**Domain-Driven Design in Cloud Computing: .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. The study aims to address the need to connect the theoretical concepts of DDD with their practical applications in cloud-native services. Using a case study approach, this paper 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 (IaaS) 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**

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

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/>

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 are DDD concepts implemented in .NET services and deployed on Microsoft Azure platform?* The 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 (BCs), and ubiquitous language [2], [3]. In addition, 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 especially applicable to microservices, functional programming (FP), and event-driven development. In addition, an integrated test suite is used to guarantee the integrity of all of them [4].

Table 1 shows the differences between the two main cloud service models: 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].

*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 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 with regard to “data” and “applications” layers.

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 [6]. 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) [7], 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 have been numerous examining 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 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. In addition, a BC 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 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 acts as a domain model by grouping multiple entities together under a single conceptual framework.

To design an approach for constructing aggregates and other DDD models, this study examines the practical aspects of using FP. FP primarily focuses on two distinct features: maintaining the integrity of method signatures and ensuring referential transparency [10]. 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 reduce code complexity, making it easier to understand and analyze logically. It is also considered to simplify unit testing and enhance 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 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. 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 [11]. 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 [12]. 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 Department of Computer Science, 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 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.

1. **Methodology**

The aim of this study is to explore and provide an in-depth understanding of software development with DDD, CQRS, and ES patterns via Microsoft .NET and Azure technologies. Selecting an appropriate research method is an important step toward reaching this goal. There is currently an uncertainty and lag 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. Case study research was deemed an appropriate research method for reaching this goal. Case studies, representing qualitative research methods, are commonly used in computer and social science. According to Runeson et al. [14], 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, shown 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***

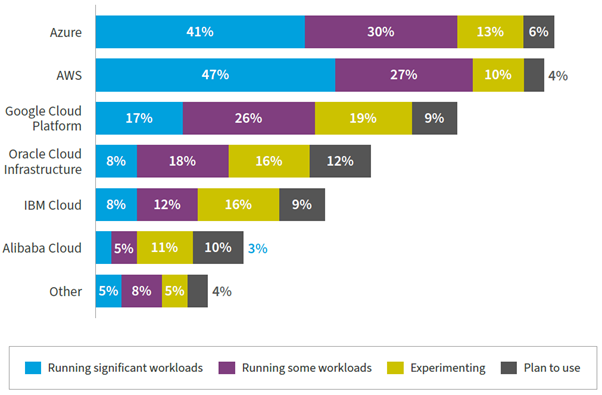
.NET is widely acknowledged as a key option for developing scalable and robust corporate applications. Based on statistics provided by “Techempower round 22 october 2023” [16], it has been observed that ASP.NET exhibits efficiency and performance compared to several alternative web application platforms 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 |

ASP.NET Core is acknowledged to be faster than NodeJS, Fiber, Laravel, Django, and Spring. This significant performance advantage showcases ASP.NET Core's efficiency and capability when handling high-performance web applications. Microsoft has outlined a strategic plan development and maintenance of .NET, guaranteeing regular upgrades and expanded library support. This plan boasts 5.7 million monthly active developers within the “Visual Studio” family [17]. Moreover, .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 highly active in the open-source space, with its GitHub repository being ranked among the “Top 30 Highest Velocity OSS Projects.” According to data collected by Github, C#, a primary language in the .NET ecosystem, is listed among the top five programming languages. [19] This ranking indicates C#'s popularity and widespread adoption. Approximately 40% of those who are new to .NET are students. This statistic highlights the growing interest in and adoption of the .NET framework in different academic fields, allowing for innovations within the ecosystem. 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. 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) 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.*

The findings, derived from a sample of 750 participants, indicate that Azure has emerged as a prominent player in the cloud services market. Some 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 approximately at approximately 13% of firms. According to data from Microsoft, Azure exhibited a substantial growth rate of 31% in the quarter ending March 2024. Meanwhile, Azure's extensive network of over 60 data centers 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. 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 establishing 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 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 Eenterprise-Level Systems*

|  |  |  |
| --- | --- | --- |
| Case | System |  |
| 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 improve 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 and 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 the design and development of cloud microservices. BC, ubiquitous language, entities, value objects, and aggregates capture and articulate the complexities of the business domain. CQRS is employed 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. In addition, the case study methodology provides a practical validation of the framework.



