Journal of Computer and Communications, 2024, 12, 142-155
https://www.scirp.org/journal/jcc
ISSN Online: 2327-5227
ISSN Print: 2327-5219

Transforming Digital Experiences: The Evolution of Digital Experience Platforms (DXPs) from Monoliths to Microservices: A Practical Guide

Sourabh Sethi[1], Sarah Panda[2]

[1]Infosys Limited, New York, NY, USA
[2]Microsoft Inc., Seattle, WA, USA
Email: sourabhsethi@ieee.org

How to cite this paper: Sethi, S. and Panda, S. (2024) Transforming Digital Experiences: The Evolution of Digital Experience Platforms (DXPs) from Monoliths to Microservices: A Practical Guide. *Journal of Computer and Communications*, **12**, 142-155.
https://doi.org/10.4236/jcc.2024.122009

Received: November 13, 2023
Accepted: February 24, 2024
Published: February 27, 2024

Copyright © 2024 by author(s) and Scientific Research Publishing Inc.
This work is licensed under the Creative Commons Attribution-NonCommercial International License (CC BY-NC 4.0).
http://creativecommons.org/licenses/by-nc/4.0/
Open Access

Abstract

The research aims to explore the transition from monolithic Digital Experience Platforms (DXPs) to Microservices-based DXPs, addressing scalability challenges. The study systematically decomposes monolithic structures into Microservices, emphasizing business capability and subdomain decomposition. Concrete insights, challenges, and solutions encountered during this transformation process are presented. The research contributes valuable insights into the challenges and benefits of adopting Microservices in DXPs. Results highlight the importance of architectural patterns and strategic scaling dimensions for improved performance and scalability. The case study on Backbase's Engagement Banking Platform showcases successful implementation, providing flexibility, integration, and efficient development in the evolving DXP landscape.

Keywords

Digital Experience Platforms (DXPs), Microservices, Software Evolution, Distributed Systems, Architectural Patterns

1. Introduction

Digital Experience Platforms (DXPs) have been at the forefront of delivering seamless and engaging user experiences across various touchpoints. However, the conventional approach of monolithic DXPs, where all functionalities originate from a single vendor, presents inherent challenges as applications grow in scale. In this context, scalability becomes a critical concern, leading to costly workarounds and hindrances in expanding applications. The limitations of monolithic DXPs, while suitable for small to medium-scale enterprises with quick time-to-market, become apparent as organizations strive for larger, more scalable solutions. This paper delves into the transformative shift towards Microservices-oriented DXPs, which is driven by the imperative to address scalability challenges. One of the foundational scalability models that inspire the Microservices architecture for DXPs is introduced in the book "*The Art of Scalability*" [1]. This model, known as the scale cube, provides a three-dimensional perspective that guides the scalability considerations for Microservices-oriented DXPs. In the subsequent sections, we will explore the motivations behind adopting Microservices in the context of DXPs, emphasizing the need for a more scalable and flexible architecture. Drawing insights from real-world case studies and industry best practices, this paper aims to provide a comprehensive understanding of the evolution from monolithic to Microservices-based DXPs. Through this exploration, we seek to contribute valuable insights into the challenges, solutions, and benefits associated with this transformative journey.

The scale cube introduces three distinct methods for application scaling, known as X, Y, and Z. X-axis scaling involves distributing the load by balancing requests across multiple instances, achieved through running several instances behind a load balancer (see **Figure 1**). This method, also known as application clustering or replication, is effective in enhancing an application's capability and accessibility. Monolithic DXPs support X-axis scaling. Z-axis scaling, on the other hand, directs requests based on specific attributes, utilizing multiple instances where each is responsible for a distinct subset of data. An orchestrator or router guides requests to suitable instances based on attributes such as UserID.

