Domain-Driven Design in Cloud Computing: .NET and Azure Case Analysis

Jordan Jordanov [1], Pavel Petrov [1], Ivan Kuyumdzhiev [1], Julian Vasilev [1], Stefka Petrova [1]

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

***Abstract* – The study explores the integration of domain-driven design (DDD) with the cloud computing framework provided by the Microsoft Azure platform. Limited research exists that connects theoretical DDD principles with practical applications in cloud environments, and this research tries to focus on how DDD concepts could be effectively implemented in PaaS and IaaS cloud models. In this regard, the main research question is: How could DDD concepts be effectively applied on the Microsoft Azure platform using .NET services? The study hypothesizes that by applying the main components of the DDD, such as event-driven patterns, aggregates, and bounded contexts, one could significantly enhance the scalability, maintainability, and efficiency of the cloud applications. The research uses a case study approach as a main research method and evaluates the practical application of DDD within the context of Microsoft Azure's cloud models. The study finds that DDD offers significant advantages in structuring cloud-native applications, especially in the design of application and data layers. Key findings of the study suggest that DDD, when combined with Azure's cloud capabilities, can provide a robust framework for building scalable, resilient software systems, although some problems remain in aligning theoretical DDD with practical cloud development frameworks.**

DOI: 10.18421/TEM141-05
https://doi.org/10.18421/TEM141-05

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

Received: 07 August 2024.
Revised: 04 December 2024.
Accepted: 10 December 2024.
Published: 27 February 2025.

***Keywords* – Domain driven design, cloud computing, case study, software architecture, Azure .NET.**

1. Introduction

DDD has become an important framework in the constantly evolving field of software development, enabling the creation of advanced applications. DDD creates a collaborative environment by closely linking software design with the main business domain [1]. This approach encourages technical and domain experts to work together to develop software that is flexible and can easily adapt to evolving business requirements. Although this approach shows potential, there is still a notable lack of practical studies examining the relationship between DDD concepts and cloud development frameworks for constructing web, mobile, desktop, or Internet of Things (IoT) applications. This paper addresses the following research question:

How could DDD concepts be effectively applied on the Microsoft Azure platform using .NET services?

The study seeks to provide a thorough view of the strategic decisions, architectural components, and outcomes associated with these integrations. To do this, the study employs a research technique that includes a variety of use scenarios.

DDD offers a philosophy and set of guidelines, such as bounded contexts (BCs) and ubiquitous language [2], [3]. Also, there are programming models such as "aggregates" and "value objects," as well as patterns such as command query responsibility segregation (CQRS) and event sourcing (ES). These principles are suitable for microservices, functional programming (FP), and event-driven development. An integrated test suite also could be used to provide the integrity of all of them [4].

The microservices architecture is defined as the process of breaking down applications into small, autonomous services, hence establishing one of the cloud-native standards [6]. Each microservice, which encapsulates a specific business function, may be deployed, scaled, and managed individually.

This allows one to take advantage of cloud platforms' inherent flexibility and resilience. Continuous integration, continuous delivery, and dynamic resource allocation are made easier to implement using microservices. The Cloud Native Computing Foundation (CNCF) [7] defines microservices as system components that are loosely coupled, robust, managed, and observable. When used in conjunction with strong automation, they enable engineers to make significant and predictable changes, frequently with minimal effort. There have been numerous studies of the world's leading corporations, such as Netflix and Uber [8]. Netflix and Uber support online platforms that offer a wide range of services. New versions of the software responsible for these services are frequently released, with thousands of web applications being deployed on a daily basis.

The primary objective of microservice architecture is to establish explicit and well-defined boundaries. This process includes identifying BCs and associated aggregates and determining the types of commands and queries that end users perform on the system. BC is a fundamental concept in DDD that acts as a means of separating different components to enhance their ease of management and scalability. It emphasizes the importance of self-reliance by encompassing entities, repositories, factories, and application services [9]. BCs are components of the solution architecture designed to address specific, logically separated sub-domains. The degree of physical isolation introduces an additional level of complexity, depending on factors such as precise specifications, codebase, and the size of the development team.

At least one aggregate is present in BC. Aggregates are identified through thorough analysis sessions, typically leading to the recognition of different entities and value types that naturally form groups under the control of a main entity. When this kind of grouping happens, it signifies the demarcation of a collective, formed exclusively by business regulations. An aggregate function acts as a domain model by grouping multiple entities together under a single conceptual framework.

The present study investigates the practical implications of utilizing FP to provide an approach for creating aggregates and other DDD models. FP is primarily concerned with two unique features: The integrity of method signatures and referential transparency [10]. The idea of method signature honesty assures that a function's signature accurately and completely captures all potential input and output values. Referential transparency ensures that a function's output is consistent for every given input, with no additional side effects.

Furthermore, FP is intended to reduce code complexity, making it easier to comprehend and analyze rationally. It is also thought to simplify unit testing while increasing the modularity and composability of software components.