*Figure 2. Conceptual framework model of the DDD Approaches in the Cloud Environment*

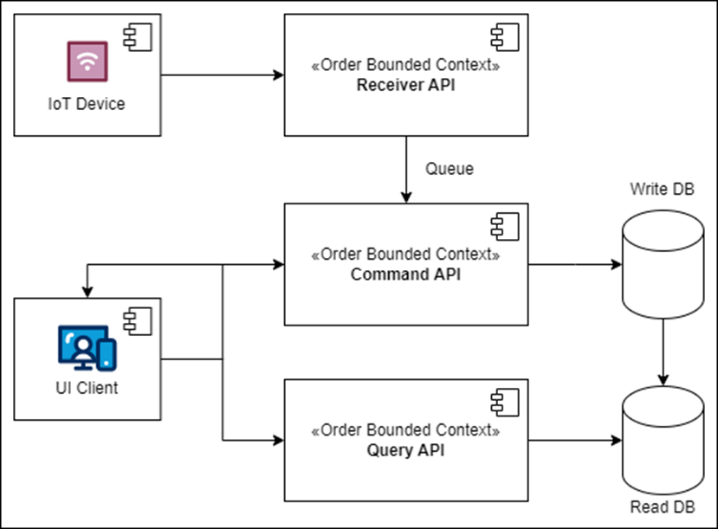
Applications currently rarely fit neatly into a single paradigm; instead, they exhibit varying degrees of complexity. Consequently, attempting to apply a single modeling strategy across all applications is ineffective. Recognizing this, we view the case study methodology as a strategy, as it aligns with 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.

1. **Results**

TODO

***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.



*Figure 3. UML Component Diagram that illustrates the structure and relationships of microservices within their respective BC*

These microservices encapsulate separate, distinct 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 combining with the read database. These two APIs provide services to user interface (UI) clients. The practice of segregation cultivates a system architecture that is modular and easy to maintain, thus 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 [27]. The mediator simplifies the communication process between components by providing a single interface for sending requests, which are then routed to in-process handlers. In this framework, commands and queries represent requests, while results and data represent responses. Both types of requests and responses are typically mapped 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*.* Consequently, at this level of abstraction, the developer will have access to the event bus, mapping, and validating logic via Serilog, Azure App Insights, Fluent Validation and Automapper.

The core arrangement of DDD consists of the application, domain model, and infrastructure layers, as mentioned above. The layers are structured into separate .NET assemblies, as shown in Figure 4.



*Figure 4. 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, 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. Simultaneously, the Business assembly accommodates the command and query handlers and establishes connections with third-party services, while the Domain assembly serves as a storage facility for 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 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 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, we propose the following generic structure of the “Either” type:

* A property of the Boolean data type called ***IsSuccessful.***
* A generic function called ***Match***, which accepts two parameters: Func<T, TResult> success and Func<TException, TResult> error.
* A generic function called ***Map,*** which uses ***Match*** internally to return another Either<TResult, TException> by accepting 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 alters 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 type of container 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 world of FP, types that include a flatMap function, among other features, are referred to as **m*onads***. 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 figure describes a 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.

*Figure 5. Process of establishing a new order*

***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 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 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. This pattern serves to guarantee the integrity of data, facilitate the 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 [32] is a specialized storage system 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, not deleted, and existing data cannot be modified. This design feature guarantees that once events are recorded, they cannot be changed, thus maintaining the accuracy and chronological order of the historical record. Another feature of the database is the way it enables the reconstruction of system states from any given point in time. Furthermore, with the use of these databases, companies have the potential to acquire a detailed understanding of system behaviors and patterns, facilitating 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. Streams serve as a foundation for organizing and categorizing events. They provide a comprehensive 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. |

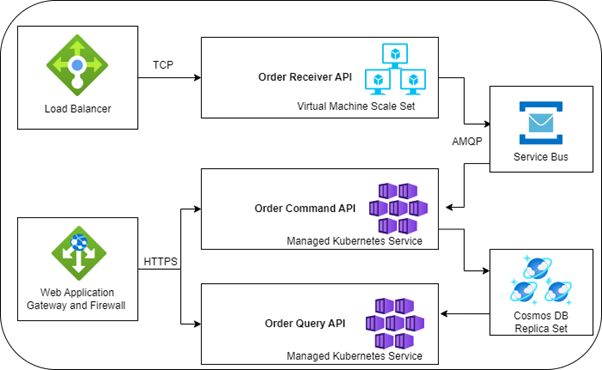
Events are fundamental units in event sourcing, capturing state changes and actions within a system. They provide historical immutability, auditability, temporal insights, decoupling, compensation, and error handling. Events are not passive records but active, ensuring consistency, accountability, and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

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 requiring 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 the previous section, figure 6 depicts a set of IaaS and PaaS services.



