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

Jordan Jordanov1, Pavel Petrov1, Ivan Kuyumdzhiev1

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

*Abstract –* This study examines the incorporation of Domain-Driven Design (DDD) into the cloud computing elements of the Microsoft ecosystem. The study seeks to showcase an effective approach for developing a software architecture that is capable of scaling, easy to maintain, and highly efficient. This research aims to address the need for connecting the theoretical concepts of DDD with their practical applications in cloud-native services. The use of a case study approach indicates that DDD is an important factor to consider when it comes to the application and data layers in Platform-as-a-Service (PaaS) and Infrastructure-as-a-Service cloud models. The case study highlights the potential of implementing DDD) as a standard approach to enhance the efficiency of software architecture in cloud environments.

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

1. **Introduction**

Domain-Driven Design (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. This approach encourages technical and domain experts to work together in developing software that is flexible and can easily adapt to evolving business requirements. Although this approach shows potential, there is still a notable

DOI: 10.18421/TEMxx-xx

<https://doi.org/10.18421/TEMxx-xx>

**Corresponding author: -** ,

*affiliation*

**Email:** -

*Received: -----.*

*Revised: -----.*

*Accepted: -----.*

*Published: -----.*

Description: Description: Description: Description: Description: Description: Description: by-nc-nd-.png© 2023 ----; published by UIKTEN. This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 4.0 License.

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

lack of research in practical studies that examine the relationship between DDD concepts and cloud development frameworks for constructing web, mobile, desktop or IoT applications. This paper aims to investigate the implementation of DDD concepts using .NET and their deployment on the Azure platform. This study aims to offer a comprehensive perspective on the strategic decisions, architectural elements, and results related to these integrations. To accomplish this, the study utilizes a research methodology that involves multiple use cases.

DDD provides a philosophy and a set of guidelines, including bounded contexts, and ubiquitous language. In addition, there are programing models like “Aggregates” and “Value objects” as well as patterns as Command Query Responsibility Segregation (CQRS) and Event Sourcing (ES). These principles are especially applicable to microservices, functional programming, and event driven development. In addition, an integrated test suite is supposed to guarantee the integrity of all of them.

Table 1 shows the differences among two main cloud service models: Infrastructure as a Service (IaaS), Platform as a Service (PaaS). Within the IaaS model, the cloud provider assumes responsibility for managing fundamental resources like 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.

*Table 1. Classification across two cloud models*

|  |  |
| --- | --- |
| Infra as a Service | Platform as a Service |
| Application | Application |
| Data | Data |
| Runtime | Runtime (Managed) |
| Middleware | Middleware (Managed) |
| OS | OS (Managed) |
| Virtualization (Managed) | Virtualization (Managed) |

Among the models that have been presented, 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. In this context, the importance of DDD concepts becomes evident, particularly for the "data” and “applications" layers, which should be the focus.

The microservices architecture is defined by the process of breaking down applications into small, autonomous services, initiating the establishment of one of the cloud-native standards. Each microservice, which contains a specific business function, can be deployed, scaled, and maintained independently. This allows for the utilization of the natural flexibility and durability of cloud platforms. Microservices facilitate the implementation of continuous integration, continuous delivery, and dynamic resource allocation. As stated by the Cloud Native Computing Foundation (CNCF), microservices allow the creation of system components that are loosely connected, resilient, manageable, and observable. When used in conjunction with strong automation, they enable engineers to make significant and predictable changes frequently, with minimal effort. There are numerous studies that have examined the leading corporations like Netflix and Uber. These companies are supporting online platforms, which are offering a wide range of services. The software responsible for these services frequently releases new versions, deploying thousands of instances on a weekly basis.

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

There is at least one aggregate 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 as a domain model by grouping multiple entities together under a single conceptual framework.

In order to design an approach for constructing aggregates and other DDD models, this study examines the practical aspects of using functional programming (FP). FP primarily focuses on two distinct features: maintaining the integrity of method signatures and ensuring referential transparency. The concept of method signature honesty ensures that a function's signature accurately and comprehensively represents all possible input and output values. Referential transparency guarantees that a function's output remains consistent for a given input, without any additional side effects. Furthermore, FP is supposed to the reduces code complexity, making it easier to understand and analyze logically. It also considered to simplifies unit testing and enhances the modularity and composability of software components.

