HONEYCOMB MONOLITH: HEXAGONAL MODULAR PATTERN FOR AGILE MICROSERVICES EVOLUTION

СТІЛЬНИКОВИЙ МОНОЛІТ: ШЕСТИКУТНИЙ МОДУЛЬНИЙ ПАТТЕРН ДЛЯ ГНУЧКОЇ ЕВОЛЮЦІЇ В МІКРОСЕРВІСИ

by Taras Shablii

Presented in Partial Fulfillment of the Requirements of the Degree

Master of Software Engineering

American University Kyiv

2024

APPROVED BY:

Sergiy Tytenko, Ph.D., Faculty Mentor

Abstract

This thesis explores the architectural dilemma faced by startups and greenfield projects: choosing between monolithic and microservices structures. It addresses the gap in research on evolutionary monolithic architectures, introducing the Honeycomb Monolith pattern. This pattern combines Domain-Driven Design with Hexagonal Architecture to create modular monoliths poised for smooth transition to microservices.

The effectiveness of the Honeycomb Monolith is demonstrated through the Opora application case study. This implementation validates the pattern viability, showing a seamless migration with minimal impact on the core domain logic. Challenges like model duplication and database management complexities are also identified, underscoring the need for strategic planning in architecture design.

Concluding with future research directions, the thesis positions the Honeycomb Monolith as a viable solution for startups and an intermediary step for existing projects transitioning to microservices. This work contributes to the software architecture field, offering a novel solution that balances initial development efficiency with long-term scalability.

Table of Contents

Introduction 3
Chapter 1: Modular Monolith as a Microservices Precursor 4
1.1. Monoliths vs. Microservices 4
1.1.1. The Migration Trend: Monoliths to Microservices 7
1.1.2. Challenges of Migrating Monoliths to Microservices 8
1.2. Modular Monolith: Problem Statement 9
1.3. Evolutionary Monolith Research Gap 11
1.4. Summary 11
Chapter 2: Concept of Honeycomb Monolith Pattern 13
2.1. Domain-Driven Design as Foundation for Modular Breakdown 14
2.1.1. Defining Domain-Driven Design 14
2.1.2. DDD Use in Microservices Design 14
2.2. Hexagonal Architecture at the Heart of Module Internal Structure 15
2.2.1. Principles of Hexagonal Architecture 15
2.2.2. Benefits in Software Design 16
2.2.3. Lightweight Approach in Honeycomb Monolith 17
2.3. Honeycomb Monolith Pattern Description 19
2.3.1. Structural Details of the Honeycomb Monolith Pattern 19
2.3.2. Benefits of the Honeycomb Monolith 22
2.3.3. Drawbacks of the Honeycomb Monolith 24
2.4. Summary 25
Chapter 3: Implementing Honeycomb Monolith 26
3.1. Opora API Implementation Context 26
3.2. Opora Application Monolith Design 27
3.2.1. Domain-Driven Modular Design 27
3.2.2. Module Hexagonal Structure 30
3.2.3. Honeycomb Monolith Architecture Enforcement 34
3.3. Opora Application Migration to Microservices 35
3.3.1. Migration Planning and Steps 35
3.3.2. Axis of Change 37
3.4. Honeycomb Monolith Implementation and Migration Results 39
3.5. Honeycomb Monolith Applicability in Different Contexts 40
3.6. Summary 41
Conclusions 42
References 43

Introduction

At the heart of modern software development lies a crucial architectural dilemma: the choice between monolithic and microservices architectural styles. This thesis explores this pivotal decision, particularly in the context of the migration trend and associated challenges.

The trend towards microservices has been gathering momentum, driven by its promises of availability, scalability, and independent deployment. However, this transition from traditional monolithic applications is a risky endeavor, fraught with complexities and pains. The challenge intensifies for greenfield projects and startups at the crossroads, contemplating whether to begin with a monolithic architecture, which may necessitate a painful migration later, or to adopt a microservices approach from the outset, a pricey strategy that comes with its own challenges.

While there exists a wealth of research and literature on microservices architecture and the migration of brownfield monolithic applications to microservices, a notable gap persists in the study of structuring greenfield monolithic applications. Specifically, there is a lack of research on evolutionary monolith architecture, poised for a seamless transition to microservices as and when the business needs demand.

This thesis introduces the Honeycomb Monolith pattern as a proposed solution to this conundrum. The pattern innovatively combines the principles of Domain-Driven Design with the adaptability of Hexagonal Architecture to create a monolithic structure that is inherently evolutionary. This approach addresses the core issue faced by startups and new projects, offering a structured path that balances the immediate benefits of a monolith with the long-term advantages of microservices.

Central to this research is a pragmatic case study demonstrating the implementation of the Honeycomb Monolith pattern and the execution of migration to microservices. This real-world application validates the proposed solution, offers insights, and provides a replicable model.

This thesis aims to spark a broader conversation about effective monolithic structures in software architecture. By bridging the current research gap and providing a tangible, tested architectural pattern, this work contributes to the ongoing dialogue about how best to navigate the architectural landscape, balancing immediate needs with future scalability and adaptability.

Chapter 1: Modular Monolith as a Microservices Precursor

Chapter 1 investigates the ongoing discourse, exploring the nuances, challenges, and implications of the two predominant architectural styles: monoliths and microservices. This chapter begins by unraveling the core attributes of monolithic and microservices architectures. We examine the ongoing migration trend and the underlying benefits that it offers. This transition, while promising enhanced scalability and agility, is not without its complexities and pitfalls.

We further focus on the challenges that surface when migrating from monoliths to microservices. These challenges involve intricate domain and codebase refactoring, along with considerations for infrastructure and data management. In addressing these challenges, we explore the concept of modular monoliths as a potential solution and an intermediary step. This approach advocates for structuring monoliths in a manner that each module represents a separate domain or subdomain, paving a smoother path towards microservices evolution.

However, such modular monolith structure is underexplored in current research, signaling a gap in scientific understanding and systematic study. This chapter highlights the need for more research and development guidelines on the topic of evolutionary monolith architecture.

1.1. Monoliths vs. Microservices

In the world of software architecture, two dominant styles have emerged, each with its set of advantages and disadvantages: monoliths and microservices. Monolith is a "*system in which all of the code is deployed as a single process*" [1]. In this architectural style all code resides in a single repository and forms a single deployment artifact, all data is most often located within a single database (Figure 1).

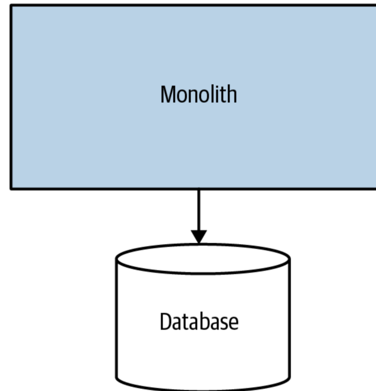

Figure 1: Single process monolith [1]

Monoliths have been the traditional choice for software development for decades. In a monolithic architecture, the entire application, from user interface to database access, is bundled together in a single codebase. This tightly coupled nature simplifies development but introduces challenges in terms of scalability and maintainability [4].

Monolithic architecture provides the following benefits [5]: simplicity of development and deployment since all code is located within a single codebase and is deployed as a single artifact; optimal performance, since components communicate via direct function calls without any network overhead; faster time to market since developing and deploying a single codebase that does not require sophisticated infrastructure. At the same time, monolithic architecture is characterized by natural degradation of scalability (increasing capacity often involves scaling the entire application, which can be inefficient) and maintainability (monoliths tend to become increasingly complex and harder to maintain) [5].

In contrast, a microservice (Figure 2) is a "*single-purpose, separately deployed unit of software that does one thing really, really well*" [2].

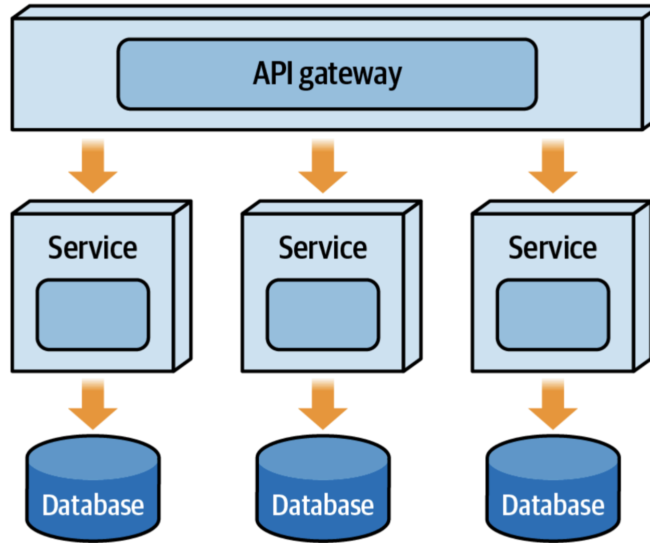

Figure 2: Microservices topology [2]

Microservices, in contrast, advocate for breaking down an application into smaller, autonomous services, each with a specific singular responsibility. These services communicate with each other over the network, allowing for flexibility and independent scalability [5].

The heightened interest in microservices over the past decade can be attributed to several key factors. First, there has been a growing demand for applications with enhanced availability, fault tolerance, and resilience, especially as businesses rely more on digital services [6]. Microservices provide a framework for achieving these goals by allowing individual services to fail gracefully without compromising the entire system's functionality. Second, the rapidly evolving technological landscape has introduced greater complexity, making it challenging to develop and maintain monolithic applications efficiently. Microservices offer a solution by breaking down complex systems into smaller, manageable components that can be developed, deployed, and scaled independently [7]. Third, the surge in data volumes, driven by the advancements of data-driven applications and IoT devices, has necessitated scalable and distributed architectures. Microservices' ability to scale horizontally makes them well-suited for handling data-intensive workloads [6]. Lastly, advancements in IaaS cloud technology have made it easier to adopt microservices, with cloud providers offering robust infrastructure, platforms and services that facilitate the deployment and management of microservices. These factors have collectively fueled the rising interest in microservices as a modern software architecture paradigm [7].

Benefits of microservices include [5]: scalability since each microservice can be scaled independently, enabling cost-effective resource allocation; smaller codebases are more maintainable, reducing the risk of system-wide disruptions; microservices are

fault-tolerant and can be easily made highly available, as the failure of one service doesn't necessarily affect the entire application.

At the same time, microservices architecture is complex, as developing and managing microservices can be complex due to the need for inter-service communication and infrastructure management; it brings added overhead in terms of communication, deployment, and monitoring; and introduces additional layers of communication between services which negatively affects overall system performance due to network latencies [4].

1.1.1. The Migration Trend: Monoliths to Microservices

While both approaches have their merits, the interest for microservices has been on the rise in the recent decade (Figure 3).

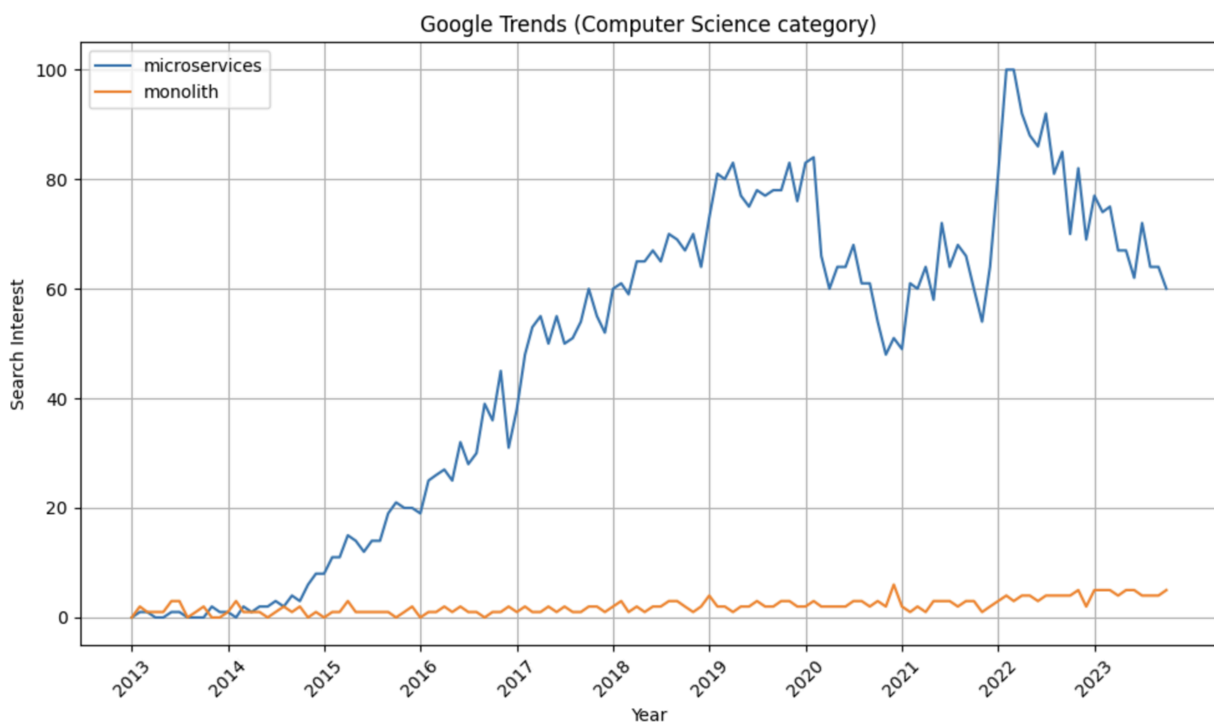

Figure 3: Raising interest in microservices architecture [3]

In recent years, many organizations have embarked on the journey of migrating from monoliths to microservices. Several compelling reasons drive this migration [7][8]:

1. Scalability: Monoliths often struggle to scale efficiently, as any increase in load affects the entire application. Microservices allow organizations to scale only the parts of the application that require additional resources. The entire system can scale up or down in response to demand changes.