The immutability in FP is considered important, as mutable operations have the potential to introduce "dishonesty" into the code. The absence of clarity hampers the capacity of a software developer to participate in rational reasoning, making the process of debugging more complex and creating barriers to multi-threading programming. Furthermore, the implementation of CQRS and the integration of fundamental domain logic improve FP utilization. Railway-oriented programming, influenced by Scott Wlaschin, offers a more efficient method of structuring processes in contrast to conventional methodologies that involve lengthy and complex code blocks containing numerous "if/else" and "try/catch" statements [11]. The functional approach employs extension methods to improve legibility by reducing redundant code and emphasizing the main logical sequence.

In this context, it is important to analyze the logic of the code in real time by putting the system under test (SUT). Unit testing for codebases of this nature primarily entails supplying input to functions and verifying the outcomes [12]. Test doubles, particularly mocks, can support these needs by replacing dependencies with unpredictable behavior, thereby achieving the desired outcome. Unit testing offers the key benefit of ensuring the integrity of existing functionality while allowing for efficient modifications to code.

Based on a case study from the Department of Computer Science at NC State University [13], unit testing is considered a crucial safeguarding measure. Within this framework, a key performance indicator (KPI) is code coverage, also known as test coverage. This metric quantifies the extent to which the source code of a program is tested by a particular test suite. Code coverage is expressed as the ratio of the number of lines of code covered by tests to the overall number of lines in the codebase, represented as follows: Code coverage = lines of code covered / overall number of lines.

This ratio provides a numerical figure that reflects the level of testing and aids in the identification of untested code segments. High code coverage is associated with improved software quality because it indicates that a large part of the code was executed during testing, potentially revealing flaws and guaranteeing that the software performs as intended under varied scenarios. However, even 100% code coverage does not guarantee the absence of problems because it does not consider the quality or thoroughness of the tests themselves.

Nonetheless, striving for increased code coverage can help to produce more robust and maintainable code by promoting thorough testing techniques.

As an illustration of useful advantages, utilizing cloud technologies allows Progressive Web Apps to scale seamlessly to handle large volumes of traffic and ensure optimal performance. Also, cloud providers offer advanced security features and compliance certifications, ensuring that applications are protected against threats and adhere to industry standards and regulations.

2. Methodology

The aim of this study is to explore and provide an overview of software development with DDD, CQRS, and ES patterns via Microsoft .NET and Azure technologies. There is currently an uncertainty and a gap in research regarding the implementation of DDD concepts. The goal of this study is to fill this gap and demonstrate strong and reliable development processes. For this goal, case study research was deemed an appropriate research method. Case studies, representing qualitative research methods, are commonly used in computer and social science. Runeson *et al.* [14] suggest choosing the case study design when the selected case serves as a critical case for testing a well-formulated theory with clearly defined propositions, as demonstrated in Subsection 2.3. The nature of the current case study is confirmatory (explanatory). The purpose of the case study is to test the DDD theories that have been deduced from previous research [15].

2.1. Tools and Technologies

Table 1 shows the differences between the two main cloud service models: IaaS and PaaS.

Table 1. Classification across IaaS and PaaS cloud models

Layers	***IaaS and PaaS* Management**
Application	The IT department manages both IaaS and PaaS.
Data	The IT department manages both IaaS and PaaS.
Runtime	The IT department manages IaaS, the cloud provider handles PaaS.
Middleware	The IT department manages IaaS, the cloud provider handles PaaS.
OS	The IT department manages IaaS, the cloud provider handles PaaS.
Virtualization	The cloud provider manages both IaaS and PaaS.

Within the IaaS model, the cloud provider assumes responsibility for managing fundamental resources such as networking, storage, servers, and virtualization. On the other hand, the user is accountable for handling the operating system, middleware, runtime, data, and applications. In contrast, the PaaS model expands the provider's obligations to encompass the operating system, middleware, and runtime. This relieves the software engineers from the burden of managing these tasks and enables them to concentrate exclusively on their data and applications [5].

Among the above-presented models, PaaS and, to some extent, IaaS have emerged as key areas of focus for DDD. PaaS and IaaS offer customers the tools and systems needed to create, construct, and deploy applications. The importance of DDD concepts is evident in this context, particularly regarding the "data" and "applications" layers.

.NET is widely acknowledged as a key option for developing scalable and robust corporate applications. Based on statistics provided by Techempower (Round 22, October 2023) [16], it has been observed that ASP.NET demonstrates efficiency and performance compared to several alternative web application platforms and full-stack frameworks, as shown in Table 2.

Table 2. Comparison of server technologies

Technology	**Programming language**	**Processed requests per second**
Actix	Rust	~ 171 484
ASP .NET Core	C#	~ 144 304
Fiber	Go	~ 116 952
NodeJS	Javascript / C++	~ 33 868
Spring	Java	~ 24 082
Django	Python	~ 14 707
Laravel	PHP	~ 7 355

Based on the provided data, ASP.NET Core could be acknowledged to be faster than NodeJS, Fiber, Laravel, Django, and Spring. The significant performance advantage shows ASP.NET Core's efficiency and capability for handling high-performance web applications. Recently, Microsoft has outlined a strategic plan for the development and maintenance of .NET, guaranteeing regular upgrades and expanded library support [17]. Also, .NET was recently recognized in Stack Overflow surveys as the "#1 Most Loved Framework" for three consecutive years (2019, 2020, 2021) [18]. The .NET ecosystem is also active in the open-source movement, with its GitHub repository being ranked among the "Top 30 Highest Velocity OSS Projects".

