International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 12 Issue: 03 | Mar 2025 www.irjet.net p-ISSN: 2395-0072

Transforming Legacy Monolith Architecture: A Pragmatic Approach to Modulith Architecture Adoption

Renjith Ramachandran[1]

[1]*Independent Researcher and Solutions Architect, Hudson, MA, USA*

Abstract - *Monolithic applications have been a part of software development since its inception and gained prominence with the rise of web applications. As applications grew more complex, companies began adopting microservices architecture for greater agility, scalability, and faster release cycles. Over the last decade, microservices architecture saw widespread adoption. With the advent of containerization and orchestration tools like Kubernetes, many organizations transitioned from monolithic applications to microservices. However, some companies eventually found themselves dealing with an unmanageable number of microservices, prompting them to consider returning to monolithic systems or opting for a middle ground, known as Modulith architecture.*

Migrating a legacy application from an outdated architecture to a modern one requires a significant investment. Without proper planning and execution, such migrations can often result in failure. The purpose of the paper is to present a strategy for building an architecture and migration strategy for Modulith application, addressing key aspects such as unit and integration testing of modules, common logging patterns, exception handling, persistence isolation, domain modularization, inter-module communication, module security, deployment strategies, configuration management, DevOps, and code maintenance. The goal of the paper is to create a robust framework that is extensible, maintainable, detachable, and secure. This approach ensures the avoidance of the common pitfalls often associated with monolithic architectures.

Key Words: **Modulith Architecture, Microservices Architecture, Monolithic Architecture, Modernization, Legacy Systems**

1.INTRODUCTION

A legacy system is a software system that, over time, has become critical to business operations but has reached a point where maintenance is difficult, and it resists modifications [4]. Due to its original design, the system continues to operate and support business functions but lacks the flexibility to accommodate changes. Debugging issues can take hours, and upgrading the system would be highly costly.

Companies take the path of software modernization to make the systems current from a technical stack perspective. Software modernization is the process of rearchitecting or replacing existing software systems with an architecture which is more flexible and cost effective [5]. Certain modernizations take years and it takes many more years to reap the benefits of the modernization initiative.

Monolithic applications have been a fundamental part of software development since its early days. With the widespread adoption of web applications, monolithic architectures gained significant popularity. They were typically built using technologies like JEE, .NET, or PHP. However, scaling and managing monolithic applications are challenging due to their large size and the single codebase shared among all developers [1]. Over the years lot of functionalities and unstructured code gets added to Monolith applications that they became resistant to modifications or costly to maintain because of licensing issues. As a result, many companies begin transitioning to more flexible architectures like Microservices Architecture (MSA).

The period from 2010 to 2020 witnessed a significant rise in the prevalence of Information Technology, with Digital and Digital Transformation emerging as hot topics by the middle of the decade. Digital transformation involves the integration of new Digital technologies like social, mobile, analytics and cloud [2][3]. One of the key advancements brought by Digital Transformation was the enhancement of scalability and agility in application delivery. With growth as a priority, businesses seek faster feature deployment and highly scalable applications.

Microservices Architecture (MSA) can effectively meet these requirements. A study by Taibi et al. highlights that key motivations for migrating to MSA include maintainability, scalability, delegation of team responsibilities, and DevOps support [6]. These factors, together, enhance the development process. Companies started developing microservices for specific functionalities, enabling different teams to work on services concurrently, thus accelerating time-to-market.

As the number of microservices increased, managing them became increasingly difficult. Some services even evolved into new monoliths due to their growing complexity. Companies started thinking about a common ground between monoliths and microservices.

2. LEGACY APPLICATION

Over time, a legacy system loses its appeal due to the characteristics listed below:

Characteristics of a Legacy Application

A system can be classified as legacy based on the following attributes.