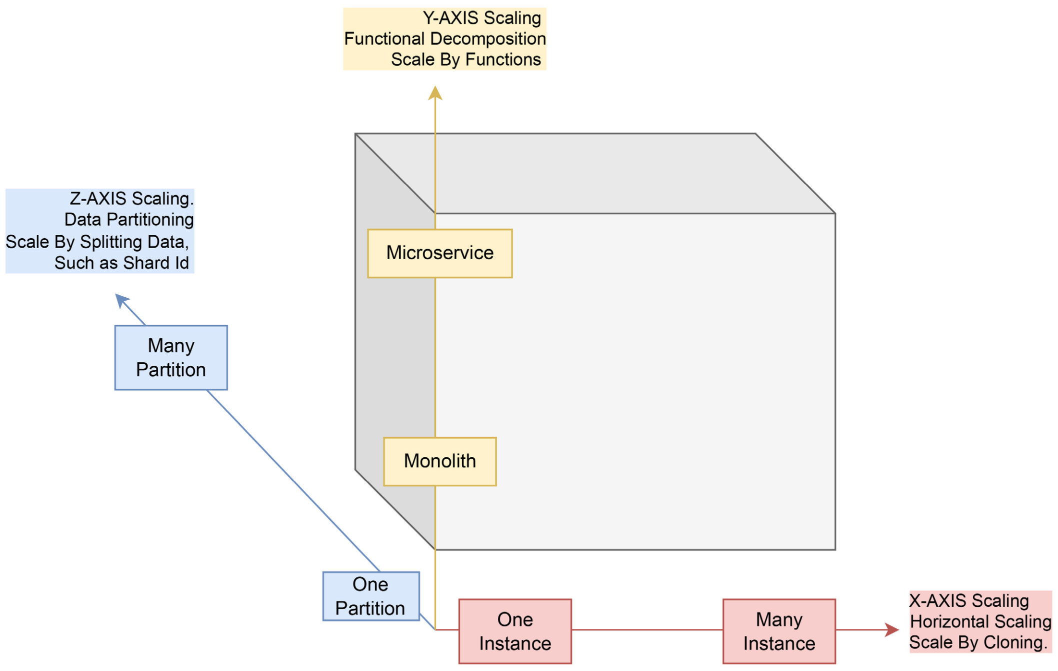

Figure 1. The scale cube.

This scaling method, often referred to as "Partitioning" or "Sharding", is supported by Monolithic DXPs. Y-axis scaling focuses on functionally breaking down an application into services. While X scaling and Z scaling improve capacity and availability, they do not effectively address the challenge of multi-vendor integration and development scalability. Microservices-based DXPs, employing Y-axis scaling or functional decomposition, offer a solution. In this context, a monolithic DXP is partitioned into a collection of services, each with well-defined, focused responsibilities. This approach addresses the need for a more flexible and scalable architecture, particularly in the context of multi-vendor integration and development augmentation. Previous research in Microservice in DXPs has explored the challenges and opportunities associated with this architectural shift. However, a critical analysis reveals certain gaps and limitations: Limited Technical Detail: Earlier studies often provide high-level discussions without delving into the technical intricacies of transitioning DXPs. This gap leaves practitioners with a lack of concrete guidance on the implementation aspects of Microservices in DXPs. Scarcity of Real-world Insights: Many existing works discuss the theoretical advantages of Microservices in DXPs, but there's a shortage of comprehensive real-world case studies that share practical experiences, challenges faced, and solutions implemented during the transition. Overlooking Specific DXP Requirements: The uniqueness of DXPs and their specific requirements, such as multi-vendor integration and scalability challenges, is not always adequately addressed in previous research. A focused examination of these aspects is crucial for practitioners aiming to adopt Microservices in DXPs. Inadequate Consideration of Scalability Models: While scalability is a recurring theme, earlier research often lacks a detailed exploration of scalability models and their application to Microservices in the context of DXPs. A nuanced understanding of scalability is crucial for effective implementation. By identifying these gaps, this research aims to contribute a more detailed and practical understanding of the challenges and solutions associated with transitioning DXPs to Microservices. The subsequent sections will delve into the technical intricacies, provide real-world insights, and address specific DXP requirements, offering a comprehensive guide for practitioners in this evolving landscape.