GitHub data indicates that C#, the primary language in the .NET ecosystem, ranks among the top five programming languages [19]. This statistic highlights the growing interest in and adoption of the .NET framework in different academic fields. Additional factors include the use of supplementary libraries such as Minimal API, EntityFramework, MediatR, Optional, Marten, SignalR, AutoMapper, Serilog, Stylecop, Swagger, FluentValidation, xUnit, Autofixture, Moq, and Shouldly.

Microsoft Azure offers extensive support for .NET applications via integrated development environments (IDE) such as Visual Studio. This integration enhances the development experience and ensures interoperability within the broader Microsoft ecosystem. Figure 1, obtained from "Flexera's 2023 State of the Cloud Report" [20], showcases the usage trends of public cloud providers across different enterprises.

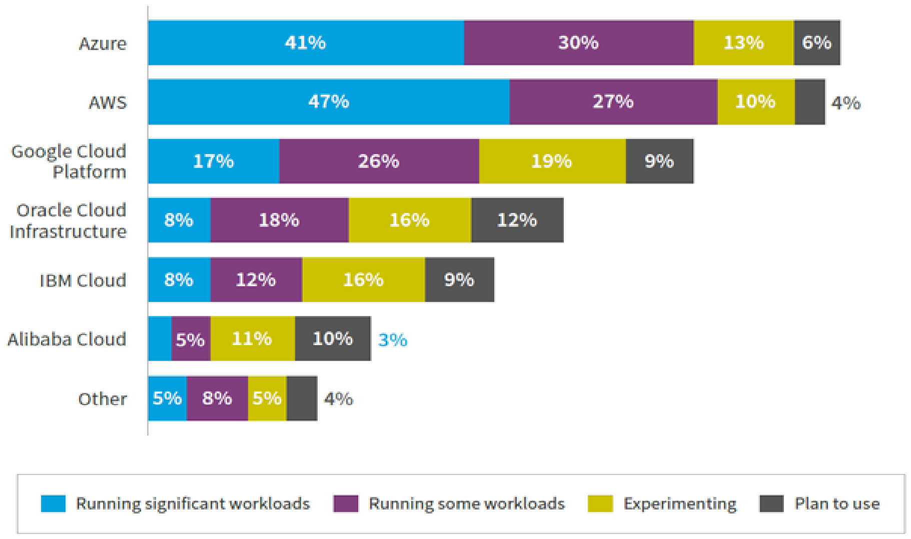

Figure 1. Cloud service providers used by organizations in the public sector in 2023 [20].

The findings derived from a sample of 750 participants indicate that Azure has emerged as a major player in the cloud services market. Around 41% of firms are utilizing its platform to execute substantial workloads, 30% are using it for certain tasks, and it is currently in the testing phase at approximately 13% of firms. According to data from Microsoft, Azure exhibited a substantial growth rate of 31% in the quarter ending March 2024. Azure's extensive network of over 60 data centres surpasses the offerings of other cloud providers. Many major clients, such as Samsung, Boeing, eBay, and BMW, rely on Azure's services. The collected data shows that using .NET and Azure is a good option for performing a thorough analysis of the implementation of DDD.

2.2. Case Selection

The process of case selection and data collection plays an integral role in the empirical foundation of this research. This study is motivated by multiple cases, specifically drawing on the Microsoft reference applications eShopOnContainers [21] and eShopOnAzure [22]. The emphasis on order administration functionalities serves as a framework for streamlining the more complex aspects of enterprise-level systems.

Three relevant demonstrations for these systems are presented in Table 3.

Table 3. Cases of enterprise-level systems

Case	System	Description
A	Order Management	A digital system that oversees the entire lifecycle of an order. It centralizes the management of all sales channels, ensuring precise picking, packing, and shipping processes.
B	E-commerce	An online platform that enables the exchange of products and services over the Internet. By doing this, e-commerce technology improves convenience for both consumers and enterprises.
C	Supply Chain Management	Software platforms for real-time visibility, ensuring the efficient flow of goods, information, and finances.

The process of data collection aligns with the functional and non-functional requirements identified through a review of existing research [23], [24] and guidelines [25]. This case study primarily provides an analysis of the implementation procedures related to the registration of order records, as well as the subsequent modifications made by end users and external devices.

2.3. Conceptual Framework

The conceptual framework illustrated in Figure 2 combines domain-centric design with several architectural patterns for cloud microservices design and development. BC, ubiquitous language, entities, value objects, and aggregates capture and articulate the complexities of the business domain.