- *Business Value:* Business value expresses the extent to which system is essential to the business of an Organization.
- Decomposability: Decomposability measures how easily main components of the software system are identifiable and independent from each other.
- Obsolescence: Obsolescence refers to the aging of a software system because of the failure to meet the changing needs.
- Deterioration: Deterioration refers to the aging of a software system because of the continuous changes that are made. [7]

3. MONOLOTHIC ARCHITECTURE

3.1 Advantages of a Monolithic Architecture

Monolithic applications consist of a single codebase, making them simpler to build and deploy. Local setup is easier since there is only one codebase to configure. Monolithic systems enable faster communication between internal components, as all components are part of the same application, reducing overhead and simplifying cross-component testing. This architecture is well-suited for smaller applications [8].

3.2 Disadvantages of Monolithic Architecture

Monolithic applications are often challenging to scale. While vertical scaling is possible, horizontal scaling presents significant difficulties. The tight coupling within a complex monolithic application makes it hard to implement changes. As the application grows, testing new changes becomes more cumbersome, and deployment times increase due to the application's size. Additionally, adopting new technologies within a monolithic structure can be problematic [8]

4. MICROSERVICES ARCHITECTURE

4.1 Advantages of a Microservices Architecture

Microservices architecture is highly flexible, reliable, and enhances developer productivity. It allows for both horizontal and vertical scaling and enables faster time to market. Additionally, it offers the flexibility to use different technologies across various services [8].

4.2 Disadvantages of Microservices Architecture

Daniel et al. studied and collected disadvantages of microservices from different sources and came up with the following top 5 disadvantages [9].

- *Operational complexity*-multiplicity of autonomous services requires intensive management and co-ordination
- *Initial Cost of Adoption*-initial setup would require specific technical skills, tools and processes.
- *Data Consistency*-As each service maintains its own data, data consistency between service becomes an issue.
- *Network Overhead*- As services will be constantly communicating, there can be degradation in network performance.
- *Increasing Complexity of DevOps*-managing build and deployment of multiple services increases the complexity of the whole ecosystem.

5. MODULITH ARCHITECTURE

A Modulith application serves as a middle ground between a monolithic application and a microservices application. In the Figure 1 below, Modulith Application sits between Monolith and Microservices applications. Monolith represents a single application and microservices represent each individual application represented as Microservice. In Modulith application, a single application is split in to individual modules. When designing a Modulith application, care should be taken to avoid it becoming another monolith. At the same time, it should be structured in a way that allows individual modules to be easily detached and transitioned into microservices if needed.

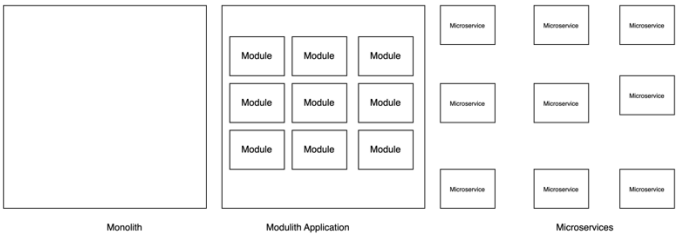

Fig. 1. Monolith vs Modulith vs Microservices

5.1 Advantages of Modulith Application

Modulith architecture offers several advantages over both microservices and monolithic architectures, provided it is designed and maintained correctly. These benefits include the absence of distributed system complexities, clear separation of concerns between modules, simplified testing, reduced operational and DevOps complexity, and greater development consistency.

5.2 Disadvantages of Modulith Application

A Modulith application can have following disadvantages

- Defining and maintaining modular boundary will be challenging
- Managing dependencies between modules can be challenging
- It will be challenging to decide when to transition from Modulith to microservices architecture

The whole Modulith application will have to be scaled if one of the modules is often or more frequently used

6. ARCHITECTING MODULITH APPLICATION

The first step in architecting Modulith Architecture is defining the modules and purpose of the module. Modules should have a defined purpose and should be linked to a domain the overall application handle. Modules should be designed such that it can be used by multiple applications.