2. The Evolution from Monolithic Digital Experience Platforms (DXPs) to Microservices-Based DXPs

The transition from Monolithic DXPs to Microservices-based DXPs initiates with a methodical decomposition of the monolithic structure, often referred to as the "big ball of mud". This deconstruction transforms the behemoth into a collection of loosely coupled and cohesive Microservices aligned with business capabilities and subdomains. One approach involves decomposition based on business capabilities, representing core activities that generate business value. These capabilities, specific to the business type, are translated into independent modular services. Structuring services around capabilities offers stability to the architecture, allowing individual components to evolve while maintaining overall architectural consistency. An alternative strategy is sub-domain decomposition, as outlined in Eric Evans' Domain-Driven Design (DDD) [2]. DDD introduces subdomains and bounded contexts, advocating distinct domain models for each subdomain within the enterprise. Subdomains, closely aligned with business capabilities, are delineated by analyzing business operations. Each bounded context corresponds to a service or set of services. This decomposition aligns with the Single Responsibility Principle (SRP) advocated by Robert C. Martin [3]. SRP encourages creating small, cohesive services with a single responsibility, minimizing service size and enhancing stability. The Common Closure Principle (CCP) [4], another principle endorsed by Uncle Bob, suggests grouping components that change for the same reason into the same services, improving maintainability. The combination of SRP, CCP, and decomposition by business capabilities and subdomains proves valuable in transitioning from a monolithic DXP to a Microservices-based DXP. This method ensures an effective and strategic evolution, addressing the challenges posed by the "big ball of mud" and laying the foundation for a modular and scalable DXP architecture. The process of transitioning a monolithic application into Microservices represents a facet of application modernization, which involves converting a legacy application into one with a contemporary architecture and technology stack. Developers have been engaged in application modernization for decades, accumulating wisdom through experience that proves valuable when refactoring an application into a Microservices architecture. An essential lesson learned over the years is the avoidance of a comprehensive rewrite, emphasizing the incremental refactoring of the monolithic application. Instead of opting for a massive rewrite, the recommended approach involves gradually refactoring the monolithic application by building a new application known as a "strangler application". This new application comprises Microservices that operate alongside the existing monolithic application. Over time, the functionality implemented by the monolithic application diminishes until it either completely disappears or transforms into just another Microservices. Three primary strategies are employed for gradually replacing the monolith with Microservices: implementing new features as services, separating the presentation tier and backend, and breaking up the monolith by extracting functionality into services. The first strategy aims to halt the growth of the monolith swiftly, serving as a quick way to showcase the value of Microservices and garner support for the migration effort. The other two strategies focus on dismantling the monolith. While the second strategy may be used occasionally during the monolith refactoring process, the third strategy is crucial as it involves migrating functionality from the monolith into the strangler application. Implementing a new feature as a distinct service is a powerful strategy that prevents the monolith from constraining growth. This approach allows developers to employ modern development techniques, such as Domain-Driven Design (DDD), to create a pristine new domain model. Since the monolith's domain is often vaguely defined and somewhat outdated, the Microservices' domain model may differ significantly in terms of class names, field names, and field values. Due to these differences, the implementation of an Anti-Corruption Layer (ACL), as per DDD terminology, becomes necessary to facilitate communication between the service and the monolith. The ACL acts as a protective layer of code, ensuring that the legacy monolith's domain model does not contaminate the service's domain model by serving as a translator between the two distinct domain models.