2. Technology diversity: Microservices support a wider range of technologies for different components of the application. This enables organizations to choose the best tool for each job, enhancing overall performance.
3. Distributed team autonomy: In a monolithic environment, changes to one part of the application can affect other components. Microservices promote team autonomy, as technologically diverse teams can work on separate services with minimal interference.
4. Faster deployment: Microservices enable independent deployment, allowing organizations to release updates and new features more rapidly.
5. Resilience: Microservices inherently offer better fault isolation. If one service fails, it doesn't necessarily lead to the entire application's collapse.

1.1.2. Challenges of Migrating Monoliths to Microservices

While the benefits of microservices are clear, the journey from monoliths to microservices is fraught with challenges [9]. These challenges encompass both domain and codebase refactoring, as well as infrastructure and data-related hurdles [4]. Moreover, the cost of migration, both in terms of time and resources, can be substantial [10].

Domain and Codebase Refactoring

Refactoring a monolith into microservices requires tearing the application into smaller bounded contexts that encompass specific business domains. This process involves identifying boundaries, separating concerns, and establishing clear APIs for inter-service communication.

These refactoring efforts often face the following common challenges:

- *Boundary identification*: Determining the right boundaries for microservices can be complex, and mistakes can lead to overly chatty communication between services or services that are too tightly coupled.
- *Data separation*: Monoliths often share a common database. Refactoring requires separating the often tightly coupled data into distinct databases for each microservice, which can be intricate for some systems [4].
- *API design*: Designing clear APIs for microservices is a critical task, as these APIs serve as a domain separation boundary and enable inter-service communication.

While there is plenty of research on the attempts to automate or semi-automate monolith decomposition by static code analysis [11] or by analyzing data flow [12], the prospect remains a daunting one [13].

Infrastructure and Data Challenges

In addition to domain and codebase refactoring, the migration from monolith to microservices entails addressing infrastructure and data-related challenges [14]:

- *Deployment and orchestration*: Microservices require robust CI/CD practices to manage the deployment of multiple services across a distributed environment.
- *Data consistency*: Maintaining data consistency and ensuring data availability in a microservices environment is a complex task, particularly when multiple services access the same data or when there is a need to implement more complex data replication or CQRS patterns.
- *Monitoring and observability*: With the increase in the number of services, monitoring and troubleshooting become more challenging. Each service needs to be individually monitored and each service call needs to be traced as it propagates through a distributed system with varying availability.
- *Communication and networking*: Microservices rely heavily on network communication. Designing and managing network traffic and security is vital.
- *Infrastructure cost*: The distributed nature of microservices can lead to increased infrastructure costs, as additional resources are required to maintain the architecture.

The cost of migration should not be underestimated. It encompasses not only the time and effort required for refactoring but also the effect on organization communication patterns and potential business disruptions during the migration process [4].

1.2. Modular Monolith: Problem Statement

Despite their shortcomings, monoliths remain an attractive architectural choice for many projects due to their speed of development, shorter time to market, infrastructure relative simplicity, and performance benefits. Not every greenfield project should start from microservices. In fact, many falling prey to the trend find themselves not being able to deliver on time. So, the question to be asked is: what architecture is the best fit for a greenfield project that needs fast time to market at first and is very likely to scale

exponentially later? Or rather: are there any options other than painful monolith to microservices migration or unnecessary and expensive microservices from the start?

One way to alleviate the challenges associated with migrating from monoliths to microservices is to consider the concept of modular monoliths. A modular monolith is a monolithic application that is structured in a way where each module already represents a separate domain or subdomain. This modular structure mimics the decomposition of a microservices architecture within a monolithic codebase. While modularizing a monolith is a challenging endeavor [15] it will certainly pay off when a need to scale arises and the organization decides on migration to microservices. In this respect, modularizing a monolith from the start can be viewed as earning architectural credit early on.

A modular monolith brings:

- *Fast time to market*: Inherent to monolithic architecture ease of development, simplicity of deployment, and lack of interservice communication overhead mean that greenfield projects can freely experiment and deliver functionality faster than with microservices.
- *Clear domain boundaries*: A well-structured modular monolith already exhibits distinct domain boundaries, making the transition to microservices more straightforward.
- *Loose coupling*: The modular structure reduces interdependencies between modules, making it easier to extract and deploy them as separate services.
- *Infrastructure simplicity*: Modular monolith inherits simplicity of deployment and required infrastructure from the monolith by the virtue of being a single deployable artifact.
- *Incremental migration*: Organizations can gradually migrate modules to microservices, reducing the disruption and risk associated with a complete migration.
- *Performance*: Direct method calls within a singular deployment artifact bring a natural performance optimization as opposed to a distributed system.

However, it's important to note that achieving a well-structured modular monolith is not a trivial task and requires careful planning and design. A clear understanding of the business domains and subdomains, along with effective API design, is crucial for success. Despite structural complexity required from the start, greenfield projects are likely to adopt a modular monolith approach as part of addressing the tech debt [16].

1.3. Evolutionary Monolith Research Gap

Despite the increasing popularity of microservices and the challenges associated with migrating from monoliths to microservices, there is a noticeable scarcity of research in the domain of structuring a monolith to facilitate a smooth transition. Among the few research attempts in this area, a bachelor's thesis by Tsechelidis should be noted [17]. The paper proposes to standardize monolith module structure by leveraging hexagonal architecture as an easy way to control granularity of services and promote microservice-like domain-centric design of each module [17]. Author emphasizes on the benefits of infrastructure simplicity but does not extend an argument for further splitting of such modular monoliths into microservices.

Apart from the lack of scientific research in the area there are conflicting opinions from the industry voices. Fowler is convinced that "*you shouldn't start a new project with microservices, even if you're sure your application will be big enough to make it worthwhile*" [18]. Newman notes that it is more practical to start with a monolithic system, warns against the pitfall of introducing unnecessary complexity overhead, but does not rule out using microservices for greenfield projects altogether [19]. Tilkov, on the other hand, argues that carving the new system into pieces should be done as early as possible and going with microservices first is the right way to achieve that [20].

While industry best practices and case studies provide valuable insights, there is a need for systematic and scientifically grounded approaches. Research question that remains unexplored is: what are the best practices for designing modular monoliths that are more amenable to migration to microservices? What design patterns and architectural principles should be applied?

1.4. Summary

The architectural choices in software development are pivotal, and the transition from monoliths to microservices is a significant endeavor for many organizations. While the benefits of microservices are clear, the migration process is challenging, both in terms of domain and codebase refactoring, as well as infrastructure and data considerations. The concept of modular monoliths, where each module already represents a separate domain or subdomain, offers a potential solution to ease this transition.

However, it's important to note that the landscape of structuring monoliths for microservices migration is under-researched. More scientific inquiry and systematic studies are needed to develop best practices, tools, and frameworks that can guide organizations in this transition. By closing the research gap, we can make the path from monoliths to microservices more predictable and efficient. This will allow organizations

to harness advantages of both architectural styles, alleviate the fear of being “stuck” with the monolith, and enable easier transitions from monoliths to microservices when the need to scale arises.

We propose the Honeycomb Monolith pattern to address this very problem. In the next chapter we will describe how this pattern combines Domain-Driven Design with Hexagonal Architecture in a novel way to forge an agile monolithic structure ready for evolution into microservices.

Chapter 2: Concept of Honeycomb Monolith Pattern

As outlined in Chapter 1 (also published as stand-alone article [21]), there is no research that addresses the problem of modeling monolithic applications for smooth transition into microservices. In this chapter we offer one such solution and introduce the "Honeycomb Monolith" pattern. At the heart of this pattern lies the integration of two pivotal concepts in software architecture: Domain-Driven Design and Hexagonal Architecture. This combination harnesses the strengths of both approaches creating an efficient pathway from monolithic applications to microservices. While Domain-Driven Design is used to identify domains and divide application into modules, the Hexagonal Architecture is used to structure each module for a seamless future transition into a microservice.

The Honeycomb Monolith pattern is more than just a structural blueprint; it is a strategic response to the challenges faced in modern software development, particularly in managing complexity, maintaining agility, and ensuring scalability. As companies and developers grapple with the decision between starting with a monolithic application or diving straight into microservices, the Honeycomb Monolith presents a middle ground. It proposes a way to enjoy the benefits of a monolith – simplicity, rapid development, and fewer cross-cutting concerns – while setting the stage for a smoother transition to microservices, should the need arise.

In this chapter, we will first cover the core principles of Domain-Driven Design, understanding its role in aligning software development with business needs and how it drives the structure of the Honeycomb Monolith. We will then explore the Hexagonal Architecture, a pattern that promotes decoupling business logic from external concerns, providing insights into its benefits, our adaptations for the Honeycomb pattern purposes, and how it is used to structure Honeycomb modules.

Having laid out the foundation we will provide a detailed description of the Honeycomb Monolith pattern. Here, we lay out the structural nuances, explaining how each module within our monolith encapsulates a distinct domain, resembling the hexagonal cells of a honeycomb. This structure not only improves maintainability within a monolithic architecture but also sets a clear path for transitioning individual modules to microservices, ensuring that the system's evolution aligns seamlessly with the changing business requirements.

In this chapter we will uncover how the Honeycomb Monolith pattern embodies a pragmatic yet innovative approach to software architecture, addressing the dilemma of choosing between a monolith and microservices for greenfield projects. It stands as a

testament to the idea that earning architectural credit early on can help create an evolutionary design strategically positioned for smooth transition into microservices.

2.1. Domain-Driven Design as Foundation for Modular Breakdown

2.1.1. Defining Domain-Driven Design

Domain-Driven Design (DDD) is a software design philosophy that focuses on the core business domain and models software around it. It emphasizes a deep understanding of the domain's complexities which are then directly reflected in the software architecture and design. The key to DDD is the creation of a model based on the real-world business context, involving a ubiquitous language that is consistently used across all stakeholders, including both developers and domain experts, in documentation and in code [22].

Relevant DDD terminology includes [23]:

- *Domain:* The sphere of knowledge and activity around which the software is built. It encompasses the business logic, rules, and processes.
- *Model:* An abstraction that describes selected aspects of the domain and can be used to solve problems related to that domain.
- *Ubiquitous Language:* A common language used by developers and domain experts to ensure clear and consistent communication.
- *Bounded Context:* A clear boundary within which a particular domain model is defined and applicable, minimizing ambiguity and confusion.

2.1.2. DDD Use in Microservices Design

Microservices architecture, characterized by its division of a system into small, independently deployable services, employs DDD principles for building a scalable and resilient system.

In a microservices architecture, each service ideally encapsulates a specific domain or subdomain, mirroring the DDD approach of a bounded context [22]. This ensures that each microservice has a clear, focused purpose and reduces the complexity associated with overlapping domains.

The DDD practice of establishing a ubiquitous language for each bounded context helps in defining clear and consistent interfaces for microservices [24]. This common language ensures that each service’s functionality is easily understandable, not just by

the development team but across the entire organization, thereby facilitating better collaboration and integration.

DDD encourages decentralized decision-making and governance, which is a natural fit for microservices. Each microservice manages its own domain logic and data storage, promoting autonomy and reducing dependencies on other services [24]. This decentralized approach enables teams to develop, deploy, and scale their services independently, enhancing the system's overall availability and resilience.

When designing microservices, DDD serves as a guiding framework for identifying and defining the boundaries of each service. This process involves [23]:

1. Identifying aggregates, which are clusters of domain objects that can be treated as a single unit. In microservices, these aggregates help define the scope and boundaries of a service, ensuring that each service is cohesive and self-contained.

2. Establishing service autonomy and integration via well-defined APIs to ensure loose coupling between independent services.

3. Complexity management by ensuring that each service is only concerned with its own domain logic. This keeps the system manageable, even as it grows and evolves.

We propose to utilize DDD in the Honeycomb Monolith pattern to identify domains and respective bounded contexts which will drive the modular structure for the monolith application. In the next section, we outline the concepts of Hexagonal Architecture and its inherent benefits as a second building block used in the Honeycomb Monolith pattern to structure each individual module.

2.2. Hexagonal Architecture at the Heart of Module Internal Structure

2.2.1. Principles of Hexagonal Architecture

Hexagonal Architecture, a term coined by Alistair Cockburn, presents a model for designing software applications around the principle of portability and adaptability [25]. This architectural style, also known as 'Ports and Adapters' [25], fundamentally rethinks how data and control flow into and out of an application, placing the business logic at the core of the design.