The importance of immutability in FP is crucial, as mutable operations have the potential to introduce “dishonesty” into the code. The absence of clearness hampers our capacity to participate in rational reasoning, making the process of debugging more complex and creating barriers to multi-threading. Moreover, the utilization of FP is improved by the implementation of CQRS and the integration of fundamental domain logic. 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. The functional approach employed in this context utilizes extension methods to enhance 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. To support these needs test doubles, particularly mocks, can be utilized to replace dependencies with unpredictable behavior, thus achieving the desired outcome. Unit testing offers a key benefit of ensuring the integrity of existing functionality while allowing for efficient modifications to code.

Based on a case study from the Computer Science department at North Carolina State University, 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:

***Code coverage = Lines of code covered / Overall number of lines***

This ratio provides a numerical value that reflects the thoroughness of testing and helps identify untested parts of the code. High code coverage is often associated with higher software quality, as it indicates that a significant portion of the code has been executed during testing, potentially uncovering defects and ensuring that the software behaves as expected under various conditions. However, achieving 100% code coverage does not guarantee the absence of bugs, as it does not account for the quality or comprehensiveness of the tests themselves. Nonetheless, striving for higher code coverage can contribute to more robust and maintainable code by encouraging comprehensive testing practices. In summary, the code coverage of DDD component serves as a useful benchmark for guiding the development process toward better software reliability and performance of the cloud microservices.

1. **Methodology**

The aim of this study is to explore and give in-depth understanding of the software development with DDD, CQRS and ES patterns via .NET and Azure technologies. To reach this goal, the selection of an appropriate research approach is an important step. Regarding the uncertainty and a lag of research for the implementation of the DDD concepts, this 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 research 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 propositions, 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 preview’s research.

***2.1. Tools & Technologies***

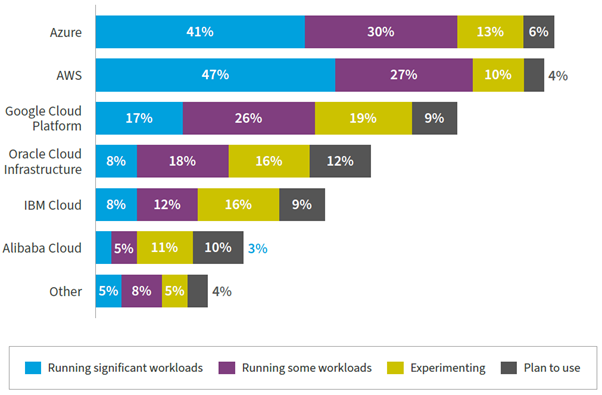
The .NET is widely acknowledged as a prominent option for developing scalable and robust corporate applications. Based on statistics provided by “Techempower round 22 october 2023” [], it has been observed that ASP.NET exhibits efficiency and performance compared to several alternative web application platforms and full-stack frameworks, given 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 |

ASP.NET Core is noted to be faster than NodeJS, Fiber, Laravel, Django and Spring. This significant performance advantage showcases ASP.NET Core's efficiency and capability in handling high-performance web applications. Microsoft has outlined a strategic plan [] for the future development and maintenance of .NET, guaranteeing regular upgrades and expanded library support which boasts 5.7 million monthly active developers within the “Visual Studio family” []. Moreover, .NET Core has been recognized as the *"#1 Most Loved Framework*" for three consecutive years (2019, 2020, 2021) according to Stack Overflow surveys []. The .NET ecosystem is highly active in the open-source space, with its GitHub repository being ranked among the "*Top 30 Highest Velocity OSS Projects*." According to data conducted by Github, C#, a primary language in the .NET ecosystem, is listed among the "*Top 5 Languages*." This ranking indicates C#'s popularity, highlighting the widespread adoption. Approximately "40% of New to .NET are Students," as indicated by a download survey. This statistic highlights the growing interest and adoption of the .NET framework among the academic’s fields, ensuring innovations within the ecosystem. Additional factors include the use of the supplemental libraries such as *Minimal API, 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 Integrated Development Environment (IDE) like the Visual Studio. This integration enhances the development experience and ensures interoperability within the broader Microsoft ecosystem. Figure 1 obtained from the "Flexera's 2023 State of the Cloud Report," [] showcases the use trends of different public cloud providers across enterprises.