Module Definition

Module Name	Module Definition			
	Purpose	I/O	Functions	Entities
Module1	Purpose of module	Module Inputs & Outputs	Module Functions	Module entities

When defining modules, it is crucial to outline the functions the module will perform, along with a high-level description of the inputs, outputs, and associated entities. This model should be created in collaboration with a broader team and maintained as a living document, updated whenever new functionality is added to the module. This approach helps to clearly define and establish the module's boundaries, while also preventing the implementation of duplicate functionalities.

Inputs and outputs define, at a high level, the list of fields that a module can accept as input or generate as output. Functions outline the list of functionalities provided by the module, while entities represent all the data models managed by the module. In cases where a module shares entity with other modules, those entities should be placed in a common module. A module may be maintained by a single team or multiple teams. Changes to the module, depending on its owner, should be properly documented and versioned to ensure consistency and trackability.

Module Name	**Module Ownership**			
	Owner	***Version***	***Changes***	***Review Sign-Off***
Module1	First Name Last Name	V1.1	Implemented X functionality	Module entities

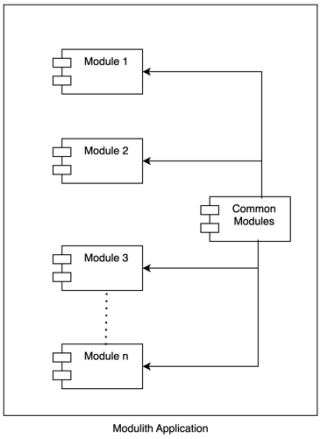

Fig. 2. Modulith Application

6.1 Common Modules of a Modulith Architecture

Figure 2 shows the high-level structure of a Modulith application. A Modulith application can comprise multiple modules. As previously mentioned, each module should have a clear purpose and defined boundaries, established through the outlined process.

Some functionalities, such as exception handling, logging, auditing, and configuration management, are common across modules. These reusable functionalities should be bundled into a common module and shared across all

modules to ensure consistency and a unified pattern in their usage.

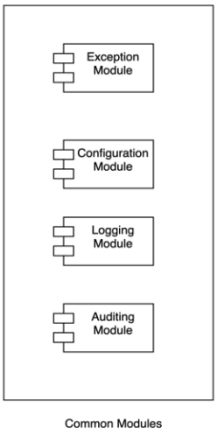

Fig. 3. Common Modules

Figure 3 shows a sample Commons module. This module should be packaged and hosted as a separate component, allowing other Modulith applications to utilize it as well. Figure 4 shows modules reusing common modules.

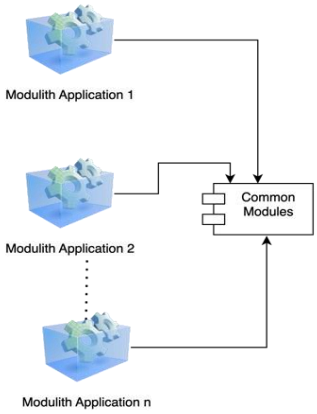

Fig. 4. Reusing Common Modules

- ***Exception Module:*** All exceptions or errors thrown by the modules should follow a standardized format, allowing the invoking modules to handle errors or exceptions in a consistent manner. This ensures easier error management and promotes uniformity across the system. Figure 5 below shows a high-level hierarchy of the Exceptions

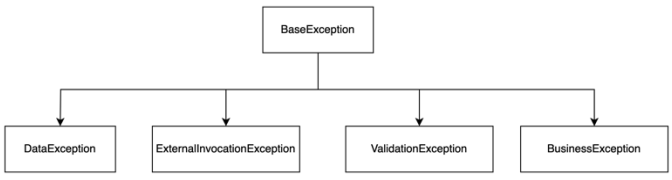

Fig. 5. Exception Hierarchy

Exception/Error definitions

#	Exception Definition	
	Exception Name	***Description***
1	BaseException	All Exception extend base Exception
2	ExternalInvocationException	All errors/exceptions while invoking external systems using any protocol is encapsulated in this exception. This can be extended for each system
3	ValidationException	Errors/exceptions thrown when validations fail on the input fields
4	BusinessException	Errors/exceptions thrown when Business Rule validations fail

TABLE III shows definition of high level exceptions. It's a basic list but can be extended based on the use case.