At the heart of hexagonal architecture lies the application's domain logic. This core encapsulates the domain model and the business rules, fundamentally driving the application's behavior. It is isolated from external concerns, ensuring that the business logic remains pure and unobstructed by peripheral elements like UI, inbound requests, database interactions, or external service calls [25].

The architecture introduces the concept of 'ports', which serve as gateways to the application. Ports define abstract interfaces for the functionality that the application offers to the outside world (outgoing port) or requires from it (incoming port) [26]. These abstractions form a contractual boundary, dictating how external agents can interact with the application. Presence of ports offers ensured that domain will only ever depend on these interfaces and not the external adapters.

Surrounding these ports are 'adapters'. An adapter is a component that converts data from the format most convenient for an external agency or technology (like a web interface, a database, or another application) into the format most convenient for the port and vice versa. This means the same business logic can interact with different external agents without any modification, simply by using different adapters.

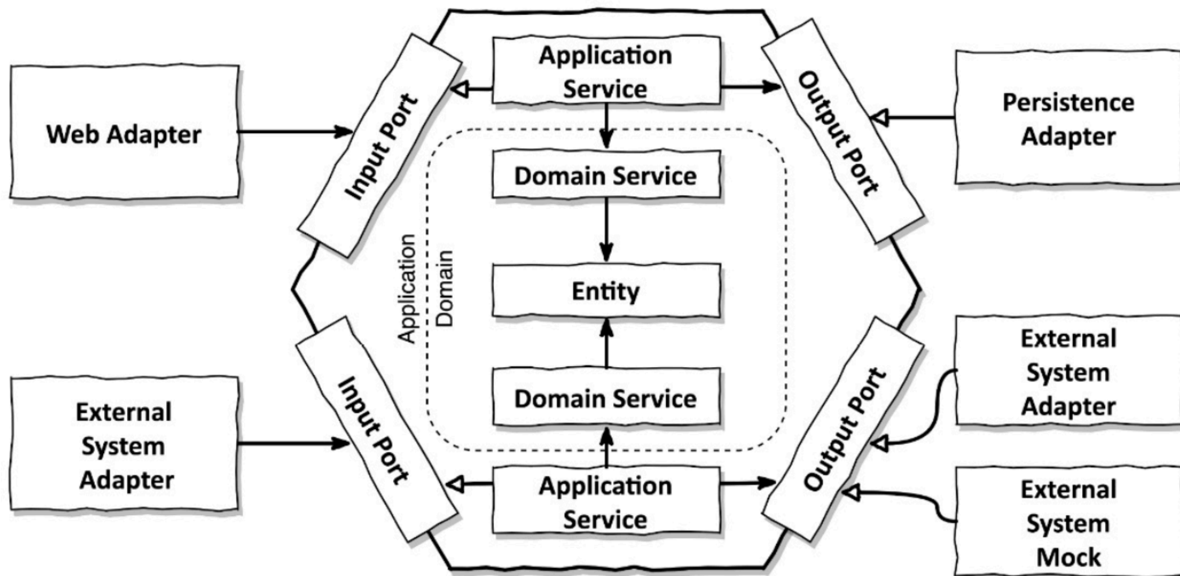

Figure 4: Hexagonal Architecture [26]

2.2.2. Benefits in Software Design

Hexagonal Architecture, with its distinctive separation of concerns and modular structure, brings several significant benefits to software design. These advantages are

not merely technical but also align with broader objectives of maintainability, scalability, and adaptability in software development.

One of the most prominent benefits of Hexagonal Architecture is the promotion of modularity. By encapsulating the business logic within a distinct, central location and interfacing with the external world through ports and adapters, the architecture inherently fosters a modular approach [25]. This modularity improves maintenance, as changes in one part of the system, such as an external interface or database, do not affect the core business logic [26].

By separating the core logic from external interactions, hexagonal architecture promotes loose coupling [26]. The domain does not depend on external components such as databases or web services; instead, it interacts with abstract ports while the specific implementation details of managing external communications are handled by adapters.

This architecture greatly simplifies testing. The business logic can be tested in isolation, without needing to set up its external dependencies. This isolation enables more focused and faster unit tests, leading to a more robust and reliable codebase [26].

The use of ports and adapters makes the system inherently portable, flexible, and adaptable. Changing or adding different types of interfaces (like a new database or a different web framework) becomes a matter of adding or updating adapters, without needing to change the core business logic. This interchangeability is particularly beneficial in environments where technological shifts are common, allowing for the easy adoption of new technologies without disrupting the core application logic.

In the context of modern software development, where systems often need to be flexible, adaptable, and easily testable, the principles of Hexagonal Architecture offer significant advantages. It provides a clear pattern for isolating the domain model and logic from the external communication or persistence concerns which in turn provides domain portability. Domain portability is of special benefit to the modules in the Honeycomb Monolith pattern as they enable smooth transition of modules into individual microservices with minimum business risks.

2.2.3. Lightweight Approach in Honeycomb Monolith

Traditional Hexagonal Architecture pattern was designed with an entire application in mind warranting extensive use of ports and adapters. Complexity scales exponentially while applying this approach to structuring individual modules in a multi-modular application. Thus, we propose to simplify the traditional hexagonal structure all while preserving the core concept of domain isolation.

A more practical structure illustrated in Figure 5 would describe *endpoints* as layers responsible for inbound requests. These layers depend only on the *domain* and can access it directly. Removing ports between *endpoint* and *domain* reduces complexity all while maintaining *domain* layer isolation. At the same time, *the endpoint* layer is expected to be subject to future change and hence removing the ports does not impair the ease of swapping endpoint implementations when the need arises.

Further, *provider* layers are responsible for the outbound requests (whether over network, via direct method calls to other modules, or to a datastore). Removing ports here would cause the *domain* layer to depend on specific provider implementations which defeats the whole purpose of domain isolation. Hence, *the domain* layer encompasses outbound port interfaces and depends on these abstractions. At the same time, the *provider* layers implement those interfaces. This use of ports as part of the *domain* layer enforces dependency inversion [27] which guarantees domain isolation. Provider implementations can be swapped without affecting the domain logic.

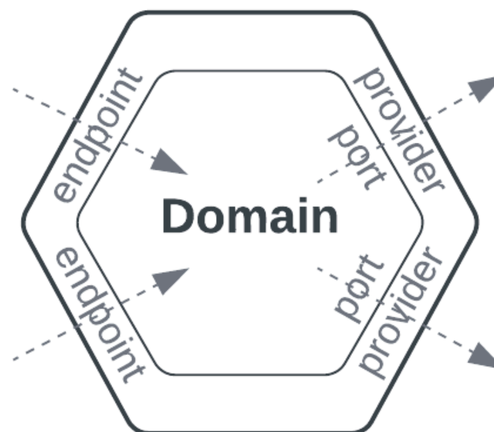

Figure 5: Lightweight hexagonal module structure

This adaptation of Hexagonal Architecture to application module structure reduces complexity all while ensuring domain isolation and endpoint/provider interoperability. We will provide a concrete implementation example in Chapter 3.

In the next section, we will describe the Honeycomb Monolith pattern and explain how both Domain-Driven Design and Hexagonal Architecture play their part in the proposed evolutionary monolith design.

2.3. Honeycomb Monolith Pattern Description

2.3.1. Structural Details of the Honeycomb Monolith Pattern

The Honeycomb Monolith pattern is a forward-thinking approach in the design of monolithic applications, where Domain-Driven Design (DDD) principles are integrated with the adaptable Hexagonal Architecture. This pattern distinctly identifies and separates business domains along with their associated data, organizing the application into modules, each representing an individual domain. These components are structured as hexagons, a design choice that not only isolates the domain logic from external communication interfaces but also symbolically represents the pattern's core philosophy. Like a honeycomb's hexagonal cells, each module in this pattern is self-contained and poised for an effortless transition into an independent microservice. The analogy with a honeycomb captures the essence of this pattern – a cohesive structure composed of distinct, efficiently organized elements, each contributing to the overall strength and adaptability of the system.

Figure 6 illustrates a detailed example structure of a Honeycomb Monolith pattern. Each hexagon symbolizes an application module corresponding to a distinct domain in the sense of DDD. Inner structure of each module isolates the *domain* layer and exposes *endpoint* and *provider* layers for external communications. Each domain is associated with its own dedicated data storage, ensuring that the data is managed independently and is decoupled from other domains.

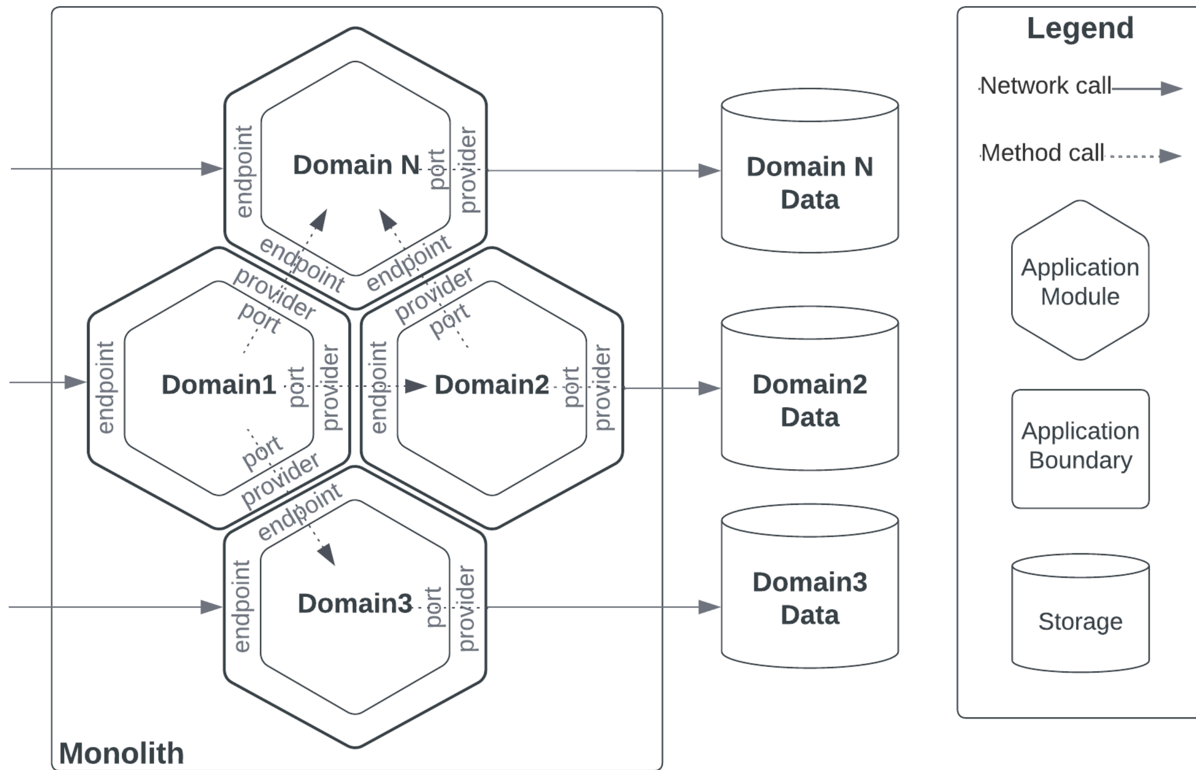

Figure 6: Honeycomb Monolith pattern

The initial step in the Honeycomb Monolith design involves using DDD to systematically identify distinct domains within the application. This process requires a deep analysis of the business requirements, collaborating with domain experts to gain understanding of the various business areas, their rules, and their interrelationships. Each identified domain is then delineated into a bounded context which translates to an application module that defines the limits of a particular domain, encapsulating its business logic, data, and language. This delineation ensures that each module in the Honeycomb Monolith pattern is focused, cohesive, and aligned with a specific area of business functionality. We understand modules in a broad sense of code organization to leave implementing teams with interpretation freedom. Modules can mean separate directories, packages, language specific constructs, etc.

A fundamental aspect of the Honeycomb Monolith pattern is the autonomy of data within each domain. When delineating the bounded contexts for each module, equal emphasis is placed on the segregation of data as well as the domain logic and models. This approach dictates that each module manages its own database or data store, independent of others. Such an arrangement not only strengthens the integrity and cohesion of each domain but also anticipates the eventual evolution into microservices. In a microservices architecture, each service typically controls its own data, thus, by

designing each module in the monolith to independently manage its data, the Honeycomb pattern effectively prepares the system for a smoother transition to microservices. This multi-database approach within the monolith represents a strategic departure from traditional monolithic architectures, where a single database is common, and underscores the pattern's foresight in aligning with modern, distributed system paradigms.

Following the identification of bounded contexts, the Honeycomb Monolith pattern conceptualizes each module as a hexagonal cell. These cells represent the physical structure of the bounded contexts and are designed to encapsulate and isolate their respective domains. The periphery of each hexagon comprises endpoints and providers that manage interactions with other modules or external components such as databases, user interfaces, or external APIs. These layers serve as the bridge between the core domain model and logic and the rest of the application or the outside world, ensuring a clean separation of concerns and domain isolation. As shown in Figure 6, communication between modules can be simplified to direct method calls while only resorting to communication over the network when integrating with external systems.

