to the Materialized View Pattern. This transformation process can be orchestrated using the publish/subscribe model, where events are disseminated via message broker systems. The structured event list bestows the capability to replay events at specific timestamps, facilitating the reconstruction of the most recent data state by retracing these events.



Event-based data stores, such as Event Store, offer a structured approach to event storage, ensuring business consistency and respect for the event sourcing approach. These stores offer an API for plain HTTP and.NET, allowing for three basic operations: writing, reading, and subscription. The store's framework manages the timestamp and eventId, eventType, and data, preventing arbitrary deletion. There are three types of subscriptions: volatile, catch-up, and persistent. Volatile subscriptions invoke a callback function every time an event is written to a stream, while catch-up subscriptions receive notifications from a given event up to the current end of the stream. Persistent subscriptions ensure events are delivered to customers at least once, but potentially multiple times, with unpredictable order. This solution is designed for high-scalable and collaborative systems and requires a software design that supports hidden potency. Catch-up subscriptions are beneficial for components called denormalizers, which play a key role in a CQRS query stack.

Key tenets of event sourcing include:

1. Events signify occurrences in the past.
2. Events are articulated in the ubiquitous language, typically utilizing past-tense verbs. For instance, "OrderCreated" is an acceptable event nomenclature, whereas "PlaceOrder" is not.
3. Event storage necessitates specific mechanisms, ranging from tailored relational tables and NoSQL solutions to specialized event store products.
4. An event store functions as an append-only repository and does not accommodate deletions.
5. Events collectively represent a domain entity's state, which can be ascertained by replaying these events.
6. Occasionally, event replay may entail extensive data processing, necessitating the use of periodic snapshots to efficiently reconstruct the state.
7. Events, once stored, are immutable and can be replicated for scalability.
8. Event-associated behaviors are executed at the time of event notification, meaning event replay doesn't necessitate behavior repetition.
9. Utilizing events ensures comprehensive tracking, capturing every occurrence in real-time, irrespective of its consequent outcomes. This granular data preservation occurs at an abstraction level that is more rudimentary than current standards.

### 4.5. Integrated Test Siute

The primary objectives of unit testing are the establishment of isolation for the system under test (SUT) from its adjacent components, hence guaranteeing the examination of a singular functionality at a time. The objective of achieving this may be accomplished by the use of test doubles, specifically mocks, which serve the purpose of substituting dependencies with unpredictable behavior. Integration tests, conversely, ascertain the functionality of a system when it is integrated with external sub-systems. Although unit testing has the potential to provide insights into design choices, it does not always ensure the implementation of superior design ideas. One of the primary advantages of unit testing is the assurance it provides, enabling expedited modifications to code without compromising the integrity of pre-existing functionality. The attainment of this confidence is facilitated by seeing unit testing as a protective measure.

Coverage metrics are widely used in the field of software development; nevertheless, their effectiveness in correctly assessing the quality of unit tests is questionable. The aforementioned metrics have the potential to reveal patterns in the proportion of code base to test suite, although they do not provide a dependable safeguard for the tests per se. There are other metrics that may be used to assess coverage, such as code coverage, branch coverage, and assertion-free testing. Code coverage is a metric that quantifies the proportion of code lines performed by at least one test inside a test suite, relative to the total number of lines in the code base. The evaluation of branch coverage involves verifying the execution of each control structure, but, it does not serve as a dependable measure of the quality of the test suite. In contrast, assertion-free testing refers to the execution of tests without the inclusion of any assertion assertions, rendering them devoid of use and perhaps detrimental.

The reliability of coverage measures as indications for assessing the quality of a test suite is questionable. The user is unable to accurately trace the code paths of third-party libraries that are often used in their everyday tasks, including the integer type of the underlying .NET Framework. The presence of high coverage metrics may suggest inadequate testing practices, while it does not inherently imply superior quality. Striving for complete unit test coverage does not always ensure a high level of quality, since it involves substantial exertion and yields decreasing results. Instead, it is advisable to prioritize the testing of the most crucial components of the code base, such as the business logic, in order to achieve optimal quality of the test suite.

Unit tests play a vital role in verifying the accuracy and reliability of an application. Nevertheless, it is important to note that not all tests has inherent value, and it is crucial to consider the associated maintenance expenses for each individual test. The primary objective of pragmatic unit testing is to provide tests of superior quality that offer a dependable safety mechanism. A test of high value has a significant likelihood of detecting regression problems, a little likelihood of generating false positives, offers prompt feedback, and incurs few maintenance expenses. These qualities pertain to the extent of code coverage during test execution, the importance of the code, the number of relevant lines, the relevance of the code, the code used in the project, and external systems. Ensuring the separation of tests from the specific implementation details of the system under test, as well as promptly delivering feedback, are crucial elements in mitigating the occurrence of erroneous positive test results and upholding the integrity of the test suite.

The inclusion of valuable testing is important in order to guarantee the quality and dependability of an application. End-to-end tests provide the most effective safeguard against regressions and false positives; nevertheless, their execution speed is quite sluggish and they may not offer prompt response. Tests that are focused on minor functionality and cover just basic aspects of a system provide prompt feedback but possess a limited likelihood of detecting regression issues. Tests that replicate the precise implementation of the system being tested may also have significant value. Achieving a balance between these components is a challenge, since relying only on end-to-end tests may result in the need for extensive code restructuring and hinder the attainment of comprehensive confidence in the bulk of the code base. Achieving a satisfactory equilibrium is feasible, while sustaining an impeccable performance across all aspects is a formidable challenge.

The test pyramid is composed of three tiers: unit tests, integration tests, and end-to-end tests. The proportion of these tests within a test suite plays a critical role in establishing confidence in the operation of software and facilitating expedited modifications. Unit tests provide rapid feedback, but end-to-end tests need a longer execution time. The value proposition of each test differs, since higher levels provide more protection against regression mistakes and false positives, but at the expense of feedback time. A pragmatic approach to unit testing entails selecting and prioritizing the most critical components of the code base, with an emphasis on worthwhile tests that have a high likelihood of detecting mistakes and entail minimal maintenance expenses.

## Architectural Decisions in the Microsoft Azure Ecosystem

Detail the architectural choices made in the .NET and Azure environment.

### 5.1. Components and Messaging

Architecture Diagram

5.2. Monitoring

### 5.2. estimation of approximate expenses

exlel

## Discussion

After the presentation of the findings from the case study, the following chapter will consolidate the observations to draw a complete picture of the implementaion of DDD.

### 6.1. Interpretation of Results

Analyzing the results.

### 6.2. Implications and Recommendations

The study is among the first to develop an in-depth understanding of how the adoption of DDD influences the cloud-native Azure and .NET services. As outlined, previous studies have provided initial indications for potential implications. Motivated by this and the growing popularity of the modern software design, this study has set out to explore the DDD concepts and to embed them into Microsoft ecosystem. It also draws on extant theory and incorporates implications into a theoretically deduced framework.

### 6.3. Challenges and Limitations

Discussing the challenges encountered and what lessons can be drawn for future DDD implementations.

## Conclusion and Future Work

Summary of Findings

Recap the key findings of this empirical study.

Future Research Avenues

Suggest topics or questions for future research, possibly as further extensions of your own work.

## TODO