*Figure 1. Cloud service providers used by organizations in the public sector.*

The findings derived from a sample of 750 participants indicate that Azure has emerged as a prominent player in the cloud services market. Specifically, 41% of firms are utilizing its platform to execute substantial workloads, 30% are using it for certain tasks, and approximately 13% are currently in the testing phase. According to data from Microsoft, Azure has demonstrated 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, reinforcing its dominance in the market. Notably, major clients such as Samsung, Boeing, eBay, and BMW rely on Azure's services. Based on the collected data, it can be deduced that the use of .NET and Azure is a favourable choice 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 establishing the empirical foundation of this research. This study is motivated by multiple cases, specifically drawing on Microsoft reference applications „eShopOnContainers“ [] and „eShopOnAzure“ []. The emphasis on functionalities related to order administration serves as a framework for streamlining the more complex aspects of enterprise-level systems. Below, we present three relevant demonstrations for these systems.

*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. These technologies consequently improving convenience for both consumers and enterprises. |
| C | Supply Chain Management | Software platforms for real-time visibility, ensuring efficient flow of goods, information, and finances. |

The process of data collection aligns with the functional and non-functional requirements identified through a literature review of existing academic research. This case study primarily emphasizes an analysis of the implementation procedures related to the registration of order records and the subsequent modifications made by end users and external devices.

***2.3. Conceptual framework***

Within the conceptual framework illustrated in Figure 3, it is combined domain-centric design with several architectural patterns for design and development of cloud microservices. BC, UL, Entities, Value Objects, and Aggregates capture and articulate the complexities of the business domain. CQRS is employed to separate the concerns and ES is incorporated to maintain a reliable audit trail of changes. TDD is driving the design of the system through the “tests first approach”. In addition to that, the case study methodology provides a practical validation of the framework.



*Figure 3. Conceptual framework model of the DDD Approaches in the Cloud Environment*

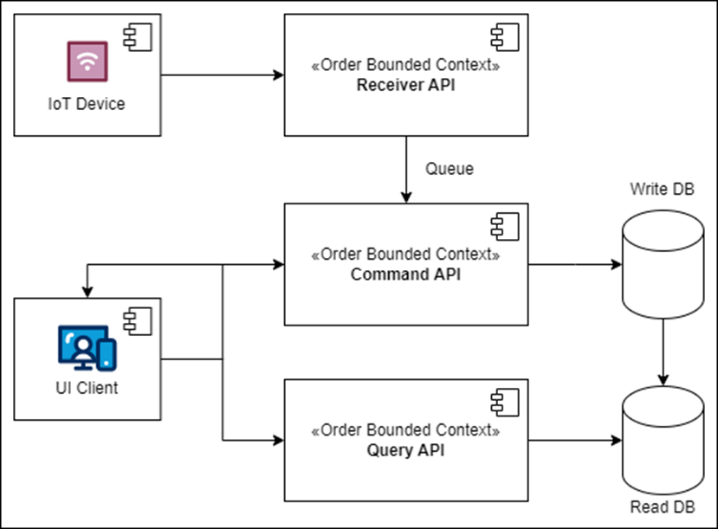
Applications these days rarely fit neatly into a single paradigm, instead, they exhibit varying degrees of complexity. Consequently, attempting to apply a single modeling strategy across is ineffective. Recognizing this, we consider the case study methodology as a strategy, as it aligns with the research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

1. **Results**

This section seeks to investigate the effectiveness of BC, CQRS, UL and ES as part of the DDD in improving system modularity, scalability, and maintainability. Addressing the research question will not only provide valuable insights to the academic discussion but also establish clear programming principles. In this context, the results of this study aim to provide valuable direction for software developers and architects in designing and implementing suitable data structures and algorithms.

***3.1. Applying BC and CQRS 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 idea, connecting functionality and data models. As seen in Figure 4, the design of the system is characterized by the presence of three primary microservices, namely the Receiver API, Command API, and Query API.



*Figure 4. UML Component Diagram that 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.

