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

## Abstract

This study provides an in-depth analysis of the incorporation of Domain-Driven Design (DDD) into the cloud computing elements of the Microsoft ecosystem. The aim is to develop a software architecture that is prepared for scaling, maintaining, and minimizing costs. Expanding upon prior research that examined the theoretical underpinnings of using DDD in cloud-native services, this article aims to provide a practical overview via the use of a case study methodology. The objective of this research is to provide a comprehensive understanding of the practical challenges and advantages associated with the topic.

## Introduction

Within the constantly evolving world of software development, Domain-Driven Design (DDD) became known as a fundamental paradigm for building of sophisticated applications. By intimately connecting the design of the software with its primary business domain, DDD enables the creation of a strong and collaborative environment where technical and domain specialists work together to develop software that is resilient and adaptive to changing business needs.

The objective of this study is to furnish a holistic view of the strategic choices, architectural factors, and subsequent results of this integration. DDD offers a philosophy and set of guidelines such as Bounded Contexts, Command Query Responsibility Segregation (CQRS), and ubiquitous language, specifically in relation to microservices, functional programming, and event sourcing. An integrated test suite's role in ensuring system integrity is also examined. Additionally, the paper is focusing on components, messaging, monitoring, and the financial implications.

## 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.

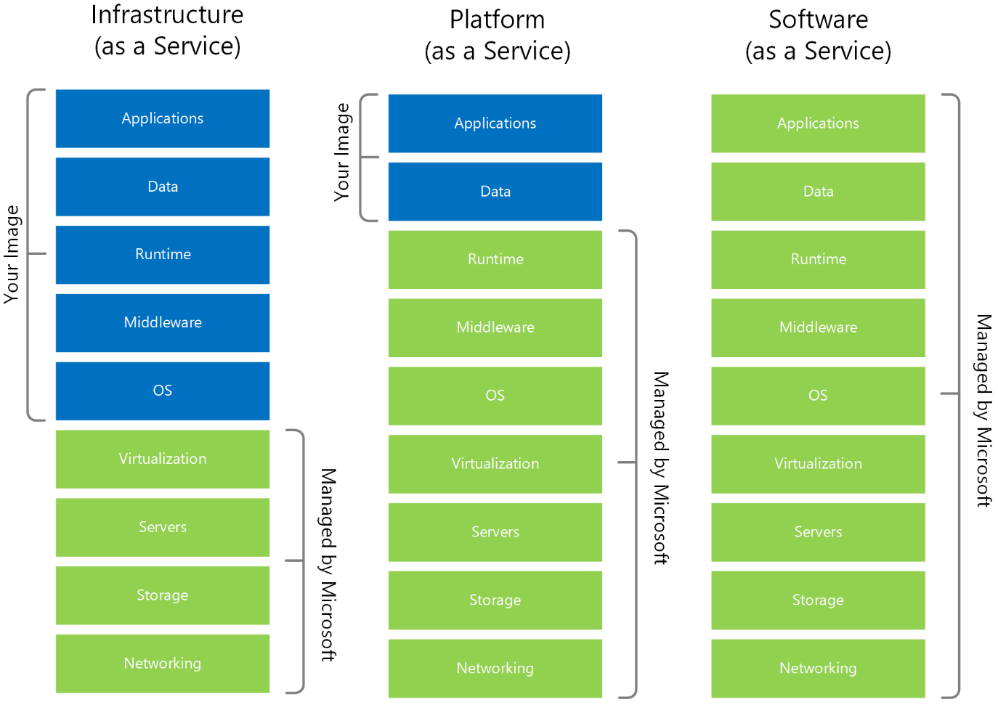


Fig 1. hierarchical distribution of responsibilities across the fundamental cloud service types

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 these particular fields.

In the year 2014, an article named "Microservices" was published by Martin Fowler and James Lewis, marking the beginning of one of a cloud native standards. The microservices architecture is characterized by the decomposition of applications into tiny, independent services. Each microservice, which encapsulates a particular business function, can be independently deployed, scaled, and maintained, thereby capitalizing on the inherent elasticity and resilience of cloud platforms. Microservices promoting continuous integration, continuous delivery, and dynamic resource allocation. According to the Cloud Native Computing Foundation (CNCF), the official definition states that microservices „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 online products 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.