CQRS is used to categorize the concerns, and ES is incorporated to maintain a reliable audit trail of changes. TDD drives the design of the system through the "tests-first" approach, and the case study methodology provides a practical validation of the framework.

Applications currently rarely fit neatly into a single paradigm; instead, they exhibit varying degrees of complexity. Consequently, attempting to apply a single modeling strategy across all applications is ineffective. Recognizing this, the case study methodology is viewed as a strategy, as it aligns with the research topic, namely the impact of DDD on cloud solutions. Case study research [26] is often regarded as a valuable method for facilitating the establishment of comprehensive knowledge of a particular phenomenon, aligning with the aims of the present study.

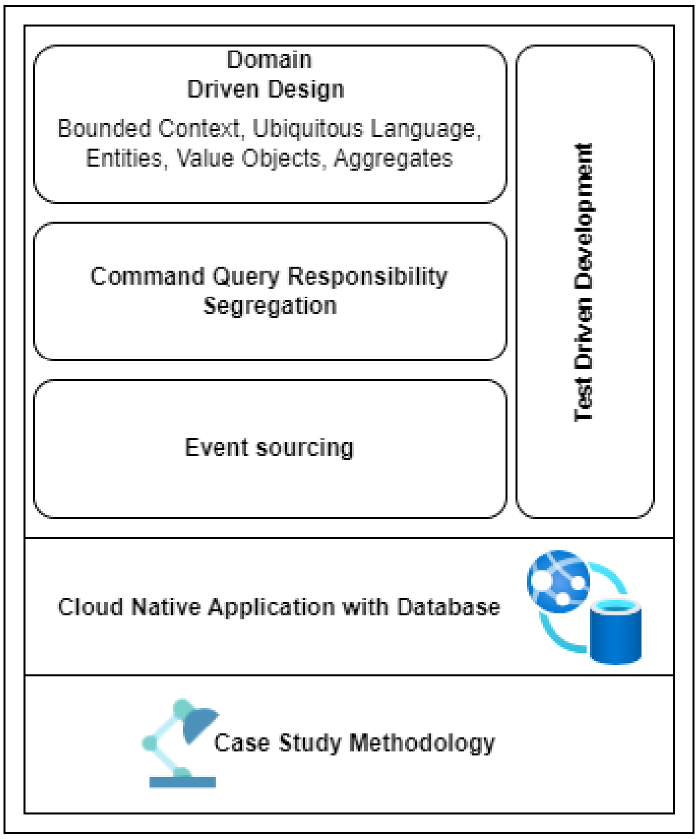
Domain
Driven Design
Bounded Context, Ubiquitous Language,
Entities, Value Objects, Aggregates
Command Query Responsibility
Segregation
Event sourcing
Test Driven Development
Cloud Native Application with Database
Case Study Methodology

Figure 2. Conceptual framework model of the DDD approaches in the cloud environment

3. Results

This section presents a combination of visual representations and data specifications of the system's architecture. These findings reinforce the adoption of DDD, CQRS, and ES within business management.

3.1. Applying BC and CQRS to Microservice Architecture

The concept of a BC, which refers to a well-defined area of responsibility delineated by a distinct border, strongly aligns with the fundamental principles of microservice design. Within a business domain, BC serves as a container for a fundamental business idea, connecting functionality and data models.

As shown in Figure 3, the design of the system is characterized by the presence of three primary microservices, namely the Receiver API, the Command API, and the Query API.

These microservices encapsulate separate, distinct duties within the order management BC. The 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 order data persistence and ensuring consistent interactions with the writing database.

On the other hand, the Query API enables the retrieval of order information by directly combining with the read database. These two APIs provide services to user interface (UI) clients. The concept of segregation fosters a modular and easy-to-maintain system architecture, hence increasing resilience to the inherent complexities of order management processes. The units of work have defined limits that are consistent with the CQRS.

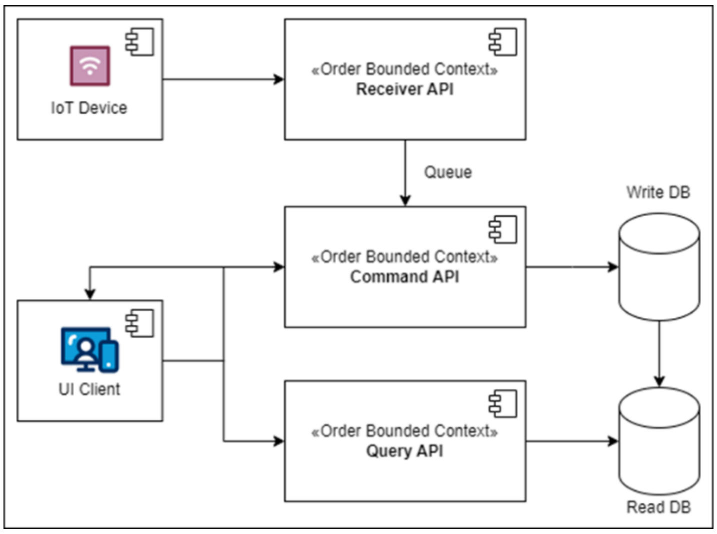

Figure 3. UML component diagram that illustrates the structure and relationships of microservices within their respective BC

