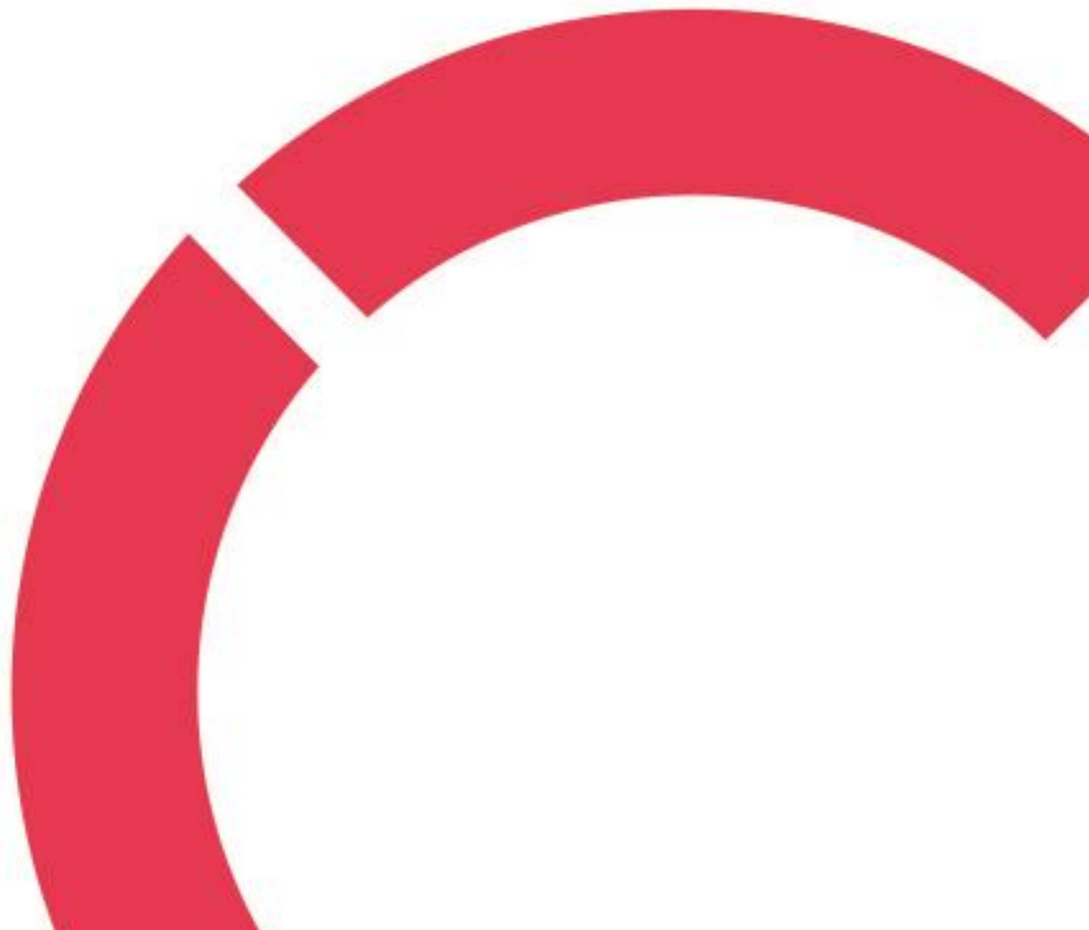
Deepsikha Shrestha

NAVIGATING SOFTWARE ARCHITECTURE: EVALUATING MONOLITHIC ARCHITECTURES IN MODERN DEVELOPMENT

Thesis
CENTRIA UNIVERSITY OF APPLIED SCIENCES
Business Intelligence Technology
May 2024



Centria University of Applied Sciences

ABSTRACT

Centria University of Applied Sciences	**Date** May 2024	**Author** Deepsikha Shrestha
Degree programme Business Intelligence Technology		
Name of thesis NAVIGATING SOFTWARE ARCHITECTURE: EVALUATING MONOLITHIC ARCHITECTURES IN MODERN DEVELOPMENT		
Centria supervisor Henry Paananen, Jari Isohanni		**Pages** 29+ 3

The attainment and scalability of a project are substantially influenced by the architectural framework selected within the dynamic realm of software development. Monolithic architectures represent one among several architectural paradigms that have endured over time, constituting a traditional approach to software system development. However, the usefulness and significance of monolithic designs are being reassessed in light of the emergence of microservices and other contemporary architectural styles.