The main goal of microservice architecture is to create clear and distinct boundaries. A methodology that may be used for this objective is the event storming, a collaborative technique utilized for the purpose of designing and delineating tasks. The implementation of event storming 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.

The advantages derived from this method include a comprehensive understanding of the business domain, by identification of the bounded contexts and associated aggregates, as well as the determination of the kinds of commands and queries that the end user executes on the system.

Aggregates are discerned through deep analysis sessions, which usually result in identifying various entities and value types, which sometimes naturally group together under the governance of a primary entity. When such a grouping occurs, it indicates the delineation of an aggregate, shaped solely by business rules. An aggregate serves as a domain model by encapsulating multiple entities under a singular conceptual umbrella. The need for transactional coherence necessitates the establishment of a wider border. The concept of a "Bounded Context" (BC) is a cornerstone in DDD, serving as a mechanism for isolating distinct parts to facilitate manageability and scalability. Furthermore, a BC underscores the need for self-sufficiency by encapsolating entities, repositories, factories, and application Services. BCs are representing distinct parts of the solution architecture tailored to address specific sub-domains that are primarily logical segregated. The level of physical separation adds another layer of complexity, depending on various factors like specific requirements, codebase and development team size.

Within the scope of this study, the intention is to examine and elucidate the pragmatic elements associated with the use of functional programming (FP) as a primary paradigm for constructing BC. The primary emphasis is on two distinct characteristics of FP: maintaining the integrity of method signatures and ensuring 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. The process of unit testing for such codebase basically consists of providing input to the functions and then checking the result.

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. 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. 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 proportion looks as follows: 

The reliability of coverage measures as indications for assessing the quality of a test suite is questionable. 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 in order to achieve optimal quality of the test suite.

## 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

The process of case selection and data collecting plays an integral part in establishing the empirical basis of this research. Motivated by qualitative, multiple case oriented study, largely using semi-structured interviews with software architects, developers, and business analysts. The emphasis on the functionalities related to order administration acts as a framework through which the more complex aspects of enterprise-level systems are streamlined. Below, we have given three relevant demonstrations for systems.

|  |  |  |
| --- | --- | --- |
| Case | System | Description |
| A | Order Managment | A digital system that manages the lifecycle of an order. It provides a centralized line for all sales channels, ensuring accurate pick, packing and shipping, as well as visibility into inventory and customer behavior. |
| B | E-Commerse | A technological framework that facilitates the buying and selling of goods and services over the Internet. These systems revolutionize the global economy by providing 24/7 shopping and a broader marketplace. |
| C | Supply Chain Managment | Software platforms for real-time visibility, facilitating collaboration among stakeholders, and synchronizing fragmented segments, ensuring efficient flow of goods, information, and finances. |

The process of collecting data is in accordance with the functional and non-functional needs that were obtained via the investigation of interview transcripts. The primary emphasis of this case study is the rigorous analysis of the implementation procedures pertaining to the registration of order records, as well as the subsequent alterations carried out by end users.

### 3.3. Theoretical Framework

Within the theoretical framework, presented on the following figure, 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.



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

## Implementation of DDD principles in .NET

### 4.1. Applying Bounded Contexts to Microservice Architecture

The concept of a BC, 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. Within a business domain, BC serves as a container for a fundamental business ideas, connecting functionality and data models. As seen in Figure 1, the design of the system is characterized by the presence of three primary microservices, namely the Receiver API, Command API, and Query API.



Fig. UML Component Diagram illustrates the structure and relationships of microservices within their respective BC

These microservices encapsulate separate and different duties within the order management BC. The Internet of Things (IoT) devices are integrated with the Receiver API, guaranteeing the effective management and queuing of incoming requests for further processing. The Command API is responsible for coordinating the persistence of order data and ensuring consistent interactions with the writing database. On the other hand, the Query API enables the retrieval of order information by directly integrating with the read database. These two APIs provides service to user interface (UI) clients. The practice of segregation cultivates a system architecture that is modular and easy to maintain, hence enhancing its resilience against the inherent intricacies involved in order management operations. The units of work demonstrate clear boundaries that are in line with the CQRS.