Another aspect of CQRS, particularly when designed as a series of re-usable requests and responses, is the adoption of the “Mediator pattern”. The “mediator” simplifies this process by providing a single interface for sending requests, which are then routed to in-process handlers. In this framework, commands and queries represent the requests, while results and data represent the responses. They are typically mapping 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 could be placed on top level by having “*abstract class BaseHandler<TCommand>”* which inherits “*ICommandHandler<TCommand>”* interface*.* Consequently, on this level of abstraction, the developer is going to have access event bus, mapping, and validating logic via Serilog, Azure App Insights, Fluent Validation and Automapper.

The core arrangement of DDD consist of the application, domain model, and infrastructure layers, as previously mentioned. The layers are structured into separate .NET assemblies, as shown in Figure 5.



*Figure 5. DDD organized project structure*

The diagram depicts the structural organization of the projects, demonstrating a methodical arrangement of different containers for predetermined objects. The 'Orders Api' is the top-level hierarchy that includes the 'Orders Command Api,' 'Orders Query Api,' and '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. Simultaneously, the "Business" assembly accommodates the command and query handlers and establishes connections with third-party services. The "Domain" assembly serves as a storage facility for aggregates, entities, events, and data transfer objects (DTOs). 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***

The notion of 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, we propose the following generic struct of the “Either” type:

* property of type bool called ***IsSuccessful.***
* generic function called ***Match***, which accepts 2 parameters: *Func<T, TResult> success* and *Func<TException, TResult> error.*
* generic function called ***Map,*** which uses ***Match*** internally to return another *Either<TResult, TException>* by accepting mapping function.
* generic function called ***FlatMap***, 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 of using an “either” type is by consistently supplying 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 if the “either” has a value that signifies success, and if so, it applies a function that alters the value. Alternatively, in the case of an exception, it will immediately provide the exception value in a "transformed" structure. The “Map” function behaves as follows:

***(C<T>, (T => T2)) => C<T2>***

The method accepts type container C<T> and applies the specified (T => T2) function on the inner value. In this regard, it worth mentioning the *Functors*, since these are the kinds that implement a map function in FP. Furthermore, FlatMap function has a strong resemblance to Map, with the key distinction being that it only takes transformation functions that yield another “either”. This enables software developers to avoid repeatedly wrapping the outcome. The Flatmap function behaves as follows:

***(C<T>, (T => C<T2>)) => C<T2>***

In the world 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. An example can be review in the following structure of the process for creating a new order.

├─── **POST HTTP request with input data**

│ Flat Map ├─── validation of input fields

│ Flat Map ├─── check duplicate content

│ Flat Map ├─── persist in the database

│ Flat Map ├─── forwarding to a message queue

│ Match ├─── result with either success object and HTTP 201 response code or predified error structure with code in diapason of 400-500.

***3.3. Referencing the Event Sourcing***

As previously discussed, the adoption of the CQRS can influence several aspects, such as storage techniques and data distribution. In this context, 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 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. To address the problem, ES incorporates the notion of "snapshots," which represents the aggregates 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.

The event store database is a specialized storage system that is built around the principles of ES. The integral nature of this pattern 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.

*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. |

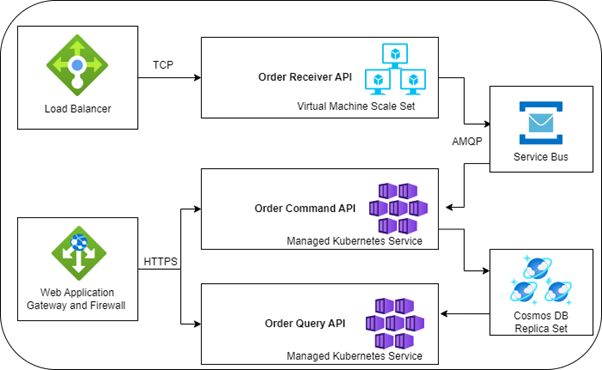
Events are fundamental units in event sourcing, capturing state changes and actions within a system. They provide historical immutability, auditability, temporal insights, decoupling, compensation, and error handling. Events are not passive records but active, ensuring consistency, accountability, and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

1. **Discussion**

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 streamline operations by necessitating only the configuration of resources and the implementation of the source code. 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 previews section, figure 6 includes a set of IaaS and PaaS services.