3. Challenges in Microservices

Implementing a Microservices-based Digital Experience Platform (DXP) holds the promise of enhanced scalability and agility. However, beneath the surface simplicity of decomposing services based on business capabilities or subdomains lies a set of intricate challenges inherent to the distributed nature of the system. This study collected responses from a pool of 500 individuals who either currently utilize or plan to adopt Microservices. Participants in the survey held key roles in software development and architecture, operating within companies with a workforce exceeding 500 employees. The distribution of the sample was nearly even among those actively using Microservices in production, those in the pilot phase, and those in the planning stage. Concerns regarding the repercussions of Microservices were notable among current users. For instance, 59 percent of respondents employing Microservices in production acknowledged heightened operational challenges, particularly in data management. Similarly, 58 percent reported a substantial surge in application data generation. The second most prevalent challenge, cited by 56 percent, pertained to identifying the root cause of performance issues. In terms of troubleshooting, 73 percent found Microservices more challenging, while only 21 percent deemed them easier compared to monolithic architectures. The key takeaway from the research is that while Microservices offer solutions to certain issues, they also introduce new challenges, particularly for those leaning towards the operational aspects of the DevOps spectrum. Interestingly, respondents exhibited a recent enthusiasm for Microservices. Considering those testing Microservices but not yet deploying them, 36 percent of the sample initiated Microservices adoption within the last year. Additionally, when asked about the anticipated default architecture for their development teams, 16 percent asserted that Microservices already holds that status, while another 19 percent predicted it would be the default by the year's end. Only a marginal 2 percent believed Microservices would never become the default. Despite concerns, individuals with Microservices in production expressed satisfaction, with 63 percent attesting to the success of Microservices in their contexts.

3.1. Network Latency and Synchronized Communication

The distributed nature of Microservices introduces network latency, potentially leading to diminished availability due to synchronized communication. Addressing inter-service communication without compromising availability is a key challenge. The adoption of asynchronous messaging emerges as a favorable choice, eliminating tight coupling and enhancing overall system availability.

3.2. Data Consistency across Services

Maintaining data consistency across services, especially when certain operations necessitate updates across multiple services, presents a significant hurdle. The conventional two-phase commit-based distributed transaction management mechanism may not be well-suited for modern applications. A "saga" approach, involving a sequence of local transactions coordinated through messaging, becomes essential. Sagas offer advantages but are more intricate than traditional ACID transactions and may not provide immediate consistency. In the saga pattern, a distributed transaction is broken down into a sequence of smaller, localized transactions, often referred to as "saga steps". Each saga step represents a distinct operation within the overall business process. These steps are executed in an orchestrated and coordinated manner, and they are designed to be idempotent, meaning that they can be safely retried without causing unintended side effects. The key characteristics and principles of the saga pattern: Local Transactions: Each Microservices involved in the saga performs its part of the transaction as a local transaction. These local transactions are typically database transactions within the Microservices' boundaries. Choreography or Orchestration: The saga pattern can be implemented through choreography or orchestration. In choreography, each service is responsible for deciding what actions to take based on the events it observes. In orchestration, there is a central component (orchestrator) that coordinates the execution of saga steps. Compensation: If a failure occurs during the execution of a saga step, a compensating transaction is triggered to undo the effects of the preceding steps. Compensation logic is designed to bring the system back to a consistent state. Event-Driven: The saga pattern often relies on event-driven communication between Microservices. Each step emits events, and other Microservices react to these events to perform their part of the transaction. Asynchronous: Saga steps are often executed asynchronously, which can help improve system responsiveness and reliability. Partial Success: In scenarios where some saga steps succeed while others fail, the system can still achieve a consistent state by executing compensating transactions for the failed steps.

3.3. Consistent Data View across Multiple Databases

Obtaining a truly consistent data view across multiple databases in a Microservices-based DXP is challenging. While each service's database may exhibit consistency, achieving a globally consistent data view becomes infeasible. If the need for a consistent data view arises, it must be confined to a single service, potentially hindering the decomposition process.