DDD is a framework that has an architectural structure characterised by the „layered approach“. As discussed in the previous article, the application layer, domain model layer, and infrastructure layer are integral components of the DDD. These layers play a crucial role in achieving the separation of concerns and effectively controlling the complexity of code. The proposed division of .NET assemblies is shown in the following diagram, as indicated by the researched statements.



Fig. DDD organized project structure

The graphic illustrates the architecture of DDD, which showcases a systematic arrangement of different components that contribute to the functional coherence of an order management system. The 'Orders Api' (which represents 'Orders Command Api', 'Orders Query Api' or 'Orders Receiver Api') is situated at the highest level of the hierarchy and plays a crucial. It acts as a central conduit, coordinating the interactions between the 'Business', 'Core', and 'Persistence' layers. The assembly known as 'Core' serves as a central hub for commands, queries and validation models. This creates a strong foundation for operational logic and data manipulation. Simultaneously, the 'Business' assembly contains the command and query handlers, along with interfaces to the ‚EventBus‘. This facilitates a smooth exchange of messages and guarantees accurate execution of CQRS. Upon further examination of the structural framework, it becomes evident that the 'Domain' assembly serves as a repository for aggregates, entities, events, and Data Transfer Objects (DTO). The ‚Persistence' assembly plays a role in the overall architectural framework by housing repository classes. These classes serve as a reliable and secure storage for data, ensuring its organized retention and enabling its efficient retrieval and utilization of a cloud database. The test project, as isolated from „source“, has a comprehensive suite of integration tests that have been developed using the Test-Driven Development (TDD) approach.

### 4.2. CQRS in Practice

As seen in Figure 1, 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. 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 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 DTOs.

The Mediator design pattern is used to organize the architecture in accordance with the principles of CQRS. It functions as communication proxy 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, while 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, as 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 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 =>

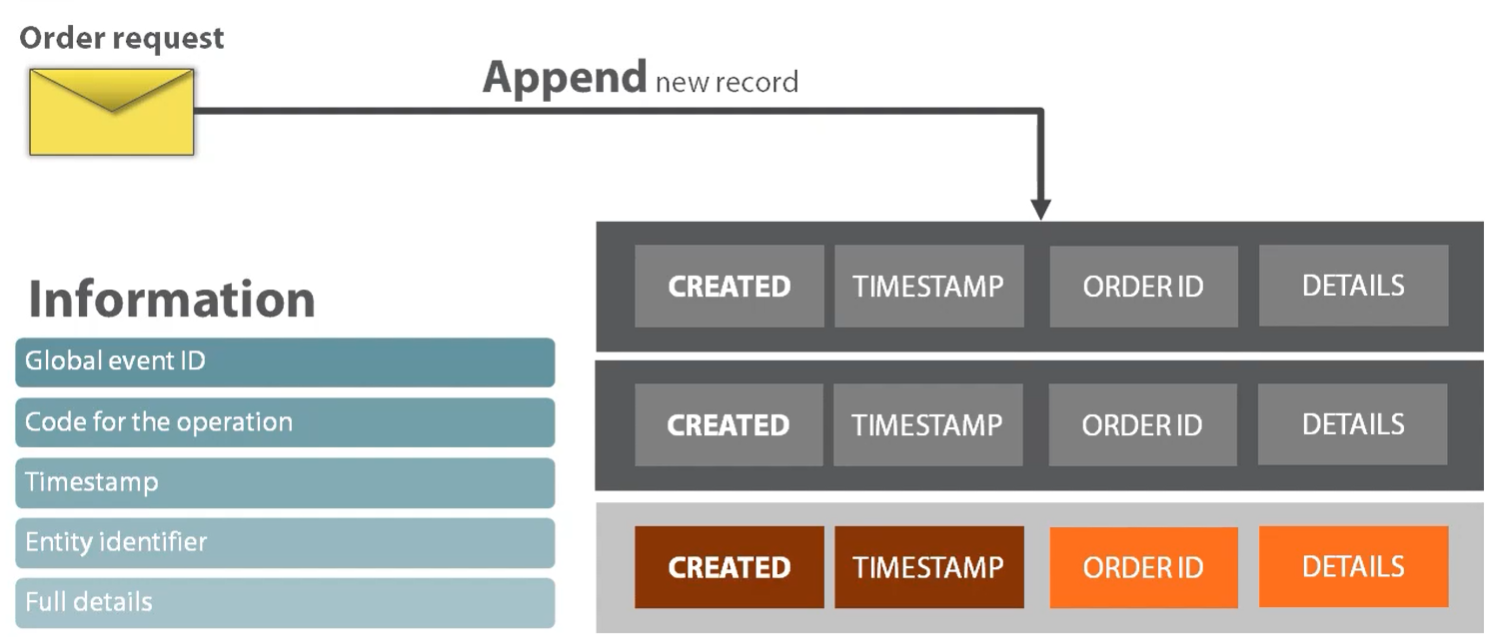
SendToExternalService (... thirdResponse.Data))));