Another feature of CQRS, especially when structured as a series of reusable requests and responses, is the use of the "mediator" pattern [27]. The mediator facilitates communication between components by providing a single interface for sending requests, which are then routed to in-process handlers. In this architecture, commands and queries represent requests, and results and data represent responses. Both sorts of requests and responses are commonly linked to user actions. To further extend the capabilities of the mediator pipeline, additional behaviors, such as contextual logging, metrics, validation, and authorization, can be integrated. For example, base algorithms may be placed at the top level by having an abstract class BaseHandler<TCommand> that inherits the ICommandHandler<TCommand> interface. So, at this level of abstraction, Serilog, Azure App Insights, Fluent Validation, and Automapper will let the developer access the event bus, map functions, and validation logic.

The core arrangement of DDD consists of the application, domain model, and infrastructure layers, as mentioned above.

Figure 4 illustrates the structuring of the layers into separate .NET assemblies.

The figure represents the project's structural organization and shows a logical arrangement of various containers for predetermined objects. The “Orders API” is the top-level hierarchy, which includes the “Orders Command API,” the “Orders Query API,” and the “Orders Receiver API.” This Web API enables communication between the “Business,” “Core,” and “Persistence” assemblies. The “Core” assembly serves as the central hub for commands, queries, and validation models.

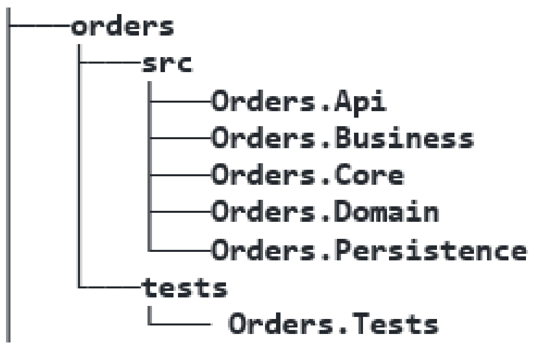
```
├───orders
    ├───src
    │   ├───Orders.Api
    │   ├───Orders.Business
    │   ├───Orders.Core
    │   ├───Orders.Domain
    │   └───Orders.Persistence
    └───tests
        └─── Orders.Tests
```

Figure 4. DDD organized project structure

The business assembly contains the command and query handlers as well as the connections to third-party services. The domain assembly, on the other hand, contains aggregates, entities, events, and data transfer objects (DTOs). Finally, the persistence assembly includes the necessary repository classes for performing data storage and retrieval operations. Moreover, the test project, which is separate from the "source" directory, consists of a comprehensive set of integration tests created using the test-driven development (TDD) methodology. This architecture guarantees a resilient and easily manageable foundation of code, adhering to the most effective methods in the field of software engineering.

3.2. Ubiquitous Language via Functional Programming

Ubiquitous language is a linguistic framework used in DDD to facilitate cohesive communication among team members regarding high-quality software code. It supports the process of defining and determining the dimensions of event handlers. The use of ubiquitous language improves the process of building specialized software by describing it via core ideas and their associated subprocesses. Successful execution requires 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 [28]. 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 ubiquitous language, the following generic structure of the "Either" type is proposed:

- A property of the Boolean data type called ***IsSuccessful.***
- A generic function called ***Match***, accepts two parameters: Func<T, TResult> success and Func<TException, TResult> error.
- A generic function called ***Map,*** uses ***Match*** internally to return another Either<TResult, TException> by accepting the mapping function.
- *A* generic function called ***flatMap***, which is similar to ***Map*** but skips wrapping the success value into an Either.

The "Match" method abstracts the success/error condition and necessitates the handling of both occurrences. The appropriate way to use an "Either" type is to consistently supply both handlers, since attempting to handle just one instance (such as only the success state) would result in a compiler error.

On the other hand, the "Map" function examines whether the "Either" has a value that signifies success. If so, it applies a function that modifies the value. Alternatively, in the case of an exception, it immediately provides the exception value in a "transformed" structure. The "Map" function behaves as follows [29]: (C<T>, (T => T2)) => C<T2>

The method accepts the container type C<T> and applies the specified (T => T2) function to the inner value. In this regard, it is worth mentioning the *functors*, since these are the types that implement a map function in FP. Furthermore, the flatMap function has a strong resemblance to the map, the key distinction being that it only takes transformation functions that yield another "Either." This enables software developers to avoid repeatedly wrapping up the outcome. The flatMap function behaves as follows: (C<T>, (T => C<T2>)) => C<T2>

In the context of FP, types that include a flatMap function, among other features, are referred to as *monads*. In summary, the fields and functions of the "Either" monad offer a streamlined method of chaining operations, making the code more readable and maintainable. As an example of this, the following structure describes the process for creating a new order.

```
|———— POST HTTP request with input data
| Flat Map  |———— validation of input fields or fail
| Flat Map  |———— check duplicate content or fail
| Flat Map  |———— persist in the database or fail
| Flat Map  |———— forwarding to a message queue
| Match     |———— matching the result with either a
                  success object and an HTTP 201
                  response code or a predefined error
                  structure with a code in the range of
                  400-500.
```