3.4. "God Classes" as an Obstacle to Decomposition

"God classes", oversized classes encapsulating business logic for various aspects of the application, pose a formidable challenge when attempting to disassemble business logic into services. Embracing Domain-Driven Design (DDD) principles becomes essential (Table 1). Treating each service as an autonomous sub-domain with its unique domain model helps eradicate "god classes" and promotes a more effective decomposition strategy.

4. Microservices Architecture Patterns

The Microservices architecture pattern language comprises a set of patterns strategically designed to guide the architectural decisions when implementing an application using the Microservices architecture. It is organized into various pattern groups, each serving a specific purpose. Initially, the pattern language assists in the determination of whether the Microservices architecture is the suitable choice. Subsequently, it offers pattern groups that function as solutions to challenges arising from the adoption of the Microservices architecture pattern.

These patterns are further categorized into three layers: Infrastructure patterns, addressing primarily infrastructure issues beyond the development scope; Application infrastructure patterns, dealing with issues that impact both infrastructure and development; and Application patterns, providing solutions to challenges faced by developers. The grouping is based on the nature of the problems these patterns address. Architectural decisions play a pivotal role in the evolution of Digital Experience Platforms (DXPs). Choosing between a monolithic or Microservices architecture requires a careful evaluation of pros, cons, and a consideration of numerous trade-offs. Opting for a Microservices architecture introduces challenges inherent in its distributed nature. In this context, architectural patterns emerge as valuable tools—reusable solutions rooted in real-world architectural concepts.

Table 1. Challenge and solution.

Challenge	Description	Impact	Mitigation
Network Latency and Synchronized Communication	Microservices communication over a network can introduce latency, impacting performance, especially in synchronous communication.	Slower response times, potential service bottlenecks.	Adopt asynchronous communication patterns, use message queues, or implement event-driven architectures to reduce the impact of network latency.
Data Consistency across Services	Maintaining data consistency across multiple Microservices becomes complex, especially when updates involve multiple services.	Inconsistencies in data, errors, and lack of integrity.	Implement compensating transactions use the Saga pattern for coordinated local transactions through messaging to manage data consistency.
Consistent Data View across Multiple Databases	Achieving a globally consistent data view is challenging as each Microservices typically has its own database.	Challenges in maintaining synchronized data across the entire system.	Design services with eventual consistency in mind consider patterns like CQRS for managing consistent data views.
"God Classes" as an Obstacle to Decomposition	Legacy monolithic applications often contain oversized classes, hindering the decomposition into small, independent Microservices.	Difficulty in breaking down large, complex classes for modularization.	Adopt Domain-Driven Design (DDD) principles, treat each service as an autonomous sub-domain, and use unique domain models to break down "God Classes".

Patterns are essential because they are context-specific solutions, acknowledging the diversity of applications. Tailored to particular contexts, they advance technology discussions, recognizing that a solution designed for giants like BOFA or Amazon might not universally fit smaller user-base applications. A well-structured pattern typically consists of Forces, Resulting Context, and related patterns [5].

The Microservices architecture pattern language serves as a roadmap for decision-making, assisting in evaluating the suitability of Microservices architecture. It delineates the attributes, benefits, and limitations of both monolithic and Microservices architectures. Should Microservices be suitable? The pattern language aids in effective implementation, offering solutions to various challenges. It is organized into Infrastructure Patterns, Application Infrastructure Patterns, and Application Patterns.

4.1. Patterns for System Decomposition into Services

Breaking down a system into services is an art, and two distinct strategies are highlighted—the "Decompose by Business Capabilities" pattern and the "Decompose by Subdomain" pattern. These provide guidance in defining the application's architecture.

4.2. Communication Strategies

Microservices architecture, operating as a distributed system, necessitates thoughtful communication strategies. These are categorized into five groups: Communication Style, External API, Discovery, Reliability, and Transactional Messaging.

4.3. Patterns for Data Retrieval in Microservices Architecture

Accessing data across multiple services when using dedicated databases presents challenges. Patterns like API Composition and Command Query Responsibility Segregation (CQRS) offer solutions to overcome these challenges.