*Figure 6. 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 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 and to maximize benefits and ensure the seamless operation of system components.

Monitoring and analytics play a vital role in cloud-based management systems [33]. Azure Monitor plays 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 the understanding of data. In addition, Azure Monitor integrates alert systems and autoscaling capabilities to facilitate proactive system management, guaranteeing timely responses to anomalies and resource limitations.

Nevertheless, 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. In addition, the limitations related to FP in the .NET 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 such as 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.

Since this article mainly focuses on the patterns and principles of “tackling complexity in the heart of” cloud-based services, it is essential to continue the subject matter by exploring the technical aspects and communication channels.

Acknowledgements

This research is financially supported by NPD-331/2023 from University of Economics -Varna Science Fund.

**References**

1. Sangabriel-Alarcón, J., Ocharán-Hernández, J., Cortés-Verdín, K., & Limón, X. (2023). *Domain-driven design for microservices architecture systems development: A systematic mapping study*. International Conference on Software Engineering and Information Management (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 (pp. 299-306).
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-asa-service development and operation*. Peter Lang.
6. Zhong, C., Li, S., Huang, H., Liu, X., Chen, Z., Zhang, Y., & Zhang, H. (2024). *Domain-driven design for microservices: An evidence-based investigation*. IEEE Transactions on Software Engineering, vol. 50, no. 6, pp. 1425-1449
7. Jiménez, V., & Sánchez, G. (2024). *Kubernetes and Cloud Native Associate (KCNA) study guide*. O’Reilly
8. Rocha, Á., Adeli, H., Reis, L. P., Costanzo, S., Orovic, I., & Moreira, F. (2020). *Trends and innovations in information systems and technologies*. Springer
9. Özkan, O., Babur, Ö., & Van Den Brand, M. (2023). *Domain-Driven Design in Software Development: A Systematic literature review on implementation, challenges, and effectiveness*. [Article, Eindhoven University of Technology]. Arxiv
10. Buonanno, E. (2022). *Functional Programming in C#*. Manning Publications.
11. Wlaschin, S. (2018). *Domain modeling made functional: Tackle Software Complexity with Domain-Driven Design and F#.* Pragmatic Bookshelf.
12. Khorikov, V. (2020). *Unit testing principles, practices, and patterns.* Manning.
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, 1985–1994.
16. Pham, A. (2024). *Popular backend Frameworks Performance benchmark comparison and ranking in 2024*. DEV Community. Retrieved from: <https://dev.to/tuananhpham/popular-backend-frameworks-performance-benchmark-1bkh> [accessed: 02 July 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: 12 July 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: 09 June 2024].
19. Sobach,R. (2023). *Top 10 programming languages of 2023 in GitHub report*. AIN.Capital. Retrieved from: https://ain.capital/2023/11/15/top-10-programming-languages-of-2023-in-github-report/ [accessed: 12 July 2024].
20. Luxner, T. (2023). *Cloud computing Stats: Flexera 2023 State of the Cloud Report*. Flexera Blog. Retrieved from: <https://www.flexera.com/blog/cloud/cloud-computing-trends-flexera-2023-state-of-the-cloud-report/> [accessed: 12 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/enus/dotnet/architecture/microservices/> [accessed: 28 June 2024].
22. Vettor, R., & Smith, S. (2023). Architecting cloud native .NET applications for Azure. Microsoft Learn. Retrieved from: <https://learn.microsoft.com/enus/dotnet/architecture/cloud-native/> [accessed: 29 June 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: 16 May 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., & Abrams, B. (2009). *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 & Research in Education/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: 18 July 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 (pp. 632–644).
33. Valiramani, A. (2022). *Microsoft Azure Monitoring & Management: The Definitive Guide*. Microsoft Press.