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

As previously discussed, the adoption of the CQRS has the ability to influence several aspects, such as storage techniques and data distribution. A significant element is the transition from the "Models to persist" to the "Events to log". This feature emphasizes the event-driven nature of DDD and CQRS, in which changes to data are not only recorded in models but also documented as aggregatable events. Event sourcing (ES) is a pattern that deviates from traditional data storage methods by encapsulating data as a series of events. It offers a systematic approach for tracking data modifications, particularly in distributed systems, by providing a comprehensive audit trail detailing when, by whom, and what specific data alterations were made.However, ES has difficulties related to the efficiency of data retrieval. In order to address the problem, ES incorporates the notion of "snapshots," which represents the aggrates from the DDD. Moreover,the use of ES 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 of all domain-related activities, and improve data exchange techniques inside distributed systems, due to its immutable nature. The capability to replay events offers flexibility in processing and deriving various data projections that have the potential to be a primary source.

A key principle of CQRS 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. Each event must be uniquely identified and fully immutable. In the context of ES, the paradigm of 'Register Operations' may be described as the procedure by which order requests are transformed into added entries inside a storage system. As seen in Figure X, when an order request is received, a collection of relevant data is organized. This includes a unique global event ID, an assigned code for the operation, an accurate date, an entity identity, and a full set of order details.



event sourcing model of cloud-based order management, order requests are systematically transformed into immutable records

The event store database is a specific storage system that is based on the concepts of event sourcing. The integral nature of this component lies in its purpose of continuously storing events that signify changes in the state of a system, rather than storing the state itself. The primary purpose of this database is to serve as a repository where new data can only be added and existing data cannot be modified. This design feature guarantees that once events are recorded, they cannot be changed, hence maintaining the accuracy and chronological order of the historical record. Another feature is enabling the reconstruction of system state from any given point in time. Furthermore, with the use of event store databases, companies have the potential to acquire detailed understandings of system behaviors and patterns, facilitating the adoption of domain-driven decision-making processes and extensive auditing functionalities.

Schema of the suggested data store encompasses two primary database tables: streams and events. Streams serving as a foundation for organizing and categorizing events. They provide a comprehensive history of an aggregate, enabling state reconstruction, concurrency control, scalability, and interoperability. The following table provides a description of the recommended persistent model.

|  |  |
| --- | --- |
| Field | Description |
| id | A universally unique identifier that likely represents the primary key for each stream. |
| type | Specifies the type of the stream, which could be a category or classification. |
| version | Denotes the version number of the stream. |
| timestamp | Capture the exact moment when the record was either created or last updated. |
| snapshot | Represents a state capture of the stream at a certain version, enabling faster data retrieval. |

Events are fundamental units in event sourcing, capturing state changes and actions within a system. They provide historical immutability, auditability, temporal insights, decoupling, compensation and error handling. Events are not passive records but active, ensuring consistency, accountability, and adaptability. They enable granular data analysis, enabling businesses to derive deep insights and make informed decisions. The table shown below illustrates the proposed structure.

|  |  |
| --- | --- |
| Field | Description |
| id | Unique identifier for each event. |
| stream\_id | Connects events to their corresponding stream, establishing a relationship with the streams table. |
| seq\_id | A sequential identifier, potentially representing the order in which events occur. |
| type | Specifies the type of the event. |
| timestamp | Specifies when the event was recorded. |
| data | Capture the data payload of each event. |

### 4.5. Integrated Test Siute

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 have inherent value, and it is crucial to consider the associated maintenance expenses for each individual test. As described in the previews article, 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. Based on the observations, Figure 1.2 is used to elucidate the intricate balance between three distinct testing paradigms, namely end-to-end tests, trivial tests, and tests that mostly prioritize implementation specifics.