4.4. Patterns for Enforcing Data Consistency in Transaction Management

Loose coupling with individual databases per service introduces challenges, making traditional distributed transactions impractical. The Saga pattern becomes essential for contemporary applications to uphold data consistency.

4.5. Observability Patterns for Insight into Application Behavior

Managing the runtime behavior of Microservices requires effective observability. Patterns like Health Check API, Log Aggregation, Distributed Tracing, Exception Tracking, Application Metrics, and Audit Logging facilitate understanding and troubleshooting in this complex environment.

In conclusion, the transition from Monolithic DXPs to Microservices DXPs demands informed decision-making, and architectural patterns provide a structured approach. These patterns empower architects and developers to make informed choices aligned with the unique needs of their applications, fostering a successful evolution in the DXP landscape.

5. Performance Optimization in Microservices

The performance of Microservices is intricately tied to various factors, encompassing Interprocess Communication (IPC), message patterns, caching strategies, choices between synchronous and asynchronous communication, and the selection between SQL and NoSQL databases. There exists no one-size-fits-all solution for all enterprise applications. Instead, general principles, as elucidated by the CAP and PACELC Theorems in distributed environments, highlight the necessity of trade-offs between consistency and availability in the presence of partition tolerance. Conversely, when partition tolerance is not a concern, trade-offs between latency and consistency come to the forefront. The industry commonly adheres to the following guidelines:

REST APIs for External Communication: Utilizing REST APIs for external communication is essential, given their widespread adoption as a standard across diverse platforms in the industry. To enhance efficiency, a recommended approach is to implement a query language on the API, such as GraphQL. This implementation helps prevent the unnecessary retrieval of extra fields or data by client applications.

Caching Strategies: Caching can play a pivotal role in enhancing application performance. However, before implementing caching, a thorough assessment of the specific use case is essential. Determining the type of caching that best fits, whether it's local, global, or a distributed caching solution, is crucial. Additionally, careful consideration should be given to the invalidation and eviction policy.

Binary-Based Messaging Formats for Internal Communication: When dealing with internal services within a Microservices architecture, the choice of communication format can impact availability and performance. Instead of using REST for inter-service communication, it is advisable to opt for binary-based messaging formats like Protocol Buffers. Solutions such as gRPC, built on top of Protocol Buffers, facilitate reduced message size during interoperability between services, consequently enhancing the overall system's performance.

Asynchronous Communication with Message Queues: Handling messages in a Microservices architecture using synchronous communication methods like REST or RPC can noticeably reduce the system's availability. In contrast, advanced systems are designed with decoupled components using message queues and brokers. This approach greatly enhances system performance, leading to a more sophisticated system that operates on an event-driven architecture.

6. Securing in Microservices

The performance of Microservices is intricately influenced by several factors, encompassing Interprocess Communication (IPC), message patterns, caching solutions, choices between synchronous and asynchronous communication, and the selection between SQL or NoSQL databases. A key consideration in distributed environments is the trade-off between consistency and availability, especially when dealing with partition tolerance, as outlined by the CAP and PACELC Theorems. Alternatively, in scenarios where partition tolerance is less critical, trade-offs between latency and consistency become pivotal. The industry commonly follows these general guidelines:

External Communication: Utilize REST APIs for external communication, as they represent a widely adopted standard across various industry platforms. To enhance efficiency, consider implementing a query language on the API, such as GraphQL, which helps prevent the unnecessary retrieval of extra fields or data by client applications.

Caching: Caching can significantly boost application performance. However, before implementation, carefully assess the specific use case and determine the type of caching that best fits—whether it's local, global, or a distributed caching solution. Additionally, thoughtful consideration should be given to the invalidation and eviction policy.

Internal Service Communication: When facilitating communication between internal services within a Microservices architecture, consider the impact on availability and performance. Instead of relying on REST for inter-service communication, it is advisable to opt for binary-based messaging formats like Protocol Buffers. Solutions such as gRPC, built on top of Protocol Buffers, contribute to reducing message size during interoperability between services, thereby enhancing the overall system's performance.