Each hexagon is inherently designed to potentially evolve into an independent microservice. This foresight in design eases the transition from a monolithic to a microservices architecture, as the *endpoint* and *provider* layers are designed to change without affecting the *domain* layer.

Figure 7 illustrates just one such module after transition into a standalone microservice. As expected, the axis of change goes only through the *endpoint* and *provider* layers at the module boundary while *domain* and data are not affected. We will provide a more detailed view into migration planning and execution in Chapter 3 example.

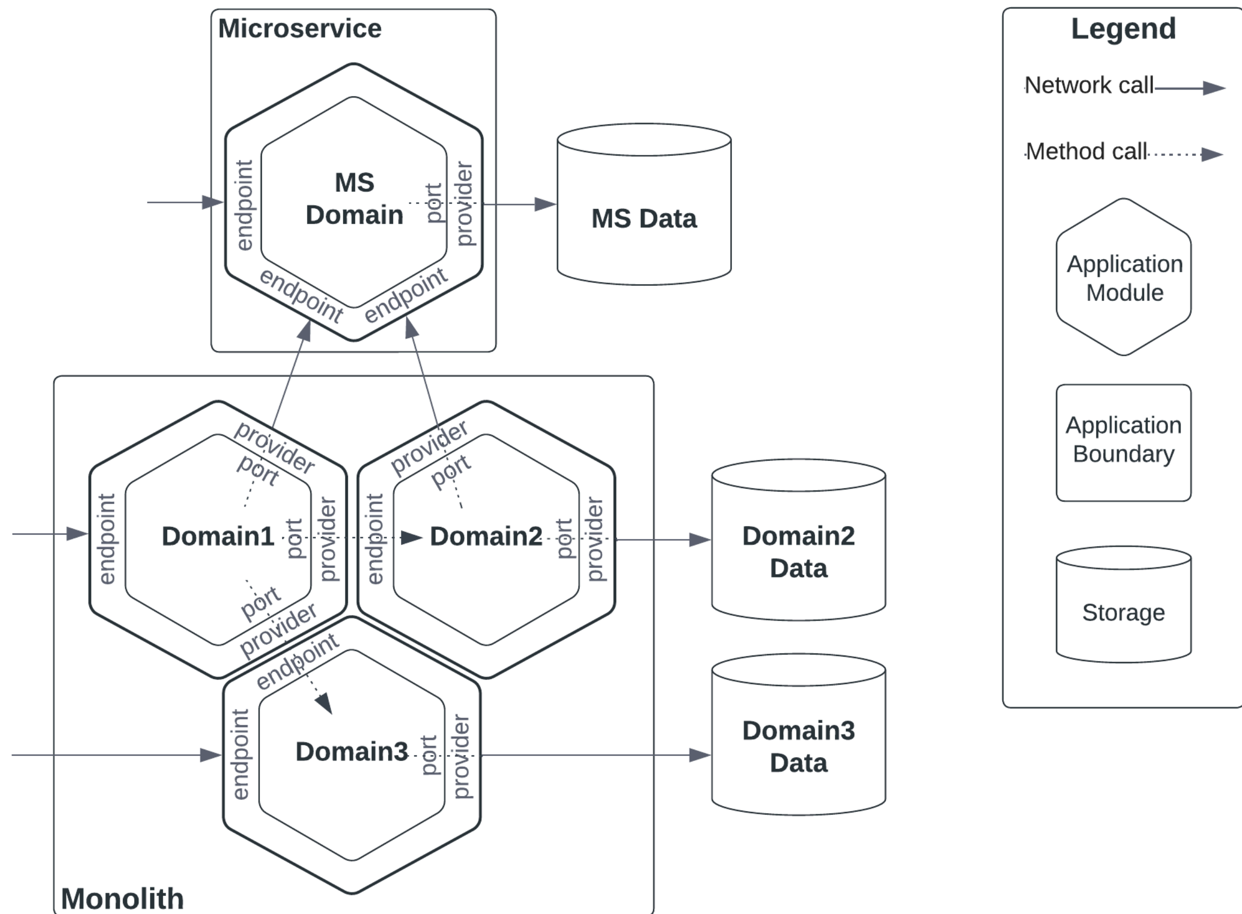

Figure 7: Honeycomb Monolith with microservice extracted

2.3.2. Benefits of the Honeycomb Monolith

The Honeycomb Monolith pattern presents a number of strategic advantages for monolithic software architecture that seeks to minimize the complexity of decomposition into microservices. It allows to reap the inherent benefits of monolithic application development while reserving the option for an agile migration of any of its parts into individual microservices.

- **Cohesive Modularity:** The pattern's core tenet is its modular approach, which offers a clear organizational structure. Each hexagonal module operates independently, housing a specific domain's logic and data. This modularity promotes cohesive development efforts, as teams can focus on discrete areas of the application without the risk of cross-contamination of concerns. Changes to a single domain business logic are limited to that single module and will not affect the rest of the application.

- **Smooth Transition to Microservices:** Each module's design within the pattern is microservice-ready, meaning that when the need arises, a module can be seamlessly transitioned into a standalone microservice. This readiness mitigates the typical challenges associated with transitioning from a monolithic architecture to a microservices one, such as domain entanglement and data migration issues.
- **Decreased Maintenance Cost:** Honeycomb Monolith pattern requires a clear domain boundary and model structure. The architectural investment in defining clear domain boundaries, along with the associated endpoints and providers for each module, requires upfront effort. This increased complexity at the outset can be viewed as accruing architectural credit that pays dividends when scaling or transitioning to microservices. While it demands a commitment to meticulous design and disciplined implementation from the very beginning; the application of the Honeycomb Monolith pattern will keep maintenance cost relatively low as the application evolves and will pay off in the long term.
- **Independent Data Management:** A distinctive feature of the Honeycomb Monolith is the separation of data stores for each domain. This independence aligns with microservices best practices and allows for more flexible data management strategies, such as choosing different database types appropriate for each domain's needs.
- **Seamless Integration with External Systems:** A significant benefit of the Honeycomb Monolith pattern is its inherent ease of integration with external systems. Owing to the hexagonal architecture, external dependencies can be incorporated with minimal disruption by simply adding a provider to the relevant module. Consequently, there's no need for widespread changes across the entire application, preserving the stability of the system while enabling targeted updates and expansions. This architectural characteristic also enables technological diversity within the application; new services can be developed in different technologies best suited for their specific purposes and integrated as needed. The adaptability offered by the Honeycomb Monolith pattern thus positions it as a highly versatile approach when integration with an ever-growing ecosystem of services and tools is required.
- **Simplified Testing and Maintenance:** Testing is simplified as each module can be tested in isolation, with mock adapters used to simulate external interfaces. Maintenance becomes more straightforward as well, with changes to one module having minimal impact on others, reducing the risk of unforeseen bugs during updates.

2.3.3. Drawbacks of the Honeycomb Monolith

While the Honeycomb Monolith pattern sets the stage for an agile, scalable application architecture conducive to a microservices transition, it is accompanied by a set of trade-offs. The upfront investment in maintenance, the requirement for rigorous data management, and the overhead of managing extensive mapping and model duplication are considerations that must be weighed against the pattern's benefits. These drawbacks must be carefully considered in the context of the application's size, complexity, and long-term strategic goals.

- **Complexity of Domain Identification and Modeling:** A considerable drawback is the inherent complexity involved in the exercise of Domain-Driven Design domain identification and modeling. This process demands a thorough understanding of the business context, requiring significant collaboration between software developers and domain experts. The task of dissecting a business into discrete domains and bounded contexts is not trivial as it requires time, meticulous analysis, and iterative design.

- **Distributed transactions:** Another critical drawback of the Honeycomb Monolith pattern arises from its emphasis on domain and data separation, potentially involving multiple databases. In such a structure, transactions spanning multiple domains effectively become distributed and will need to be managed with distributed transaction mechanisms. This shift often leads to complex transaction management strategies, such as implementing a two-phase commit or utilizing the Saga pattern commonly seen in microservices architectures. These approaches introduce additional complexity in terms of development, testing, and maintenance.

- **Governance effort**: Honeycomb Monolith pattern requires significant governance effort to maintain its rigorous module structure. This architecture demands meticulous architectural decisions to ensure each module maintains its domain context boundary and the internal hexagonal structure. Maintaining this consistency and enforcing architectural guidelines requires continuous oversight. This is particularly evident in the need for architectural unit tests. The ongoing need for such governance not only adds to the complexity of the development process but also places a substantial responsibility on the project's architecture team to continuously monitor, guide, and adjust the application's structure.

- **Rigorous Data Discipline:** The pattern's insistence on domain autonomy extends to the management of separate data stores for each module, which can place a burden on the system's data discipline. Maintaining multiple database connections and ensuring transaction management across these databases

necessitates a robust infrastructure. This architecture may lead to potential data duplication and demand complex configuration management.

- **Model Duplication and Mapping Overhead:** Each module within the Honeycomb Monolith is expected to maintain its own model representation, adhering to the DDD principles. Similarly, to reduce dependencies between ‘endpoint’ and ‘provider’ layers they too need to maintain their own model representation (covered in more details in Chapter 3). While this reinforces the modularity and independence of domains, it also leads to a significant amount of model duplication and excessive mapping across the system. This not only increases the development workload but also adds to the cognitive load for developers who must navigate and manage these multiple, sometimes overlapping model representations.

2.4. Summary

In conclusion, Chapter 2 layed out a conceptual framework of the Honeycomb Monolith pattern, an architectural blueprint that harmonizes the Domain-Driven Design with the Hexagonal Architecture. This pattern is a robust solution for constructing maintainable monolithic applications that are inherently prepared for a smooth transition into microservices. Through the strategic separation of domains and their data, and the encapsulation of business logic within distinct hexagonal modules, the Honeycomb Monolith pattern brings a structured and agile approach to application design.

However, the pattern is not a silver bullet and introduces its own set of challenges, including complex transaction management, the necessity for strict data management disciplines, and the complexities inherent in domain modeling. The pattern demands a forward-thinking mindset and a willingness to invest in more complex architectural structure from the outset.

As we transition from the conceptual framework of the Honeycomb Monolith pattern to the practical considerations of its implementation, Chapter 3 will address how the theoretical advantages of the Honeycomb Monolith translate into real-world application development. We will explore the architectural decisions, the development process, and the step-by-step migration strategy that turns the theory into a functioning system. The subsequent chapter serves as a bridge from the conceptual groundwork laid here to the pragmatic journey of bringing the Honeycomb Monolith to life.

Chapter 3: Implementing Honeycomb Monolith

Chapter 3 is a journey from concept to concrete, detailing the implementation of the Honeycomb Monolith pattern, its migration to microservices, and the broader implications in the field of software engineering. It provides valuable insights for architects, developers, and project managers seeking to navigate the complexities of monolith to microservices transition.

We begin with an exploration of the implementation context for the Opora API, providing insights into the specific environment, requirements, and technical stack that shaped the application's architecture. The heart of the chapter lies in the delineation of the Opora application's monolith design. We examine how each domain is encapsulated within a hexagonal module and outline the structural nuances that define the Honeycomb Monolith pattern. This is followed by an in-depth discussion on the enforcement of the architecture, highlighting the mechanisms employed to maintain the integrity of the design throughout the development process.

The chapter then navigates through the Opora application migration to microservices. This section is an exploration of migration steps and operational considerations that guided the transition. We provide a granular view of the code changes undertaken during the migration, shedding light on the practical realities of moving from a monolithic to a microservices architecture with the help of the Honeycomb Monolith pattern.

The results of implementing and migrating the Honeycomb Monolith are then critically evaluated, offering a perspective on the successes and challenges encountered. Finally, the chapter concludes by broadening the lens to examine the applicability of the Honeycomb Monolith pattern in various contexts.

3.1. Opora API Implementation Context

To showcase Honeycomb Monolith in action we implemented the Opora API [28] [29]. The Opora application connects donors with humanitarian initiatives that address urgent crises and help those in need, a relevant topic amid the ongoing russian invasion in Ukraine. The API provides simple CRUD (create, read, update, delete) operations and leaves complex logic out of scope of this project.

We identified three subdomains driving the structure of the monolithic application: Initiatives, Sponsors, and Media. Initiatives embody the humanitarian drives in response to ongoing crises that are created with the specific goal of collecting tangible help (blankets, water, power banks, clothes, etc.). Initiatives are organized and managed by registered users - Sponsors. We also identified Media subdomain as the means to store

and retrieve media used within the application (Sponsors images or illustrations used for the Initiatives). Opora API and back-end implementation were designed specifically for this project with the goal of illustrating the Honeycomb Monolith pattern. Therefore, the primary goal was having several distinct subdomains with simple domain logic.

Opora back-end was implemented with Java 17 and Spring Boot 3 with Gradle as the build tool [30]. Opora OpenAPI specification [31] was used for controller and model code generation with the help of OpenApi Generator Gradle plugin [32]. Purely for technology diversity demonstration purposes we used a variety of database technologies: PostgreSQL [33], MongoDB [34], and MinIO [35]. The application is designed to be packaged and deployed in a Docker container [36]. JUnit 5 [37], Mockito [38], AssertJ [39], and Testcontainers [40] were used for testing purposes.