The fundamental idea of the image is on the notion that an optimal testing environment should inherently include all advantages.

The primary objective of this program is to advance the field of software testing by the creation and refinement of a specialized testing technique known as 'subcutaneous tests'. This kind of testing is specifically intended to assess the comprehensive behavior of a system, specifically targeting the layers immediately under the user interface. Ensuring the appropriate emulation of 'real-world' application situations in these tests is of the highest priority. This requires the creation of a structured transactional procedure that encompasses setup, execution, and verification stages. The repeatability of this technique is an important factor since it guarantees the constant and dependable execution of tests over multiple runs.

The baseline for tests is the so called “app fixture”. By using libraries like xUnit, the application fixture effectively sets the starting state of the program by applying the same startup configuration that is used in the live application. Ensuring a tight alignment between the test environment and the system's runtime configuration is crucial in order to avoid differences that might lead to false positive results. In the process of managing order data, users often follow a certain sequence of actions. Hence, it is essential for the test suite to replicate this procedural sequence. The AppFixture is tasked with the creation of a client scope, enabling the sending of a request and subsequent verification of the resulting outcome, connecting to the mediator handlers. For example, the order command API tests share a dependency on the AppFixture and customized auto data helpers, which will enable the suite to include many methods that provide straightforward instructions for the arrange, act, and assert stages.

## Architectural Decisions in the Microsoft Azure Ecosystem

The Azure cloud platform offers a comprehensive range of over 200 products that are specifically intended to facilitate the development and implementation of creative ideas. Utilizing pre-existing cloud services is sometimes seen as a more advantageous strategy in contrast to constructing services from scratch, due to the multitude of advantages it offers. Fully managed cloud platforms streamline operations by necessitating just the set up of the Azure resource and the deployment of code. However, these advantages are accompanied by essential costs that must be justified in terms of the IT infrastructure and the improvement of operational efficiency.

### 5.1. Components and Messaging

The order management system, which utilizes the Azure cloud-based platform, has undergone enhancements in section 2. These improvements include the implementation of a high-level architecture that has been purposefully developed to enhance security measures, optimize operational efficiency, and ensure the integrity of data. As seen in Figure 2, the entrance of the system is regulated by a Web Application Gateway, Firewall, and Load Balancers. These components assist the distribution of incoming requests to the respective API endpoints.

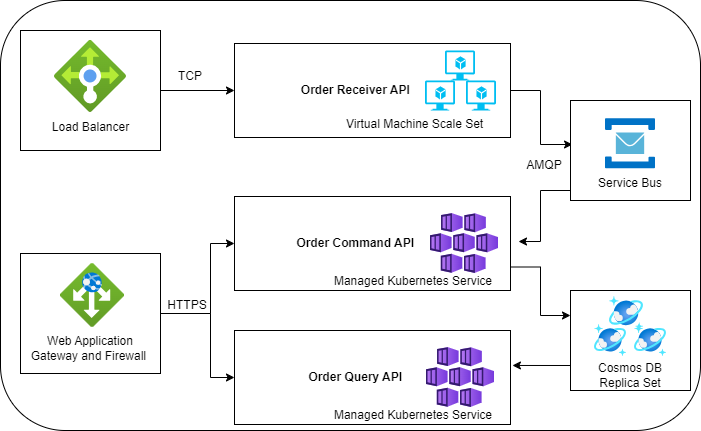


Fig. Archeture diagram of enhanced order managment components

The 'Order Received API' is running on a Virtual Machine Scale Set and relies on TCP communication to interact with IoT devices. The current API establishes asynchronous communication with the 'Order Command API' via the use of the Advanced Message Queuing Protocol (AMQP) and Azure Service Bus. This implementation guarantees the reliable delivery of messages. Simultaneously, the 'Order Command API' and 'Order Query API' are dealing with complexities inside the fundamental units of the software, in order to provide responses to the user interface clients. The Managed Kubernetes Service incorporates both of these basic APIs, maximizing the advantages of containerization. The data persistence layer of the system is supported by the Cosmos DB Replica Set, which guarantees the implementation of event sourcing principles via real-time distribution, redundancy, and consistency.