3.3. Referencing the Event Sourcing

As shown above, the adoption of CQRS can influence several aspects, such as storage techniques and data distribution [30]. In this context, a significant element is the transition in the software mindset from "models to persist" to "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 aggregable events. ES is a pattern that differs from traditional data storage methods in that it encapsulates 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 problems with data retrieval efficiency. To address this problem, ES incorporates the notion of "snapshots," which represent the aggregates from the DDD. Moreover, the use of ES is intrinsically aligned with event-driven architectures [31], facilitating the dissemination of targeted event notifications. Because it cannot be changed, this pattern protects the accuracy of data, makes it easier to track all activities related to a domain, and makes it easier to share data in distributed systems. The capability to replay events offers flexibility in processing and deriving various data projections that have the potential to be a primary source.

The event store database [32] is a specialized storage system based on the ideas of ES. The integral character of this pattern stems from its goal of continuously storing events that indicate changes in a system's state rather than storing the state itself. The primary goal of this database is to serve as a repository where new data may only be added, not destroyed, and old data cannot be changed. This design feature ensures that once an event has been recorded, it cannot be modified, preserving the historical record's correctness and chronological order. Another aspect of the database is its capacity to reconstitute system states at any point in time. By using these databases, companies have the potential to acquire detailed information of system behaviors and patterns, which facilitates the adoption of domain-driven decision-making processes and extensive auditing functionalities.

The schema of the suggested data store encompasses two primary database tables: "streams" and "events". The Nuget package Marten, a .NET Transactional Document DB and Event Store that exclusively works with PostgreSQL, serves as the foundation for this schema. Streams serve as a foundation for organizing and categorizing events. They provide a history of an aggregate, enabling state reconstruction, concurrency control, scalability, and interoperability. Table 4 provides a description of the recommended persistent model.

Table 4. Description of the "streams" ES table

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. They capture 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 rather active, ensuring consistency, accountability, and adaptability. They enable detailed data analysis, which enables the administrators to get deep insights and make informed decisions. Table 5 describes the proposed structure.

Table 5. Description of the "events" ES table

Field	Description
ID	Unique identifier for each event.
StreamID	Connects events to their corresponding stream, establishing a relationship with the streams table.
SeqID	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. Discussion

This section aims to assess the effectiveness of the fundamental DDD elements in enhancing an Azure cloud system. This analysis will not only provide valuable insights for the academic discussion, but it will also establish distinct programming principles by addressing the research question. The purpose of the findings is to provide practical advice for software developers and architects who are responsible for creating robust data structures and algorithms.

The Azure cloud, which consists of more than 200 products, is specifically designed to facilitate the creation and implementation of innovative solutions. Managed cloud platforms simplify operations by requiring only resource configuration and source code implementation. Nevertheless, these benefits are offset by associated expenses that need to be justified through the IT department. To showcase this advanced methodology based on the architecture from the previous section, figure 5 depicts a set of IaaS and PaaS services.

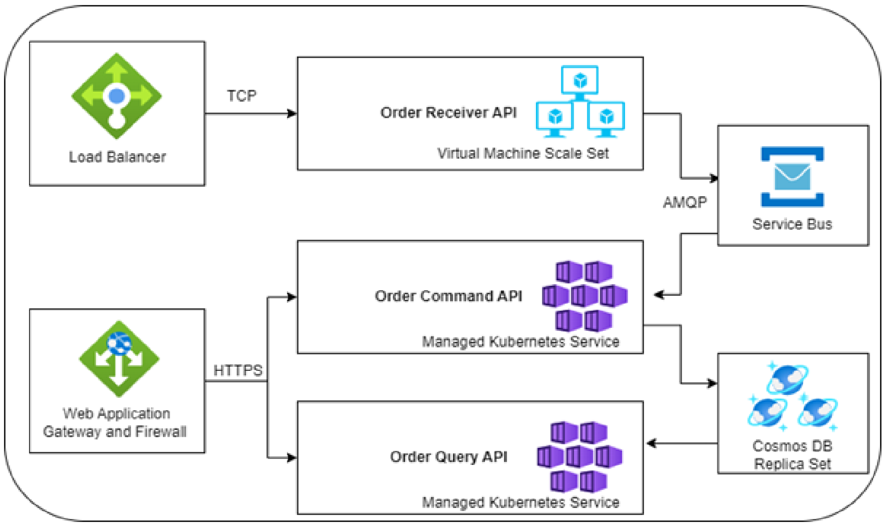
Load Balancer
TCP
Order Receiver API
Virtual Machine Scale Set
AMQP
Service Bus
Order Command API
Managed Kubernetes Service
HTTPS
Web Application Gateway and Firewall
Cosmos DB Replica Set
Order Query API
Managed Kubernetes Service