In the Opora application Honeycomb Monolith modules were implemented as plain Java packages to provide the simplest technology and tool agnostic example. At the same time, modules can be implemented as Java Platform Module System JSR 376 [41] or with Spring Modulith project [42].

3.2. Opora Application Monolith Design

3.2.1. Domain-Driven Modular Design

Opora API was designed with three clear domains in mind: Initiatives, Sponsors, and Media. Each of these domains were placed in a corresponding module within the Opora application. An API Gateway module was also identified primarily to illustrate inter-module communication and to expose as a uniform API. It serves a utility function and therefore does not have its own domain. Production code might not include this module at all or designate it with additional functionality such as authentication, monitoring, metric collection, etc.

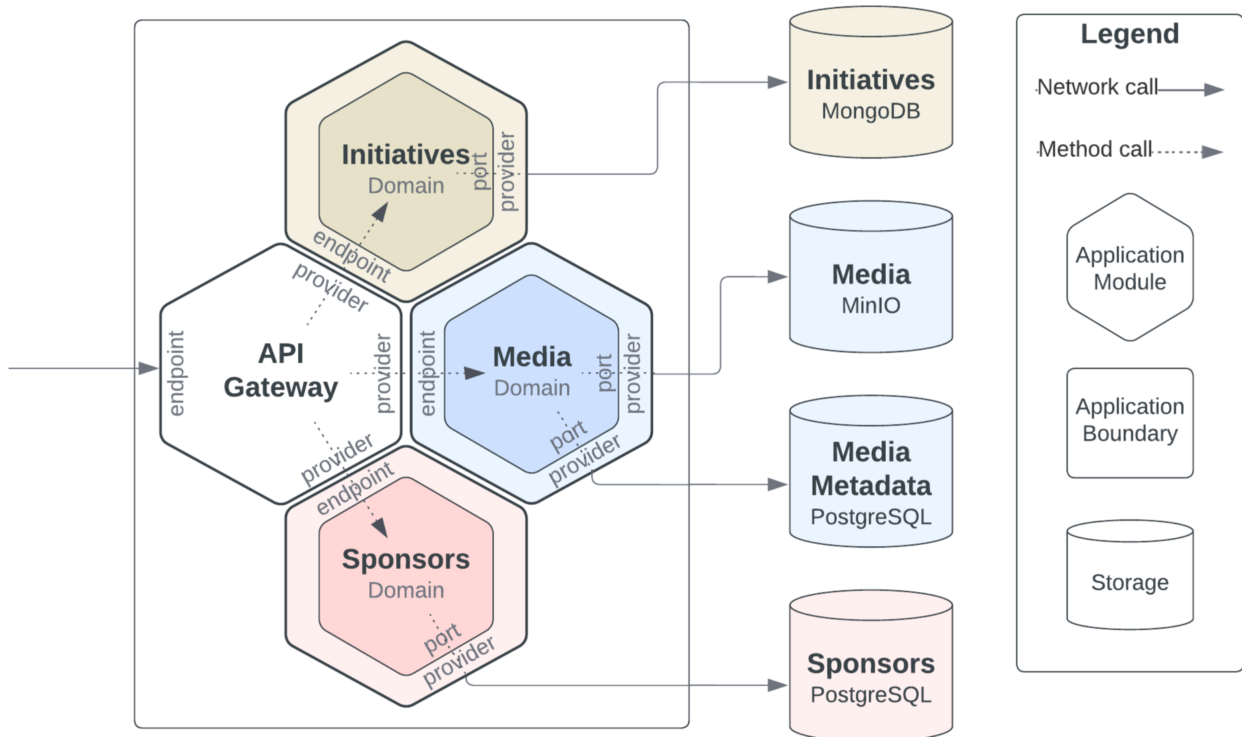

Figure 8: Opora application as a monolith

Figure 8 displays the modular design: incoming requests are accepted by the API Gateway module over the network and are propagated further to the Initiatives, Media, or Sponsors modules via direct method calls. Code organization is reflected in the same structure illustrated below in Figure 9 with each module residing in a dedicated package. ‘Common’ package accessible to any domain modules contains shared mapping configs and exceptions. It illustrates that some parts of the application may remain common, the goal of Honeycomb Monolith is to keep such dependencies to the minimum.

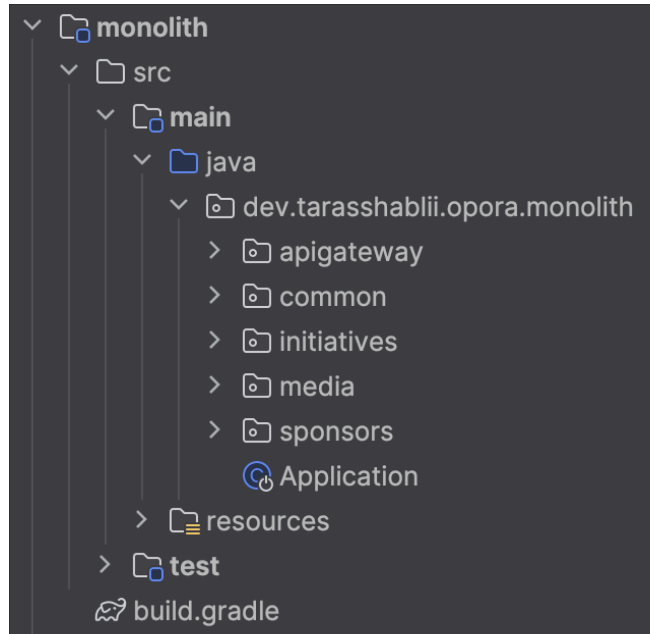

Figure 9: Opora application package structure

An important aspect of Honeycomb Monolith is data isolation. Figure 8 illustrates how each module manages its own data, all strategically placed in distinct databases. To illustrate technology diversity that the Honeycomb Monolith pattern affords Initiatives data is stored as documents in MongoDB, Media files reside in MinIO, Media metadata is kept in relational PostgreSQL, and Sponsors data is also stored in PostgreSQL. Note that while Sponsors and Media metadata are both stored in PostgreSQL the data are isolated and reside in separate instances. Each database connection is separately configured in the application.properties [43]:

```
# Sponsors Datasource
sponsors.database.host=localhost
sponsors.database.port=5432
spring.datasource.jdbc-url=jdbc:postgresql://${sponsors.database.ho
st}:${sponsors.database.port}/sponsors?stringtype=unspecified
spring.datasource.username=sponsors_user
spring.datasource.password=sponsors_pass
spring.jpa.hibernate.naming.physical-strategy=org.hibernate.boot.mo
del.naming.PhysicalNamingStrategyStandardImpl

# Initiatives Datasource
spring.data.mongodb.username=initiatives_user
spring.data.mongodb.password=initiatives_pass
spring.data.mongodb.host=localhost
spring.data.mongodb.port=27017
spring.data.mongodb.database=initiatives
```

```
spring.data.mongodb.authentication-database=admin
spring.data.mongodb.uuid-representation=standard

# Media Datasource
media.database.host=localhost
media.database.port=9000
media.database.username=media_user
media.database.password=media_pass
media.database.url=http://${media.database.host}:${media.database.p
ort}
media.database.bucket=media

# Media Metadata Datasource
media.metadata.database.host=localhost
media.metadata.database.port=5433
media.metadata.datasource.jdbc-url=jdbc:postgresql://${media.metada
ta.database.host}:${media.metadata.database.port}/media_metadata
media.metadata.datasource.username=media_user
media.metadata.datasource.password=media_pass
spring.jpa.generate-ddl=true
```

In this section we described the overall modular structure of the Opora application focusing on domain identification and placement in separate modules. In the next section we will describe each module's structure shaped by the Honeycomb Monolith pattern.

3.2.2. Module Hexagonal Structure

Each module within the Opora application is structured as a lightweight hexagon described in Section 2.2.3. Inbound requests are handled by the *endpoint* adapter layer which depends directly on the *domain* layer (with the exception of API Gateway domainless module). *Domain* layer depends on *port* abstraction which is implemented in the *provider* adapter layer responsible for the external communication.

Port abstraction ensures that the *domain* layer does not depend on *provider* layer implementations. Figure 10 illustrates this dependency inversion as implemented in the Initiatives module. InitiativesService depends on InitiativeProvider interface *port*, part of the *domain* layer. InitiativeProviderImpl implements this interface and the implementation sits within the *provider* layer. This simple structure inverts the dependency and ensures the *domain* is self-contained and does not depend on *provider* implementations.

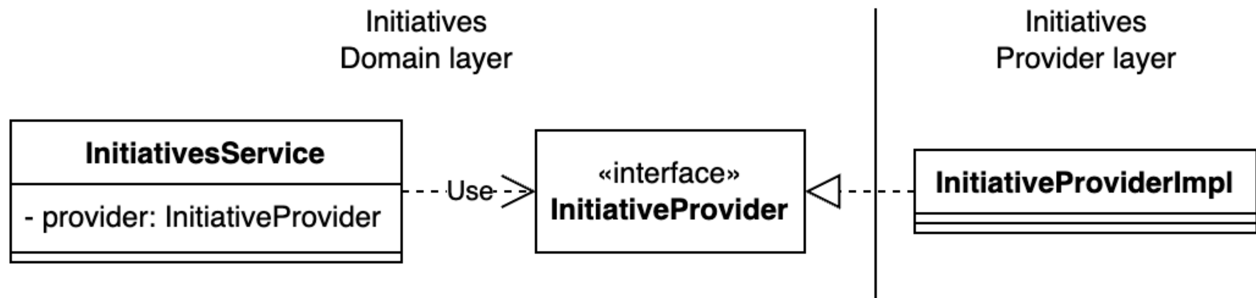

Figure 10: Dependency inversion

Figure 11 illustrates code organized within the Initiatives module (other modules follow suit):

- **Endpoint** package contains the InitiativesFacade responsible for accepting external calls from the API Gateway module, mapping data transfer object (DTO) into domain model, and calling domain InitiativesService. Endpoint package also contains its own DTOs and mappers.
- **Domain** package consists of InitiativesService responsible for executing domain business logic on the model. InitiativeProvider interface is injected into the InitiativesService to provide an abstraction layer between domain logic and provider implementation details.
- **Provider** package contains InitiativesProviderImpl which implements the domain InitiativesProvider interface. InitiativesProviderImpl is injected with InitiativesRepository and knows how to map domain model into repository entity and how to communicate with the repository thus shielding domain from this concern. Provider package maintains its own entity model and mappers.

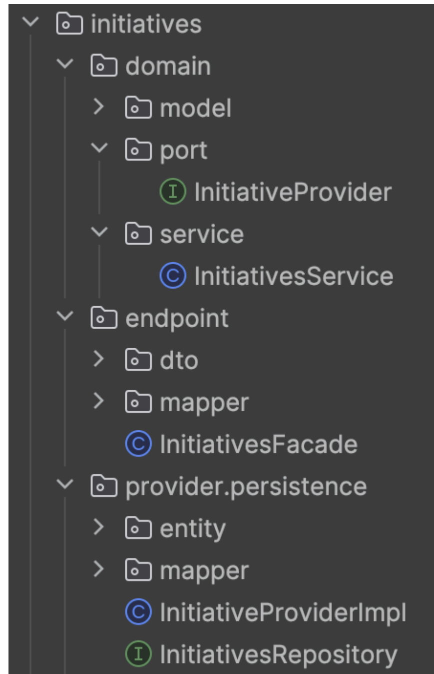

Figure 11: Initiatives module code organization

In order to maintain independence from the upstream calling layers each downstream layer needs to maintain its own model. For instance, when API Gateway module InitiativesProvider calls the Initiatives module InitiativesFacade it can either pass its own InitiativesDto or map its DTO into the Initiative's counterpart. The first case would create InitiativesFacade dependency on the API Gateway provider layer as InitiativesDto would need to be imported. Figure 12 illustrates this scenario with unwanted circular dependency highlighted in red.