- ***Configuration Module:*** The module stores all the configuration information necessary for its operation. This configuration data can be loaded from an external configuration system, a database, or a configuration file within the application. There should be a provision to refresh the configurations in real time. The configurations will be segregated by module to ensure proper organization and management.

- ***Logging Module:*** The module ensures that all logging statements adhere to a defined format, making it easier to trace issues. The module should follow the following pattern:

<DateTime>:<TraceId>:<Module-Name>:<Class Name>:<Method Name><Log Statement>

- ***Auditing Module:*** The module ensures that auditing is performed in a consistent format across the application. Uniform auditing simplifies the process of grouping records and tracing issues, making it easier to identify and resolve problems.

6.2 Intermodule Communication

Modules frequently communicate to utilize the specific functionalities they expose. Each module has a defined set of functionalities and interfaces that are available for other modules to consume. In a modulith framework, it is crucial that modules remain loosely coupled, allowing a module to be transitioned into an independent service without affecting the modules it

interfaces with. Communication between modules can occur either synchronously or asynchronously. In synchronous communication, the invoking module waits for a response from the invoked module before continuing execution.

To achieve loose coupling, the approach illustrated in Figure 6 involves using a mediator or connector to transform requests and responses into the format required by the module. This allows for flexibility and ensures that modules can communicate without being tightly bound to one another's specific formats.

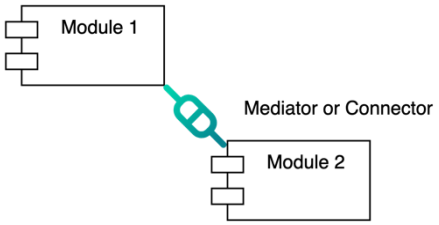

Fig. 6. Synchronus Intermodule Communication

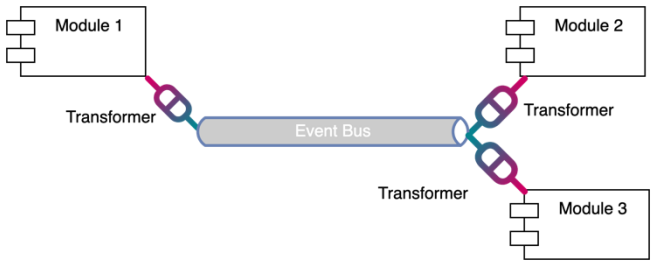

Fig. 7. Asynchronus Intermodule Communication

As illustrated in Figure 7, in the case of asynchronous communication between modules, the source module will send a message transformed by a transformer to an event bus. The destination module(s) will then consume the message after performing another transformation. The source module does not wait for a confirmation. However, it may receive confirmation asynchronously through the same event-driven pattern.

6.3 Unit Testing and Integration Testing Modules

Each module should have its own independent unit test cases, which are directly linked to the module and can be executed with little to no dependency on other modules. As shown in Figure 8, these unit test cases are embedded within each module, allowing for independent testing. To achieve this, unit tests may need to mock external interfaces, database calls, and invocations of other modules.