Figure 5. Diagram of high-level cloud services

The list includes a load balancer that distributes incoming traffic to the Order Receiver API. This API is deployed on a virtual machine scale set. Utilizing a service bus enables independent communication between services, thereby improving the system's robustness and capacity for growth. Furthermore, the Order Command API and the Order Query API are implemented on Managed Kubernetes Services, thus enhancing the ability to scale and effectively manage containerized applications. Cosmos DB replica sets are implemented to ensure data availability and fast access in multiple regions. The translation process from component to high-level abstraction underscores the integration of diverse capabilities necessary to meet the demands of new features and their increasing complexity [33]. The findings indicate the need to implement a comprehensive set of technologies and patterns in order to maximize benefits and ensure the seamless operation of system components.

Monitoring and analytics play a vital role in cloud-based management systems [34], [35]. Azure Monitor plays a crucial role in this ecosystem, consolidating data from various sources. Different components of the infrastructure, including mobile and web applications and APIs, containers, virtual machines, load balancers, and databases, provide data for App Insights.

Visualization tools, such as dashboards and workbooks from Power BI, improve user involvement and aid in the understanding of data.

DDD solutions do have specific limitations that can lead to heightened complexity. For example, the decisions regarding persistence with ES might result in the gathering of large amounts of event logs, which can pose difficulties regarding long-term maintenance and support. Programmers accustomed to traditional object-oriented programming (OOP) may find the limitations of FP in the .NET framework to be inefficient and challenging to learn.

Integrating and conducting unit testing within a DDD framework requires careful planning. The reason for this is the nature of domain models, which can make it difficult to isolate individual classes. In Azure, the wide array of services and configurations can sometimes be overwhelming, causing confusion when trying to make the best choices. Also, depending only on .NET and Azure could result in vendor lock-in, which would restrict the system's flexibility and its potential to be migrated to alternative platforms such as Java and Amazon Web Services (AWS) or Go and Google Cloud Platform (GCP).

5. Conclusion

Motivated by the growing interest in DDD within the software development community, this study aimed to assess the implications of incorporating DDD into cloud-native services using Azure and .NET. The ideas of microservices, BC, and CQRS are very important to this adoption because they make sure that each part is logically separate and can work on its own. The practical use of FP and ES persistence was investigated. The paper outlines both the benefits and challenges of using these advanced programming paradigms, giving significant insights for organizations and developers navigating comparable technological shifts. Implementing TDD practices ensures that the codebase is durable and flexible in the face of modifications. The efficiency of all these patterns is based on managing complex web platforms that require ongoing integration, delivery, and flexible resource allocation. The inclusion of .NET alongside Azure emphasizes its importance and ability to foster creativity and growth. To summarize, incorporating DDD into cloud-native apps not only follows established industry standards but also addresses the changing demands of modern software development. This strategy keeps applications strong, adaptive, and capable of fulfilling new requirements.

Given that this article primarily focuses on examining the patterns and principles for handling complexity in cloud-based services, it is considered important to pay more attention to the technical aspects and communication techniques used by DDD-oriented microservices.

Acknowledgements

This research is financially supported by NPI-67/2023 from the University of Economics - Varna Science Fund.

References:

[1]. Sangabriel-Alarcón, J., et al. (2023). Domain-driven design for microservices architecture systems development: A systematic mapping study. *Proceedings of the 11th International Conference in Software Engineering Research and Innovation (CONISOFT)*, 25-34.

[2]. Satapathi, A., & Mishra, A. (2022). *Developing Cloud-Native Solutions with Microsoft Azure and .NET*. Apress.

[3]. Kapferer, S., & Zimmermann, O. (2020). Domain-specific Language and Tools for Strategic Domain-driven Design, Context Mapping and Bounded Context Modeling. *MODELSWARD,* 299-306. Doi: 10.5220/0008910502990306

[4]. Litvinov, O., & Frolov, M. (2024). On the migration of domain-driven design to CQRS with event sourcing software architecture. *Information Technology: Computer Science, Software Engineering and Cyber Security, 1*(1), 50-60.

[5]. 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*. Lang.

[6]. Zhong, C., et al. (2024). Domain-driven design for microservices: An evidence-based investigation. *IEEE Transactions on Software Engineering, 50*(6), 1425-1449

[7]. Jiménez, J. V., & Sánchez, A. G. (2024). *Kubernetes and Cloud Native Associate (KCNA) Study Guide: In-Depth Exam Prep and Practice.* O'Reilly Media.

[8]. Rocha, Á., et al. (2020). *Trends and Innovations in Information Systems and Technologies.* Springer.

[9]. Özkan, O., Babur, Ö., & Brand, M. V. D. (2023). Domain-Driven Design in Software Development: A Systematic Literature Review on Implementation, Challenges, and Effectiveness. *arXiv preprint arXiv:2310.01905*.

[10]. Buonanno, E. (2022). *Functional Programming in C#.* Simon and Schuster.