Message Handling: Handling messages in a Microservices architecture using synchronous communication methods like REST or RPC can noticeably reduce the system's availability. In contrast, advanced systems are designed with decoupled components using message queues and brokers. This approach greatly enhances system performance, leading to a more sophisticated system operating on an event-driven architecture. These guidelines underscore the importance of thoughtful decisions in various aspects of Microservices design, emphasizing the need to align choices with specific use cases and the overarching goals of the system.

7. Security in Microservices

Microservices effectively implementing authentication and authorization can pose significant challenges. It is recommended to utilize a reputable security framework, with the choice of the framework depending on the technology stack of your application. Some popular frameworks include Spring Security, Apache Shiro, Passport, and others [6]. In a Microservices architecture, user authentication is often managed by the API gateway. Subsequently, the API gateway needs to transmit user-related information, including identity and roles, to the services it interacts with. A proven approach to address this challenge is to leverage the

Access Token pattern. This involves the API gateway dispatching an access token, such as a JSON Web Token (JWT), to the services. These services can then validate the token and extract relevant user details.

Within a Microservices architecture, all external requests are initially processed by the API gateway, which then proceeds to relay the request to one or more services. For example, when handling a query, the API gateway might activate multiple services such as Payment Service, Plan Service, User Service, and Accounting Service. Each of these services needs to consider various security aspects. For instance, the User Service must restrict consumers to viewing their own plans, requiring a combination of authentication and authorization. To implement security effectively in a Microservices architecture, it's crucial to determine who is responsible for authenticating users and who is responsible for authorizing their actions. The preferred approach is to have the API gateway authenticate a request before it's relayed to the services. Centralizing API authentication in the API gateway provides the advantage of having a single focal point for ensuring security. Consequently, this reduces the likelihood of security vulnerabilities. Additionally, this approach alleviates the burden of managing various authentication mechanisms for the other services, as it abstracts this complexity from them. JWT, or JSON Web Token, is a standardized approach for securely conveying assertions, including details like user identity and roles, between two entities. A JWT consists of a payload, which is a JSON object containing user-specific information, such as identity, roles, and additional metadata like an expiration date. It is cryptographically signed with a secret known exclusively to the JWT creator, such as the API gateway, and the recipient of the JWT, like a service. This secret acts as a safeguard, preventing malicious third parties from counterfeiting or tampering with the JWT.

8. Scalability in Microservices

The Microservices concept was introduced as a solution to address scalability challenges within large organizations. It involves implementing a modular team structure, where team sizes are limited to those that can be fed with just two pizzas. This approach aims to enhance the scalability and efficiency of application management. In this model, each team is responsible for managing a set of services, and these services function as self-contained web servers. To further improve availability and scalability, embracing a cloud-native application approach is essential. This approach aligns with the principles of cloud computing, allowing organizations to leverage scalable and flexible cloud infrastructure. Building applications with a cloud-native mindset enhances their resilience and responsiveness to varying workloads.

Having previously explored the concept of scaling along the X, Y, and Z axes, organizations can efficiently utilize these scaling dimensions to enhance application performance. Scaling along the X axis involves adding more instances of the same service, scaling along the Y axis involves distributing services across different servers, and scaling along the Z axis involves scaling services independently based on specific attributes.

By strategically employing these scaling dimensions, organizations can tailor their approach to meet the unique scalability requirements of their applications. This enables efficient resource utilization, improved performance, and the ability to adapt to changing demands within the dynamic landscape of modern software development.