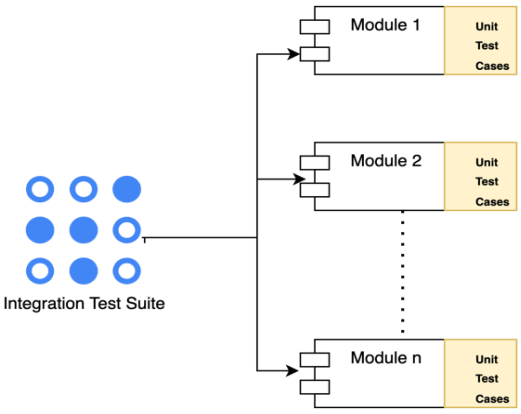

Fig. 8. Testing Modules

As illustrated in Figure 8, it would be advantageous to have an Integration Test Suite to test scenarios involving multiple modules. This suite could be an extension of the existing Application Test Suite. Such a framework would allow for testing interactions between different modules. However, further details on the Integration Test Suite are beyond the scope of this document.

6.4 Securing Modules

Modules must be secured to ensure that their exposed functionalities are not compromised. The application may already have a security framework in place that modules can inherit. As illustrated in Figure 9, in addition to the overall security provided by the application, each module can implement its own security measures as an extension of the application's security framework.

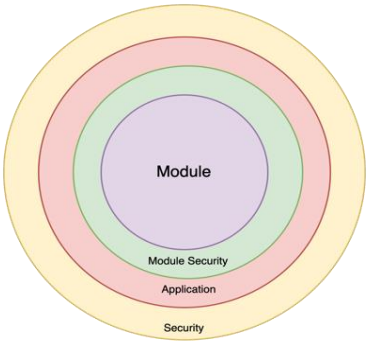

Fig. 9. Module Security

Module level security: An approach to accept requests only from registered modules can be implemented by requiring any module that wishes to interact with another module to register and obtain a token or credential. This credential must be passed along with the base security tokens when interacting with the module. This method promotes loose coupling between modules and simplifies the process of

detaching a module to convert it into a microservice when necessary.

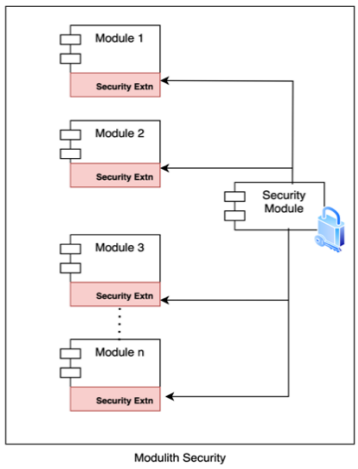

Fig. 10. Module Security

The application will utilize a common security mechanism such as OAuth 2.0 [12]. When one module invokes another, the tokens provided by the base application are passed to the invoked module to authorize the incoming request. As illustrated in Figure 8, each module will have a Security Extension that interacts with the Security Module to validate these tokens. Like common modules, the security module will also be reused throughout the system. Following the separation of concerns design model [11], the security module will manage tasks such as validating and authorizing tokens. While each module focuses on its core functionality, the security module handles all security-related aspects. This ensures that any updates to the security mechanism do not affect individual modules.

6.5 Database Resiliency

Modulith architecture can have different database resiliency methods depending upon the criticality of the application.

One Database with set of tables for each module: All modules share the same database, with each module having its own set of tables. While this pattern works well for simpler applications, it presents challenges as the modules begin to share tables, eventually leading to a monolithic structure. Additionally, any issues with the database can affect the entire application. Figure 11 illustrates how modules are organized with separate sets of tables.

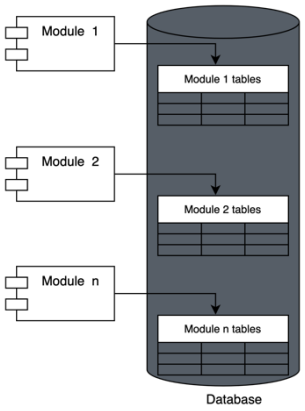

Fig. 11. Modules having separate tables

One Database with different schemas for modules: All modules use the same database, but each module operates within its own schema, ensuring no sharing of tables between modules. The primary drawback of this pattern is that any issue with the database can affect the entire application. Figure 12 illustrates how modules are organized with different schemas.