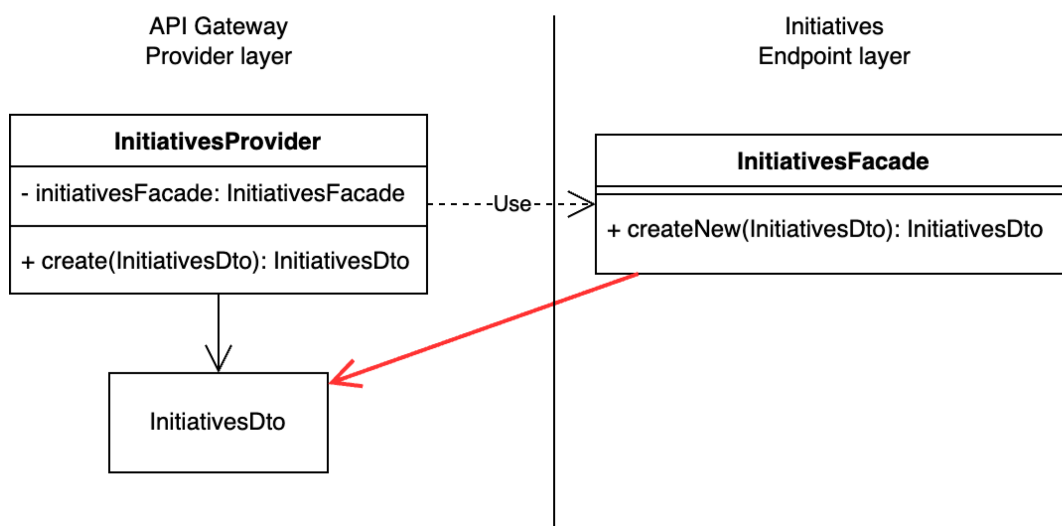

Figure 12: Shared DTO

To avoid this circular dependency each layer needs to maintain its own model. Upstream layers need to map their model into downstream model and only then call the downstream layers. Illustrated in Figure 13, Initiative *endpoint* layer now maintains its own InitiativeDto and does not depend on any of the API Gateway *provider* layer upstream components. API Gateway *provider* layer however does depend on the newly added initiatives.endpoint.dto.InitiativeDto, but the direction of dependencies is in line with the data flow and will not further encumber the system design. InitiativeDtoMapper is injected into InitiativesProvider and is tasked with mapping between the apigateway.provider.dto.InitiativeDto and initiatives.endpoint.dto.InitiativeDto.

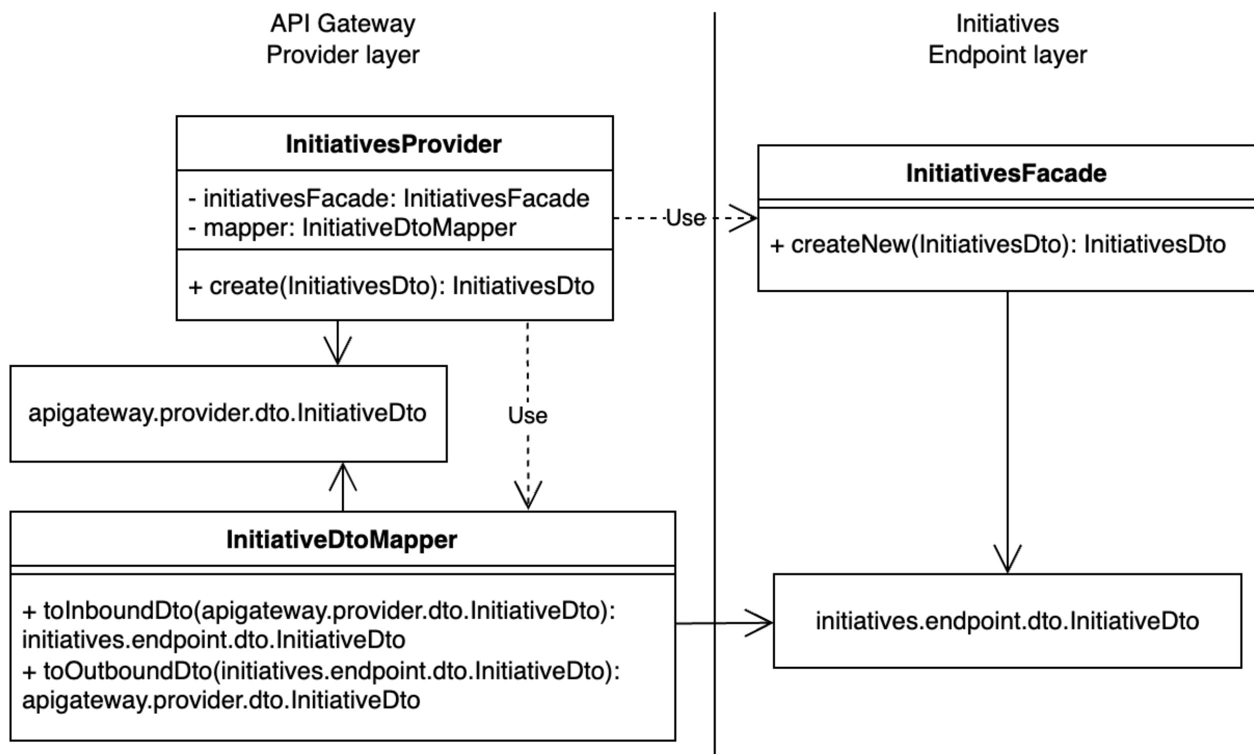

Figure 13: Separate DTOs per layer

The latter approach creates code duplication and forces extensive mapping, but achieves clear direction of dependencies which is vital for further smooth module migration.

The hexagonal module structure described in the section achieves several tactical advantages: *domain* logic and model isolation, clear separation of concerns, ease of integration with any number of upstream/downstream modules or external systems, and conscious dependency direction to achieve loose coupling. In the next section we will illustrate how both the modular application structure and the hexagonal design of each module can be enforced with a hard quality gate.

3.2.3. Honeycomb Monolith Architecture Enforcement

Within the Opora application, the structured code organization prescribed by the Honeycomb Monolith pattern is enforced with the ArchUnit [44] architectural tests. These tests are integrated into the Gradle *:test* task, which is executed as a part of the build process, serving as a hard quality gate within the continuous integration pipeline. This ensures that any undesired dependencies outside of the defined architectural standards are detected early, maintaining the integrity of the Honeycomb Monolith design throughout the development lifecycle.

The application structure illustrated in Figure 8 above shows that API Gateway *endpoint* layer depends on (accesses) API Gateway *provider* layer while the latter calls the respective *endpoint* layers of Initiatives, Media, and Sponsors modules. Within each of the Initiatives, Media, and Sponsors modules the *endpoint* layer can only access the respective *domain* layer. The latter are self-contained and do not depend on any other layers due to the *port* abstraction. Finally, *provider* layers depend on the respective *domain*. Most importantly, each module is only accessed from the outside through the *endpoint* layer. The core of the test enforces this structure [45]:

```
// API Gateway Hexagon (domainless)
.whereLayer(APIGATEWAY_ENDPOINT).mayOnlyAccessLayers(APIGATEWAY_PRO
VIDER)
.whereLayer(APIGATEWAY_PROVIDER).mayOnlyAccessLayers(
       INITIATIVES_ENDPOINT, MEDIA_ENDPOINT, SPONSORS_ENDPOINT)
// Initiatives Hexagon
.whereLayer(INITIATIVES_ENDPOINT).mayOnlyAccessLayers(INITIATIVES_D
OMAIN)
.whereLayer(INITIATIVES_DOMAIN).mayNotAccessAnyLayer()
.whereLayer(INITIATIVES_PROVIDER).mayOnlyAccessLayers(INITIATIVES_D
OMAIN)
// Media Hexagon
.whereLayer(MEDIA_ENDPOINT).mayOnlyAccessLayers(MEDIA_DOMAIN)
.whereLayer(MEDIA_DOMAIN).mayNotAccessAnyLayer()
.whereLayer(MEDIA_PROVIDER).mayOnlyAccessLayers(MEDIA_DOMAIN)
// Sponsors Hexagon
.whereLayer(SPONSORS_ENDPOINT).mayOnlyAccessLayers(SPONSORS_DOMAIN)
.whereLayer(SPONSORS_DOMAIN).mayNotAccessAnyLayer()
.whereLayer(SPONSORS_PROVIDER).mayOnlyAccessLayers(SPONSORS_DOMAIN)
```

In this section we described how the Honeycomb Monolith structure is implemented in the Opora monolith application. We described the overall modular design with each module organized as a hexagon and how the whole structure is enforced with architectural tests. In the next section we will describe the Opora application monolith to

microservices migration process and look at the changes that were required to execute the transition.

3.3. Opora Application Migration to Microservices

3.3.1. Migration Planning and Steps

Honeycomb Monolith pattern provides evolutionary agility to a monolithic application. This flexibility is afforded by the virtue of modular hexagonal structure. Promoting a single module into a microservice would not affect the entire application, but only the modules it interacts with. Any number of modules can migrate to microservices at the same time allowing flexibility in planning the transition.

Transforming any single module requires changes to the *endpoint* and *provider* adapter layers of this module as well as all other modules it interacted with. Direct method calls can be replaced by any means of communication that fulfills architectural requirements (REST, gRPC, async queue, etc.).

To illustrate Honeycomb Monolith solution of monolith to microservices migration we transformed the Opora application described in Section 3.2. The application resulted in four microservices: API Gateway, Initiatives, Media, and Sponsors Services. Admittedly, API Gateway makes little sense as a stand-alone bespoke microservice and would be replaced by one of the third-party solutions in production. We still went through with migrating it if only to illustrate the complete transition journey and the changes required to execute it. Figure 14 highlights the final Opora application structure implemented as microservices.

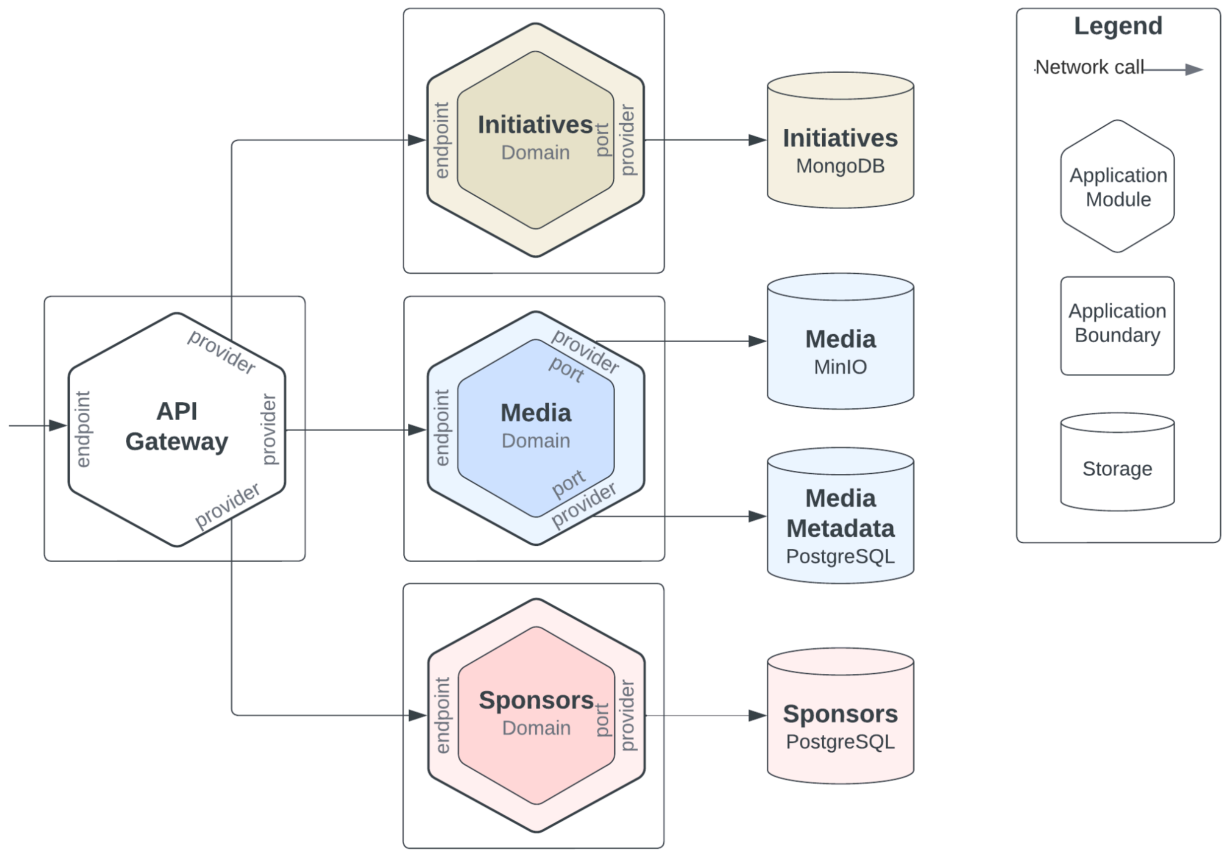

Figure 14: Opora application as microservices

Migrating each module into a stand-alone microservice followed the same pattern and required the following steps:

1. Creating new empty project
2. Transferring the code-base:
    a. Copy-pasting entire module package structure (production and test code)
    b. Transferring relevant resources such as application.properties
    c. Migrating pertinent build configuration
3. Code adjustments:
    a. Renaming packages
    b. Reimplementing *endpoint* and/or *provider* layers as needed
    c. Build configuration changes where needed
    d. Covering new implementations with unit and integration tests
    e. Enforcing application hexagonal structure with architectural tests

This transition resulted in creating microservices which retained their data connection and previous hexagonal module structure. Figure 15 illustrates the structure of the

microservices directory and the Initiatives service in particular (contrast with Figure 11 above):