9. Microservices-Based DXPs Case Study

Backbase has introduced the Engagement Banking Platform, an open platform designed to facilitate the rapid modernization of banking operations [7]. According to Backbase and our research findings, this digital experience platform provides an opportunity to break free from vendor lock-in and legacy systems. It offers a genuinely open digital banking platform built using Microservices, presenting an alternative to the conventional "build or buy" dilemma and transcending the confines of traditional platform monolith models. With this platform, organizations gain the flexibility to acquire solutions for speed and build custom differentiators, enabling them to swiftly bring unique offerings to market. The platform seamlessly integrates with existing heterogeneous technology landscapes, allowing the continued utilization of various programming languages and technology frameworks. It empowers organizations to tap into thriving ecosystems and communities surrounding popular front-end technologies like Angular, React, Flutter, Vue.js, Swift, and Kotlin. Leveraging micro-frontends and module federation, the polyglot architecture enables organizations to access these ecosystems and harness a wide array of tech-specific libraries, tools, and community support. By minimizing the need for extensive rework and reducing complexity, the platform facilitates rapid integration. The adoption of a containerized approach for component deployment supports system integration, data sharing, and the development of bespoke Microservices. This approach harnesses the strengths of different frameworks, enhancing developer productivity, optimizing performance, and promoting flexible, modular development. It ensures smooth interoperability across various infrastructural components in a diverse polyglot environment.

Recognizing that there is no one-size-fits-all solution, organizations have the flexibility to choose the most suitable persistence layers and runtime engines to align with specific banking requirements. Options include document-based storage, non-relational databases, or distributed key-value systems. The platform also accommodates event-based communication and the integration of serverless functions for highly scalable microtasks.

The Backbase platform serves as the foundation for digital transformation, acting as the "construction site" where organizations collaborate with cross-functional teams dedicated to business transformation initiatives. These teams leverage the same industrialized platform capabilities, encompassing reusable building blocks and repeatable processes. They use these capabilities to create tailored customer journeys and value propositions for key client segments. The key to the Digital Factory's success lies in small, agile teams closely aligned with the business side, functioning as accelerators to drive the rapid modernization of customer journeys within the banking sector using Microservices architecture [8].

10. Conclusion

Digital transformation is causing disruptions across industries, organizations, and processes. A growing number of organizations are actively embracing digital transformation initiatives to modernize their processes, cut costs, enhance user experiences, gain competitiveness, foster innovation and efficiency, and achieve greater agility. The shift towards Microservices plays a crucial role in fostering agility, enabling the adoption of DevOps practices, embracing lean organizational structures, and facilitating rapid digital transformation.

Acknowledgements

The authors would like to express their gratitude to Dr. Shailesh Kumar Shiva Kumar for his extensive research in Digital Experience Platforms and in the field of Digital Portals. Additionally, the authors extend their thanks to all the developers who contributed to completing case studies on different Backbase and other DXPs available across the market.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Abbott, M.L. and Fisher, M.T. (2015) The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise. Addison-Wesley Professional, New York.

[2] Evans, E. (2004) Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional, New York.

[3] Martin, R.C. (2009) Clean Code: A Handbook of Agile Software Craftsmanship. Pearson Education, Upper Saddle River, NJ.

[4] Martin, R.C. (2017) Clean Architecture. Pearson Education, Upper Saddle River, NJ.

[5] Richardson, C. (2018) Microservices Patterns: With Examples in Java. Simon and Schuster, New York.

[6] Shivakumar, S.K. and Sethii, S. (2019) Building Digital Experience Platforms: A Guide to Developing Next-Generation Enterprise Applications. Apress, New York, NY. https://doi.org/10.1007/978-1-4842-4303-9

[7] Sethi, S. and Shivakumar, S.K. (2023) DXPs Digital Experience Platforms Transforming Fintech Applications: Revolutionizing Customer Engagement and Financial Services. *International Journal of Advance Research, Ideas and Innovations in Tech-*

nology, **9**, 419-423. https://www.ijariit.com/

[8] The Engagement Banking Platform. https://www.backbase.com/engagement-banking-platform#break-free-from-vendor-lock-in-&-tech-debt