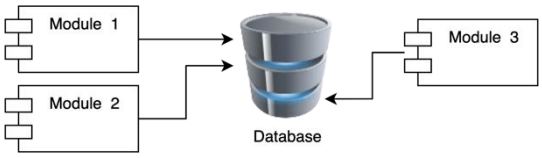

Fig. 12. Modules having different schema's

Each module with different databases: Each module in the modulith application can utilize a different database, with the choice of database being determined by the functionality that the module supports. Figure 13 shows sample modules with various databases. However, a key disadvantage of this approach is the potential for data consistency issues between the systems.

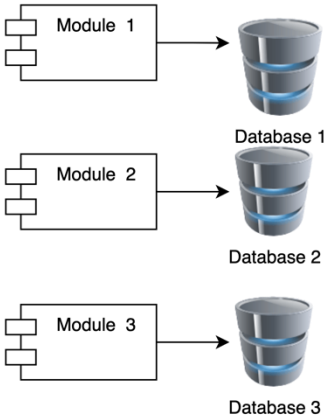

Fig. 13. Modules having different database's

6.6 Documentation of the Module

Each module should include detailed documentation outlining the functionalities it provides and how those functionalities can be consumed. Tools can be utilized to automatically generate this documentation. This would assist developers from other modules or applications in easily integrating the module. Additionally, documenting the internal workings of the modules would also be beneficial for future maintenance and development.

6.7 Code Maintenance, DevOps and Deployment

Module code can be maintained by either a single team or a group of teams. It is essential to establish an agreed-upon structure and reach a consensus on how the module's code will be organized. Each module should be maintained as a separate project within the code repository, allowing any application that requires it to import and use the module. Initially, a module may be integrated into the main application, but as demand for the module grows, it can eventually be separated into its own service. After the build process, modules should function as independent units that can be aggregated with other modules and deployed together as a single unit. For instance, in the Java ecosystem, each module should be built as a Java Archive (JAR) file and embedded into a Web Archive (WAR) file, which can then be deployed and scaled as needed.

Both the application code and its modules can be containerized, deployed, and scaled. However, not all modules in a Modulith application will experience the same traffic patterns. Despite this, the entire application must be scaled, even if only one module is experiencing the majority of the traffic.

7. Detaching a module

As a Modulith application grows and becomes more complex, the number of modules will increase. Some modules will undergo significant changes, while others may remain largely unchanged or experience minimal modifications. One of the disadvantages of the Modulith architecture, as mentioned, is that the entire application must be scaled when a single module, often referred to as a “Hot Module,” is frequently or heavily used. When a module reaches this level of demand, it should be detached and transformed into a separate service. Another scenario arises when multiple applications begin relying on a specific module, leading to frequent requests for updates. In this case, it would also be advantageous to move the module into a separate service to handle the growing demands more efficiently.

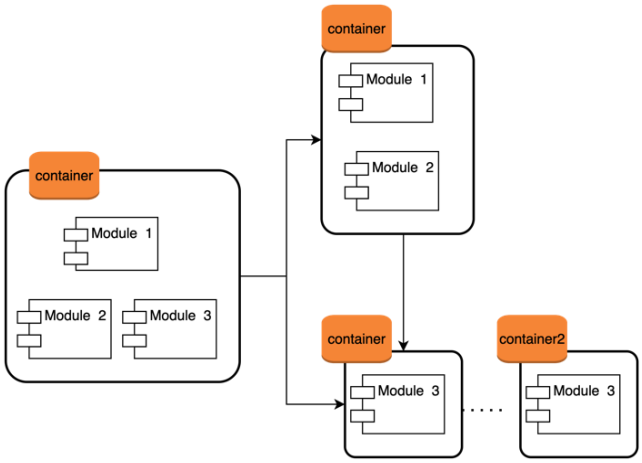

Fig. 14. Module detached as a separate container during runtime

8. Future Scope

One of the drawbacks of the Modulith architecture is that the entire application must be scaled due to "hot modules" within the system. A module may consistently become a hot spot or experience high traffic on specific days of the year. This would necessitate extracting the module into a separate service and modifying the core application. The future goal is to enable the dynamic detachment of modules as separate services at runtime, as outlined in Google’s Service Weaver framework [10], allowing them to scale based on traffic patterns. Figure 14 illustrates the process of separating Module 3 into its own container during runtime, depending on traffic demand.