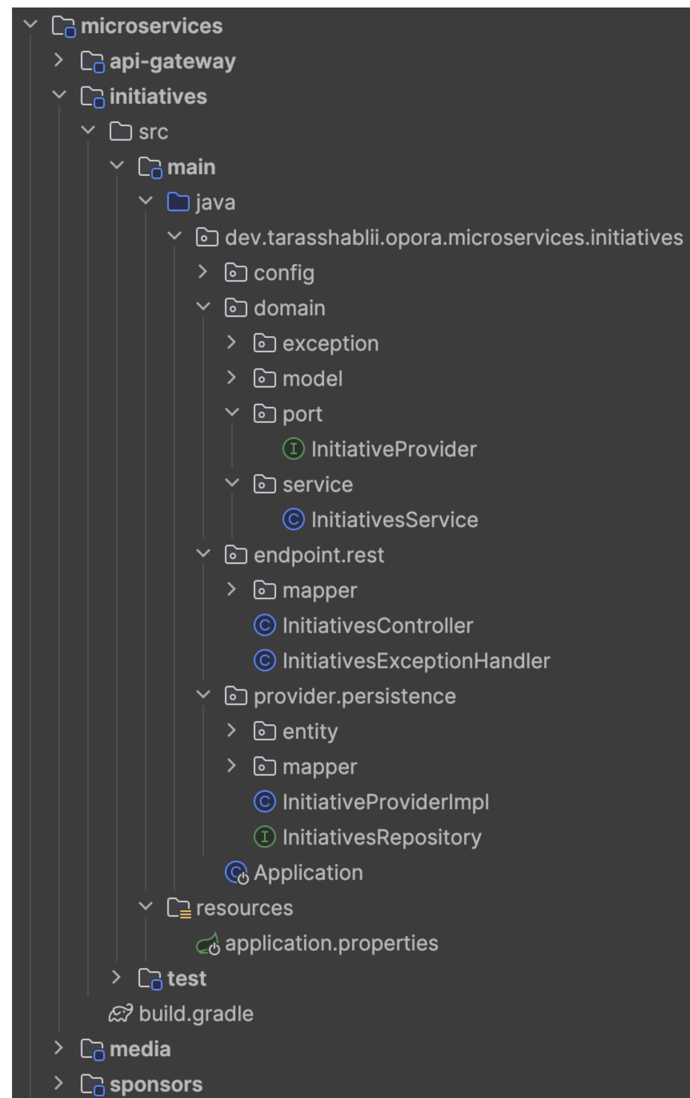

Figure 15: Initiatives service structure

This hexagonal structure is enforced by an architectural unit test similar to monolith [46]:

```
.whereLayer("endpoint").mayOnlyAccessLayers("domain")
.whereLayer("domain").mayNotAccessAnyLayer()
.whereLayer("provider").mayOnlyAccessLayers("domain")
```

3.3.2. Axis of Change

The code changes required to reimplement a Honeycomb Monolith module as a microservice are kept to the minimum highlighting the true strength of this pattern. By

identifying the domains' bounded contexts and keeping code and data separate, migrating a single module affects the other modules only along the communication boundary (*provider-endpoint*). The data was ready for migration since database inception and requires no restructuring. Hexagonal structure of each module further isolates domain logic and model shielding them from changes to the *endpoint* and *provider* layers that will undergo changes. In this section we will illustrate the changes that were required to migrate the Opora monolithic application to microservices.

During Opora application migration to microservices the only layers that required implementation changes were API Gateway *providers* and Initiatives, Media, and Sponsors *endpoint* layers as highlighted in Figure 16. In the monolithic application API Gateway *providers* communicated with downstream modules' *endpoints* via direct method calls. With the shift to distributed microservice architecture API Gateway *providers* communicate with downstream services via HTTP [47]. Likewise, the downstream services' *endpoint* layers are reimplemented into controllers exposing REST endpoints [48]. These are all the significant changes that will be required to transition Honeycomb Monolith to microservices.

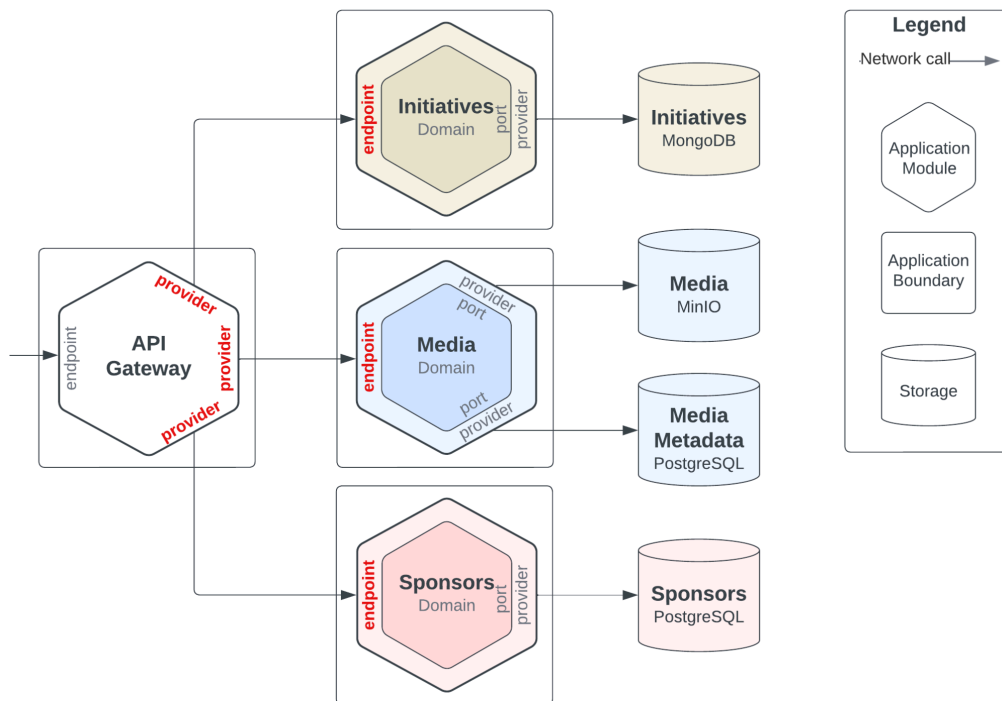

Figure 16: Axis of change (in red)

Perhaps, even more important are the changes that did not happen. Figure 17 illustrates that comparing monolith Initiatives module and Initiatives microservice *domain* layers reveals that the only changes were the package names. Neither the

InitiativesService business logic, nor the model structure underwent any changes. The Initiatives *provider* layer responsible for communication with the Initiatives database also remained unchanged. The same is true for the other Opora services.

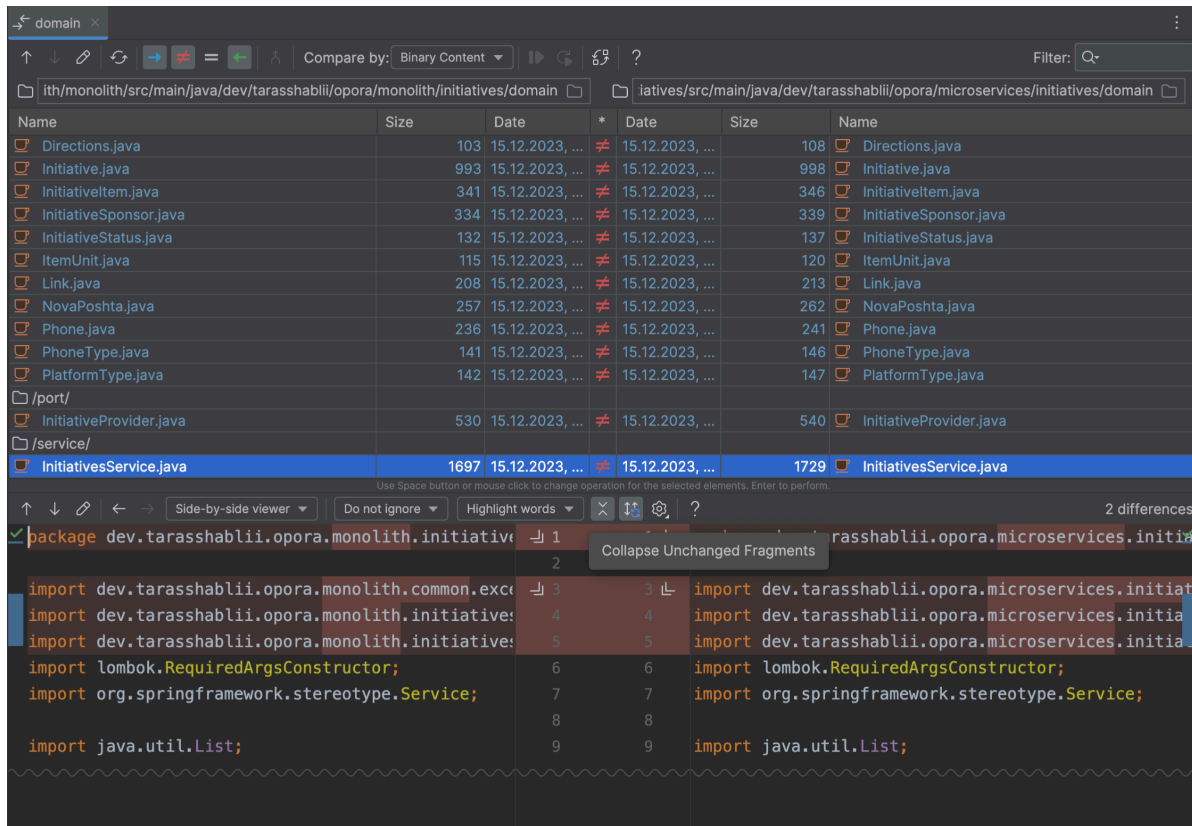

Figure 17: Monolith to microservice Initiatives *domain* comparison

3.4. Honeycomb Monolith Implementation and Migration Results

The transition from the Honeycomb Monolith to a suite of microservices within the Opora application yielded insightful results that validate the pattern's theoretical foundation and highlight practical considerations encountered during the migration process.

1. **Smooth Migration with Predictable Changes:** The migration process itself was characterized by its smooth execution. As anticipated, the most significant change involved transforming direct method calls into REST requests over the network. This shift was an expected aspect of moving from a monolith to a distributed system. The stability provided by the Honeycomb Monolith's structure facilitated a methodical conversion process. There were no surprises thanks to the clear domain boundaries enforced by the pattern.

2. **Stability of Business Logic and Data:** The most profound result of the migration was the stability of the business logic and data. Neither required changes, emphasizing the efficacy of the Honeycomb Monolith pattern's emphasis on strict domain separation and isolation. The pre-existing separation of concerns meant that domain logic and data were already prepared for the distributed nature of microservices. This result underpins low business operations impact during the migration process.
3. **Challenges with Model Duplication and Mapping:** The enforcement of strict layer separation necessitated that each layer maintain its own model representation, leading to extensive mapping between these models as described in Section 3.2.2. This introduced a degree of model redundancy and required considerable use of mapping between neighboring layers.
4. **Managing Multiple Database Connections:** Another practical challenge encountered was the management of multiple database connections within the Opora monolith. The pattern's commitment to domain data autonomy meant maintaining separate databases per module. Using PostgreSQL for Media metadata and Sponsors required additional configuration.

In summary, Section 3.4 highlighted the tangible outcomes of implementing the Honeycomb Monolith pattern in a real-world scenario. The migration smoothness confirmed pattern strategic value, while the challenges with model mapping and database management provided practical lessons in the trade-offs of this architectural choice. These results shape a deeper understanding of the Honeycomb Monolith's practical implications. In the next section we will discuss the pattern’s foreseen applicability in various contexts.

3.5. Honeycomb Monolith Applicability in Different Contexts

The Honeycomb Monolith pattern emerges as an architectural solution best suited for greenfield projects. Particularly in startups poised for rapid growth, the Honeycomb Monolith strikes a balance between the need for swift market entry with the capacity to scale efficiently. Startups benefit from the pattern’s modularity, which allows for the rapid development of core functionalities without sacrificing future scalability. When the need for scaling arises, each module can be evolved independently, providing an agile scaling strategy easily aligned with business priorities.

Additionally, an application designed with the Honeycomb Monolith pattern is not bound to an all-or-nothing migration strategy. Individual modules can be scaled by migrating them into microservices as needed, based on their specific load and performance requirements. This agility is particularly advantageous for systems experiencing uneven growth in demand across different domains.

In the context of existing monolithic systems the pattern serves as an intermediary step towards a microservices architecture. It offers a structured path for gradually disintegrating a monolith by identifying and migrating one domain at a time. This incremental approach reduces risk and allows for a controlled evolution of the application, aligning the migration process with ongoing business operations.

A serendipitous benefit that emerged is the pattern’s ease of integration with external systems and services. Due to the hexagonal modular design, any new integration or change affects only the relevant modules rather than the entire application. This localized impact simplifies updates, mitigates the risk of widespread system disruption, and enhances the application ability to integrate with external systems.

The Honeycomb Monolith pattern promotes technological diversity as organizations are not constrained to a single technology stack across their entire application. Instead, they can choose the most suitable technologies for additional services and easily integrate them with the monolithic application. This flexibility encourages innovation and allows teams to adopt new technologies that may offer performance or functionality benefits.