### 5.2. Monitoring

Monitoring and analytics are crucial components in the domain of cloud-based order management systems, as they are vital in guaranteeing the resilience and optimization of system performance. As seen in Figure 3, Azure Monitor plays a crucial role as a fundamental component inside this ecosystem.



architectural overview of "Azure Monitor"

The system integrates information from several sources. The data points are mostly seen in the form of metrics and logs, supporting the event based approach. The data that has been absorbed undergoes a series of complex processing stages. Insights refer to several components inside the Azure platform, including apps, containers, virtual machines, and specialized monitoring solutions. Visualization technologies like as dashboards, views, Power BI, and workbooks are used to improve user engagement and facilitate the comprehension of data. The analytical capabilities may be categorized into two main types: Metric Analytics and Log Analytics. These two types enable developers to explore detailed data points and historical logs, respectively. In addition, Azure Monitor incorporates warning systems and autoscaling capabilities to promote proactive system management, enabling prompt reactions to abnormalities or limitations in resources. The process of integration is made more efficient with the use of Logic Apps and Export APIs, facilitating a smooth interaction between Azure Monitor and other systems for example project management tools.

### 5.2. estimation of approximate expenses

Azure price calculator[] is a very useful online tool that aids in converting projected cloud use into approximate cost estimates, hence optimizing the process of budget planning for expenses linked to the services. This tool plays a role in making educated choices on the cloud finance strategy. The calculator has the ability to provide a cost estimate that accurately reflects the Azure use, while also considering any negotiated or discounted pricing. The following table provides a complete analysis of estimated expenses derived from the proposed architecture encompassing computing resources, load balancing, application gateways, monitoring mechanisms, databases, service buses, and container management systems, indicates a total monthly expense of $1,999.40.

|  |  |  |
| --- | --- | --- |
| **Service type** | **Description** | **Estimated monthly cost** |
| Virtual Machine Scale Sets | 2 D4s v4 (4 vCPUs, 16 GB RAM) (3 year reserved), Linux, (Pay as you go) | $127.60 |
| Load Balancer | Standard Tier: 5 Rules, 1,000 GB Data Processed | $23.25 |
| Application Gateway | Web Application Firewall V2 tier, 730 Fixed gateway Hours, 5 GB Data transfer | $352.15 |
| Azure Monitor | Log analytics: 0.25 GB Daily logs ingested, 24 months Data retention; Application Insights: 0 GB Daily logs ingested, 3 months Data retention, 0 Multi-step Web Tests; 1 VM monitored X 1 metric monitored per VM, 1 Log Alert at 5 Minutes Frequency, 0 Additional events, 0 Additional emails, 0 Additional push notifications, 0 Additional web hooks (in millions) | $46.45 |
| Azure Cosmos DB | Standard provisioned throughput (manual), Single Region Write (Single-Master) - West Europe (Write Region); 400 RU/s x 730 Hours; 4,000 GB transactional storage, 2 copies of periodic backup storage; Dedicated Gateway not enabled | $1,023.36 |
| Service Bus | Standard tier: 7 Throughput unit(s) x 730 Hours, 10 million Ingress events | $153.58 |
| Kubernetes Service | Standard Tier; 1 S1 (1 Core(s), 1.75 GB RAM, 50 GB Storage) x 730 Hours; Linux OS | $273.00 |
|  | **Licensing Program** | **Microsoft Online Services Agreement** |
|  | **Total** | **$1,999.40** |

## 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. 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. Motivated by the growing popularity of the modern software design, this study has set out to explore the DDD concepts and embed them into Microsoft ecosystem. It also draws on extant theory and incorporates implications into a theoretically deduced framework. The findings point out a necessity of implementing the full set of technologies and patterns, to ensure the seamless functioning of system components and maximize the benefits. Previous research has presented preliminary findings that suggest potential implications, which are further substantiated in the present study and may assist IT professionals in implementing these outcomes in a production-type environment.

### 6.2. 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.