*Figure 6. Diagram of high-level cloud services*

In the list, we can find 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 Order Query API are implemented on Managed Kubernetes Services, which enhances the ability to scale and effectively manage containerized applications. Cosmos DB Replica Sets are implemented to guarantee data availability and fast data 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. The findings indicate the need to implement a comprehensive set of technologies and patterns to ensure the seamless operation of system components and to maximize benefits.

Monitoring and analytics play a vital role in cloud-based management systems. Azure Monitor has a crucial role in this ecosystem, consolidating data from various sources. Insights are obtained from different components of the infrastructure, such as mobile and web applications and APIs, containers, virtual machines, load balancers and databases. Visualization tools, such as dashboards and workbooks from Power BI, improve user involvement and aid in understanding data. In addition, Azure Monitor integrates alert systems and autoscaling capabilities to facilitate proactive system management, guaranteeing timely responses to anomalies or resource limitations.

Nevertheless, DDD solutions do have specific limitations that can lead to heightened complexity. For example, the decisions regarding the persistence with ES, might result in gathering of large amounts of event logs, which can pose difficulties in terms of long-term maintenance and support. In addition, the limitations related to FP in the .NET Core framework can lead to inefficiencies and pose a challenging learning process for programmers who are used to traditional Object-Oriented Programming (OOP).

In addition, integrating and conducting unit testing within a DDD framework requires careful planning because of the nature of domain models, which can make it challenging to isolate individual classes. In the context of Azure, the wide array of services and configurations can sometimes be overwhelming, causing confusion when trying to make the best choices. Additionally, depending heavily 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 like Java and Amazon Web Services (AWS), or Go and Google Cloud Platform (GCP).

1. **Conclusion**

Inspired by the increasing popularity of the software design with DDD, the study aimed to investigate the impact of adopting DDD on cloud-native Azure and .NET services. The principles of microservices, BC, and CQRS are crucial as they facilitate the logical segregation and autonomy of distinct components. Furthermore, the study highlights the practical implementation of FP and ES persistence, along with the advantages and difficulties associated with their adoption. Implementing TDD practices guarantees the codebase's durability and flexibility in the face of modifications. The effectiveness of all these patterns is in managing complex online platforms that necessitate continues integration, delivery, and flexible resource allocation. The incorporation of .NET alongside Azure highlights its significance and capacity to promote creativity and advancement. In summary, integrating DDD into cloud-native applications not only adheres to established industry standards but also addresses the evolving demands of contemporary software development. This approach ensures that applications remain robust, adaptable, and capable of meeting emerging requirements.

Acknowledgements (If any)

These and the Reference headings are in bold but have no numbers. Titles and text of sections references and acknowledgements should be formatted with 10 – point font, yet text in acknowledgements section should be in italic font. Include the names of the funding agencies or organizations, grant numbers - number of project, and any relevant details about the funding.

**References**

References section is not enumerated. The use of hyperlinks should be avoided as much as possible. When including in-text references, they should be on the same level as the rest of the text e.g.: “References give proper credit to all work included [1]”. For in-text references TEM Journal prefers IEEE style (reference numbers in square brackets “[2], [3]”) in combination with APA style (the use of authors’ names with reference numbers, yet it is better to use only IEEE format as much as is possible). When citing the author's name, use the following format: 'as shown by Brown [4].' In cases where only the reference itself is included, use this format: 'References are of great importance in scientific papers [2]’.When including multiple sources, format them as follows: [1], [2], [3].

When introducing references in the reference list use 10 point font, following the form provided below. Examples of the form:

1. Wong, B., & Kokko, H. (2005). Is science as global as we think?. *Trends in ecology & evolution*, *20*(9), 475-476.
2. Hennessy, J. L., & Patterson, D. A. (2012). *Computer architecture: a quantitative approach*. Elsevier.
3. Miller, T. (2019). *Enhancing readiness: An exploration of the New Zealand Qualified Firefighter Programme* [Master's thesis, Auckland University of Technology]. Tuwhera.
4. Herculano-Houzel, S., Collins, C. E., Wong, P., Kaas, J. H., & Lent, R. (2008). The basic nonuniformity of the cerebral cortex. *Proceedings of the National*