3.6. Summary

In this chapter, we journeyed through the practical implementation and migration aspects of the Honeycomb Monolith pattern, using the Opora application as a case study. This exploration provided valuable insights into how the pattern operates in a real-world setting and its implications for architecturally agile software development.

The Opora application's development journey showed the effectiveness of the Honeycomb Monolith pattern in facilitating a smooth transition of the Opora application to a microservices architecture revealing the pattern's readiness for future scalability needs and validating its practical utility.

However, this journey also highlighted some challenges. The necessity for extensive model mapping and managing multiple database connections within the monolith added complexity and operational overhead. Moreover, the initial investment in setting up the strict architectural enforcement and domain modeling demanded significant upfront effort.

The pattern's applicability in different contexts, from greenfield projects in startups to brownfield applications in established enterprises, was also established. Honeycomb Monolith flexibility in integrating with external systems without impacting the entire application emerged as an additional advantage.

Conclusions

This thesis commenced with a critical examination of the dilemma faced by startups and greenfield projects in choosing between monolithic and microservices architectures. By considering intermediary options such as a modular monolith, we identified a gap in existing research regarding evolutionary monolith structures. Traditional approaches often forced a binary choice between monoliths and microservices, overlooking the potential of a middle path that combines the strengths of both.

The Honeycomb Monolith pattern emerged as a bridge for this gap. It introduces a novel approach by integrating the principles of Domain-Driven Design (DDD) with Hexagonal Architecture to achieve domain isolation and portability. This pattern is designed to facilitate the evolution of a monolithic application into a microservices architecture seamlessly, without the need for extensive reworking or refactoring thus avoiding major business risks.

The practical application of the Honeycomb Monolith pattern was demonstrated through the Opora application case study. This implementation not only validated the viability of the pattern but also offered valuable insights into its operation. The executed migration was as smooth as anticipated, confirming the pattern's efficacy. The domain logic remained unaffected, with changes confined to the peripheral layers of *endpoints* and *providers*.

Despite its success, the implementation also brought to light certain challenges, such as the need for model duplication with extensive mapping and the complexity of managing multiple database connections. The requirement for upfront architectural investment in design and planning was also noted as a consideration against adopting this pattern.

The Honeycomb Monolith pattern holds significant applicability for startups seeking the best of both worlds: ease of development with fast time to market combined with the ability to seamlessly scale as the need arises. The pattern applicability extends to brownfield projects as an intermediary step towards microservices.

Looking ahead, future research could focus on simplifying the structure further where possible, exploring inter-module communication patterns for complex monolithic architectures, and addressing the added complexity in transaction management. This could potentially expand the pattern usability and efficiency further contributing to the field of software architecture.

References

[1] S. Newman, "Building microservices," O'Reilly Media, Inc., 2021.

[2] M. Richards, "Software architecture patterns, 2nd edition," O'Reilly Media, Inc., 2022.

[3] Google Trends, "Explore - Google Trends," [Online]. Available: https://trends.google.com/trends/explore?cat=1227&date=2013-01-01%202023-10-15&q=microservices,monolith&hl=en. [Accessed: Jan. 14, 2024].

[4] M. Kalske, N. Mäkitalo, and T. Mikkonen, "Challenges when moving from monolith to microservice architecture," in ICWE 2017 International Workshops, Rome, Italy, June 5-8, 2017, Revised Selected Papers, vol. 17, Springer International Publishing, 2018, pp. 32-47. Available: https://doi.org/10.1007/978-3-319-74433-9_3. [Accessed: Jan. 14, 2024].

[5] K. Gos and W. Zabierowski, "The comparison of microservice and monolithic architecture," in 2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH), April 2020, pp. 150-153. Available: https://doi.org/10.1109/MEMSTECH49584.2020.9109514. [Accessed: Jan. 14, 2024].

[6] C. Pahl and P. Jamshidi, "Microservices: A Systematic Mapping Study," in CLOSER, vol. 1, 2016, pp. 137-146. Available: https://www.scitepress.org/PublishedPapers/2016/57855/57855.pdf. [Accessed: Jan. 14, 2024].

[7] T. Salah et al., "The evolution of distributed systems towards microservices architecture," in 2016 11th International Conference for Internet Technology and Secured Transactions (ICITST), Dec. 2016, pp. 318-325. Available: https://doi.org/10.1109/ICITST.2016.7856721. [Accessed: Jan. 14, 2024].

[8] J. Fritzsch et al., "Towards an architecture-centric methodology for migrating to microservices," arXiv preprint arXiv:2207.00507, 2022. Available: https://doi.org/10.48550/arXiv.2207.00507. [Accessed: Jan. 14, 2024].

[9] A. Razzaq and S.A. Ghayyur, "A systematic mapping study: The new age of software architecture from monolithic to microservice architecture—awareness and challenges," Computer Applications in Engineering Education, vol. 31, no. 2, pp. 421-451, 2023. Available: https://doi.org/10.1002/cae.22586. [Accessed: Jan. 14, 2024].

[10] D. Faustino, N. Gonçalves, M. Portela, and A.R. Silva, "Stepwise migration of a monolith to a microservices architecture: Performance and migration effort evaluation," arXiv preprint arXiv:2201.07226, 2022. Available: https://doi.org/10.48550/arXiv.2201.07226. [Accessed: Jan. 14, 2024].

[11] J.P. Gouigoux and D. Tamzalit, "From monolith to microservices: Lessons learned on an industrial migration to a web-oriented architecture," in 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), April 2017, pp. 62-65. Available: https://doi.org/10.1109/ICSAW.2017.35. [Accessed: Jan. 14, 2024].

[12] R. Chen, S. Li, and Z. Li, "From monolith to microservices: A dataflow-driven approach," in 2017 24th Asia-Pacific Software Engineering Conference (APSEC), Dec. 2017, pp. 466-475. Available: https://doi.org/10.1109/APSEC.2017.53. [Accessed: Jan. 14, 2024].

[13] M. Seedat, Q. Abbas, and N. Ahmad, "Systematic Mapping of Monolithic Applications to Microservices Architecture," arXiv preprint arXiv:2309.03796, 2023. Available: https://doi.org/10.48550/arXiv.2309.03796. [Accessed: Jan. 14, 2024].

[14] V. Velepucha and P. Flores, "A survey on microservices architecture: Principles, patterns and migration challenges," IEEE Access, 2023. Available: https://doi.org/10.1109/ACCESS.2023.3305687. [Accessed: Jan. 14, 2024].

[15] N. Gonçalves, D. Faustino, A.R. Silva, and M. Portela, "Monolith modularization towards microservices: Refactoring and performance trade-offs," in 2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C), March 2021, pp. 1-8. Available: https://doi.org/10.1109/ICSA-C52384.2021.00015. [Accessed: Jan. 14, 2024].

[16] O. Cico, R. Souza, L. Jaccheri, A. Nguyen Duc, and I. Machado, "Startups transitioning from early to growth phase - a pilot study of technical debt perception," in Software Business: 11th International Conference, ICSOB 2020, Karlskrona, Sweden, Nov. 16-18, 2020, Proceedings 11, Springer International Publishing, 2021, pp. 102-117. Available: https://doi.org/10.1007/978-3-030-67292-8_8. [Accessed: Jan. 14, 2024].

[17] M. Tsechelidis, "Developing distributed systems with modular monoliths and microservices," 2023. [Online]. Available: http://dspace.lib.uom.gr/handle/2159/29357. [Accessed: Jan. 14, 2024].

[18] M. Fowler, "Monolith First," 2015. [Online]. Available: https://www.martinfowler.com/bliki/MonolithFirst.html. [Accessed: Jan. 14, 2024].

[19] S. Newman, "Microservices for Greenfield?" 2015. [Online]. Available: https://samnewman.io/blog/2015/04/07/microservices-for-greenfield/. [Accessed: Jan. 14, 2024].

[20] S. Tilkov, "Don’t start with a monolith when your goal is microservices architecture," 2015. [Online]. Available: https://www.martinfowler.com/articles/dont-start-monolith.html. [Accessed: Jan. 14, 2024].

[21] T. Shablii and S. Tytenko, "Modular Monolith as a Microservices Precursor," Modern engineering and innovative technologies, vol. 29-01, pp. 25-32, 2023. Available: https://doi.org/10.30890/2567-5273.2023-29-01-038. [Accessed: Jan. 14, 2024].

[22] E. Evans, "Domain-Driven Design: Tackling Complexity in the Heart of Software," O'Reilly Media, Inc., 2003.

[23] V. Vernon, "Implementing Domain-Driven Design," O'Reilly Media, Inc., 2013.

[24] V. Khononov, "Learning Domain-Driven Design," O'Reilly Media, Inc., 2021.

[25] A. Cockburn, "Hexagonal Architecture," 2005. [Online]. Available: https://alistair.cockburn.us/hexagonal-architecture/. [Accessed: Jan. 14, 2024].

[26] T. Hombergs, "Get Your Hands Dirty on Clean Architecture, 2nd edition," O'Reilly Media, Inc., 2023.

[27] R. C. Martin, "Agile Software Development: Principles, Patterns, and Practices," Pearson, 2003.

[28] T. Shablii, "Opora API," [Online]. Available: https://tarasshablii.github.io/opora-api/. [Accessed: Jan. 14, 2024].

[29] T. Shablii, "tarasshablii/opora-api," GitHub, [Online]. Available: https://github.com/tarasshablii/opora-api. [Accessed: Jan. 14, 2024].

[30] T. Shablii, "tarasshablii/honeycomb-monolith," GitHub, [Online]. Available: https://github.com/tarasshablii/honeycomb-monolith. [Accessed: Jan. 14, 2024].

[31] T. Shablii, "opora-api.yml," GitHub, [Online]. Available: https://github.com/tarasshablii/honeycomb-monolith/blob/main/openapi/opora-api.yml. [Accessed: Jan. 14, 2024].

[32] OpenAPI Generator, [Online]. Available: https://openapi-generator.tech/. [Accessed: Jan. 14, 2024].

[33] PostgreSQL, "PostgreSQL: The World's Most Advanced Open Source Relational Database," [Online]. Available: https://www.postgresql.org/. [Accessed: Jan. 14, 2024].

[34] MongoDB, Inc., "MongoDB," [Online]. Available: https://www.mongodb.com/. [Accessed: Jan. 14, 2024].

[35] MinIO, Inc., "MinIO | High Performance, Kubernetes Native Object Storage," [Online]. Available: https://min.io/. [Accessed: Jan. 14, 2024].

[36] Docker, Inc., "Empowering App Development for Developers | Docker," [Online]. Available: https://www.docker.com/. [Accessed: Jan. 14, 2024].

[37] JUnit, "JUnit 5," [Online]. Available: https://junit.org/junit5/. [Accessed: Jan. 14, 2024].

[38] Mockito, "Mockito framework site," [Online]. Available: https://site.mockito.org/. [Accessed: Jan. 14, 2024].

[39] AssertJ, "AssertJ - Fluent assertions java library," [Online]. Available: https://assertj.github.io/doc/. [Accessed: Jan. 14, 2024].

[40] Testcontainers, "Testcontainers," [Online]. Available: https://testcontainers.com/. [Accessed: Jan. 14, 2024].

[41] Oracle Corporation, "Java Platform Module System," [Online]. Available: https://openjdk.org/projects/jigsaw/spec/. [Accessed: Jan. 14, 2024].

[42] Spring, "Spring Modulith," [Online]. Available: https://spring.io/projects/spring-modulith/. [Accessed: Jan. 14, 2024].

[43] T. Shablii, "application.properties," in honeycomb-monolith, GitHub, [Online]. Available: https://github.com/tarasshablii/honeycomb-monolith/blob/main/monolith/src/main/resources/application.properties. [Accessed: Jan. 14, 2024].

[44] ArchUnit, "ArchUnit," [Online]. Available: https://www.archunit.org/. [Accessed: Jan. 14, 2024].

[45] T. Shablii, "ArchUnitTest.java," in honeycomb-monolith, GitHub, [Online]. Available: https://github.com/tarasshablii/honeycomb-monolith/blob/main/monolith/src/test/java/dev/tarasshablii/opora/monolith/ArchUnitTest.java. [Accessed: Jan. 14, 2024].

[46] T. Shablii, "ArchUnitTest.java," in honeycomb-monolith, GitHub, [Online]. Available: https://github.com/tarasshablii/honeycomb-monolith/blob/main/microservices/initiati

ves/src/test/java/dev/tarasshablii/opora/microservices/initiatives/ArchUnitTest.java. [Accessed: Jan. 14, 2024].

[47] T. Shablii, "InitiativesProvider.java," in honeycomb-monolith, GitHub, [Online]. Available: https://github.com/tarasshablii/honeycomb-monolith/blob/main/microservices/api-gateway/src/main/java/dev/tarasshablii/opora/microservices/apigateway/provider/InitiativesProvider.java. [Accessed: Jan. 14, 2024].

[48] T. Shablii, "InitiativesController.java," in honeycomb-monolith, GitHub, [Online]. Available: Available: https://github.com/tarasshablii/honeycomb-monolith/blob/main/microservices/initiatives/src/main/java/dev/tarasshablii/opora/microservices/initiatives/endpoint/rest/InitiativesController.java. [Accessed: January 14, 2024].