9. CONCLUSIONS

The concept of a Modulith emerged to address the challenges of both monolithic and microservices architectures. To preserve the integrity of a Modulith architecture, it is essential to establish a clear process before developing any new modules. Before making changes to a module or group of modules within a project, the team must obtain approval from the module owner. By adhering to defined processes and procedures, a strong Modulith architecture can be maintained, provided there is collaboration and flexibility among all teams involved in the development of the Modulith application.

REFERENCES

[1] Villamizar, M., Garces, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., & Gil, S. (2015). Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. *2015 10th Computing Colombian Conference (10CCC)*.

https://doi.org/10.1109/columbiancc.2015.7333476 p.583-585

[2] Schwertner, K. (2017). Digital Transformation of Business. *Trakia Journal of Science*, *15*(Suppl.1), 388–393. https://doi.org/10.15547/tjs.2017.s.01.065

[3] Mydyti, H., Ajdari, J., & Zenuni, X. (2020, September). Cloud-based Services Approach as Accelerator in Empowering Digital Transformation. In *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)* (pp. 1390-1396). IEEE.

[4] K. Bennett. Legacy systems: Coping with success. IEEE Software, 12(1):19–23, 1995.

[5] Khadka, R., Batlajery, B. V., Saeidi, A. M., Jansen, S., & Hage, J. (2014). How do professionals perceive legacy systems and software modernization? *Proceedings of the 36th International Conference on Software Engineering*. https://doi.org/10.1145/2568225.2568318

[6] Taibi, D., Lenarduzzi, V., & Pahl, C. (2017). Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. *IEEE Cloud Computing*, *4*(5), 22–32. https://doi.org/10.1109/mcc.2017.4250931

[7] Cimitile, A., Fasolino, A. R., & Lanubile, F. (2001). Legacy systems assessment to support decision making. In *IEEE Workshop on Empirical Studies of Software Maintenance* (pp. 145-150)

[8] ELGHERIANI, N. S., & AHMED, N. A. S. (2022). Microservices vs. Monolithic architectures [The differential structure between two architectures]. *Minar International Journal of Applied Sciences and Technology*, *3*(6).

[9] dos Santos Krug, D., Chanin, R., & Sales, A. Exploring the Pros and Cons of Monolithic Applications versus Microservices.

[10] Johnson, J., Kharel, S., Mannamplackal, A., Abdelfattah, A. S., & Cerny, T. (2024). Service Weaver: A Promising Direction for Cloud-native Systems?. *arXiv preprint arXiv:2404.09357*.

[11] De Win, B., Piessens, F., Joosen, W., & Verhanneman, T. (2002, November). On the importance of the separation-of-concerns principle in secure software engineering. In Workshop on the Application of Engineering Principles to System Security Design (pp. 1-10).

[12] Jones, M., & Hardt, D. (2012). The oauth 2.0 authorization framework: Bearer token usage (No. rfc6750)

BIOGRAPHIES

Renjith Ramachandran received his Bachelor's Degree in Electronics and Communications Technology from India and his Master's Degree in Computer Science from the US. He spent 12 years as a consultant, taking on various roles from Software Engineer to Architect, and working with clients in industries such as Telecom, Banking, and Insurance. He currently serves as a Solutions Architect, with research interests that focus on software architectures, emerging technologies, and the development of innovative tools and frameworks.