[11]. Wlaschin, S. (2018). *Domain modeling made functional: Tackle Software Complexity with Domain-Driven Design and F#.* The Pragmatic Bookshelf.

[12]. Khorikov, V. (2020). *Unit Testing Principles, Practices, and Patterns*. Simon and Schuster.

[13]. Williams, L., Kudrjavets, G., & Nagappan, N. (2009). On the Effectiveness of Unit Test Automation at Microsoft. *ISSRE 2009, 20th International Symposium on Software Reliability Engineering*, 81-89.

[14]. Runeson, P., Host, M., Rainer, A., & Regnell, B. (2012). *Case study research in software engineering: Guidelines and Examples*. John Wiley & Sons.

[15]. Jordanov, J., & Petrov, P. (2023). Domain driven design approaches in cloud native service architecture. *TEM Journal, 12*(4), 1985–1994. Doi: 10.18421/TEM124-09

[16]. Pham, T. A. (2024). *Best Popular Backend Frameworks by Performance Benchmark Comparison and Ranking in 2024.* DEV Community. Retrieved from: https://dev.to/tuananhpham/popular-backend-frameworks-performance-benchmark-1bkh [accessed: 02 May 2024].

[17]. Ramel, D. (2022). *VS Code and Visual Studio Rock the 2022 Stack Overflow Developer Report.* Visual Studio Magazine. Retrieved from: https://visualstudiomagazine.com/articles/2022/06/23/stack-overflow-2022-survey.aspx [accessed: 03 May 2024].

[18]. Ozkaya, M. (2024). *Why .NET Rocks: The Latest Scoop on .NET 8 and C# 12.* Medium. Retrieved from: https://mehmetozkaya.medium.com/why-net-rocks-the-latest-scoop-on-net-8-and-c-12-064cba68e4fe [accessed: 12 May 2024].

[19]. AIN. (2023). *Top 10 programming languages of 2023 in GitHub report.* Ain.ua.. Retrieved from: https://ain.capital/2023/11/15/top-10-programming-languages-of-2023-in-github-report/ [accessed: 05 July 2024].

[20]. Luxner, T. (2024). *Cloud computing Stats: Flexera 2024 State of the Cloud Report.* Flexera Blog. Retrieved from: https://www.flexera.com/blog/cloud/cloud-computing-trends-flexera-2024-state-of-the-cloud-report/ [accessed: 07 July 2024].

[21]. 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: 15 July 2024].

[22]. 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: 18 July 2024].

[23]. Singh, U. (2022). *Order Management System - UX case study*. Medium. Retrieved from: https://medium.com/@urvashi_s/order-management-system-ux-case-study-f1a2f874161f [accessed: 19 July 2024].

[24]. Pagell, M., & Wu, Z. (2009). Building a more complete theory of sustainable supply chain management using case studies of 10 exemplars. *Journal of supply chain management*, *45*(2), 37-56.

[25]. Cwalina, K., Barton, J., & Abrams, B. (2020). *Framework design guidelines: conventions, idioms, and patterns for reusable. net libraries*. Addison-Wesley Professional.

[26]. Phelan, S. (2011). Case study research: design and methods. *Evaluation and Research in Education, 24*(3), 221-222.

[27]. Pai, P., & Xavier, S. (2017). *. NET Design Patterns*. Packt Publishing Ltd.

[28]. Teatro, A., Eklund, M., & Milman, R. (2018). Maybe and Either Monads in Plain C++17. *2018 IEEE Canadian Conference on Electrical & Computer Engineering (CCECE)*, 1-4.

[29]. Nikolov, D. (2019). *Shipping pseudocode to production*. DotNetCurry. Retrieved from: https://www.dotnetcurry.com/patterns-practices/1497/deploy-pseudocode-production [accessed: 20 August 2024].

[30]. Garofolo, E. (2020). *Practical microservices: Build Event-Driven Architectures with Event Sourcing and CQRS.* Pragmatic Bookshelf.

[31]. Rocha, H. F. O. (2021). *Practical Event-Driven microservices architecture: Building Sustainable and Highly Scalable Event-Driven Microservices*. Apress.

[32]. Esser, S., & Fahland, D. (2019). Storing and querying multi-dimensional process event logs using graph databases. *Lecture notes in business information processing*, *362.*

[33]. Petrov, P., et al. (2021). Petrov, P., Radev, M., Dimitrov, G., Pasat, A., & Buevich, A. (2021). A systematic design approach in building digitalization services supporting infrastructure. *TEM Journal: Technology, Education, Management, Informatics*, *10*(1), 31-34. Doi: 10.18421/TEM101-04

[34]. Valiramani, A. (2022). *Microsoft Azure Compute: The Definitive Guide*. Microsoft Press.

[35]. Petrov, P., et al. (2022). Petrov, P., Radev, M., Dimitrov, G., & Simeonidis, D. (2022). Infrastructure Capacity Planning in Digitalization of Educational Services. *International Journal of Emerging Technologies in Learning (iJET)*, *17*(3), 299-306. Doi: 10.3991/ijet.v17i03.27811