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

Authors

## Abstract

Text

## Introduction

Text text

## 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 for individual microservices. One methodology that may be used, while not originally designed for this purpose, is event storming. Despite its lack of specificity for constrained settings, event storming produces results that can be effectively aligned with them. Event storming is a collaborative process used for design and scoping purposes in software development. It involves identifying software needs via the use of aggregates, which may be likened to constrained contexts. These aggregates can then be mapped to microservices. In contrast to the widely used ubiquitous language method, the implementation of event storming requires a more comprehensive organizational framework. This framework includes the involvement of a dedicated facilitator, generally a product owner or manager, who has a deep understanding of both business objectives and the principles governing event storming. The technique is distinguished by its hands-on approach, use colorful adhesive notes to visually represent software solutions in a globally understandable way, avoiding intricate diagrams. The designs, often shown on large paper rolls for ease of movement, include the whole microservice architecture, including individual services and their intercommunication. This approach is especially well-suited for bigger teams who are using Agile methodologies. The next parts will provide a practical illustration of an event storming session, accompanied by concise cheat sheets for reference.

Todo:

## 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 BMW, ASOS, and HP.

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 |
|  |  |  |
|  |  |  |
|  |  |  |

### 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, using the capabilities of Azure. It employs a case study approach to provide empirical observations on the feasibility and effectiveness of implementing these 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.

Core ideas, such as 'product' or 'inventory', consist of subconcepts that are associated with relevant data or functions. The bounded context is characterized by its clear exterior border, 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. In conclusion, the concept of bounded context provides a strategic framework for the identification and definition of the scope of microservices.

### 4.2. CQRS in Practice

Text

### 4.3. Ubiquitous Language via Functional Programming

Asdasda

### 4.4. Referencing the Event Sourcing

Sasa

### 4.5. Integrated Test-Driven Siute

Discuss the role of TDD in the development cycle, and how it contributed to the robustness and reliability of the application.

## Architectural Decisions in the Microsoft Azure Ecosystem

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

## Discussion

Text

### 6.1. Interpretation of Results

Analyzing the results in relation to the research questions and theoretical framework.

### 6.2. Challenges and Limitations

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

### 6.3 Implications and Recommendations

Provide actionable insights for researchers and practitioners Discuss the broader implications of these findings for practitioners and academics.

## 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