Within the framework of contemporary software development, this thesis offers a thorough examination of monolithic architectures. This thesis conducts a thorough analysis to look at the benefits, drawbacks, and trade-offs of monolithic architectures, taking into account factors such as deployment complexity, maintenance overhead, scalability, and development speed. In order to give the insight into when monolithic architectures might still be a good option and when alternatives might be more beneficial, the examinations of real-world case studies and industry best practices are to be done in the thesis.

In addition, this thesis approaches to modernize and evolve current monolithic systems, such as refactoring, containerization, and modularization. This thesis work is to provide software architects and developers with the information and resources it required to make wise decisions while designing and developing software systems in the fast-paced world of today. It does this by emphasizing the potential and problems that come with monolithic designs.

Key words
CSS, HTML, Java, JavaScript, Maven, Microservices, Modular Architecture, Monolithic Architecture, Software Architecture, Software Development, Spring Boot, TomCat

ABSTRACT

CONTENTS

1 INTRODUCTION..1

2 DIFFERENT DESIGN ARCHITECTURES 4

2.1 Monolithic Architecture 5

2.2 Layered Architecture..6

2.3 Client-server architecture 8

2.4 Microservices Architecture..10

2.5 Event-Driven Architecture 12

2.6 Modular Architecture 13

3 COMPARISON BETWEEN MONOLITHIC, MICROSERVICES, AND MODULAR ARCHITECTURE..17

3.1 Structure..18

3.2 Development..19

3.3 Deployment...20

3.4 Scalability 21

3.5 Flexibility...22

3.6 Complexity..22

3.7 Suitability...23

4 CONCLUSION..25

REFERENCE 27

1 INTRODUCTION

The software development industry is continually evolving due to the rapid appearance of new techniques and technologies. One of these innovations that has sparked a debate is the choice between monolithic and microservices architecture. Software development was mostly dependent on monolithic architecture since the 1970s, which was a traditional approach with a single, unified codebase. On the other hand, microservices architecture has gained popularity since about the last ten years.

Monolithic design has several advantages, including simplicity, scalability, and ease of development. In general, monolithic app design and implementation are simpler, especially for smaller applications. In order to handle increasing workloads, it may also be easily horizontally expanded by adding additional servers. Furthermore, the integrated nature of a monolithic codebase often simplifies testing and debugging processes. While there are benefits to monolithic architecture, its drawbacks increase with larger and more intricate software systems. The degree to which the component parts are interconnected is a major disadvantage. Because the components are interconnected, it is challenging to update or modify one without also affecting the others, which increases the complexity of the code and maintenance expenses (Awati & Wigmore, 2022).

Moreover, monolithic program deployment and updates can be difficult and time-consuming. Because all of the components are housed within a single codebase, updates to any one of them must be done at the same time to avoid disrupting and causing downtime for the program as a whole. Furthermore, monolithic systems experience scalability problems beyond a certain size. Further growth is challenging to maintain because network latency and resource constraints restrict the efficiency of horizontal scalability (Newman, 2023b).

The development of programs can be done more distributedly and modularly with the help of microservices architecture. It breaks up an application into more manageable, independent services that communicate with one another through well-defined interfaces. This microservice-based approach has several benefits over monolithic design, including increased scalability, fault isolation, and agility (Fowler, 2015).

As the Microservices architecture allows faster development, testing, and deployment cycle, the majority of the applications currently are made using the Microservices architecture. Besides the quicker DevOps cycle, these applications are also able to adapt to the changing requirements quickly as per the need of the user. Since each component of a microservice is independent of other services, the bugs, or issues in any one of the services do not affect other services, causing the system to be resilient. Also, since each microservice can be scaled independently and individually, it allows efficient resource management. Also, it provides an effective approach to handle the growth and scalability of the applications. This architecture of software development is generally beneficial to the apps with varying service requirements (Newman, 2023a).

On the other hand, the monolithic architecture is particularly beneficial to simple as well as easy applications, requiring minimal changes. The need of the monolithic architecture is appropriate depending upon the needs, requirements, scale, and the complexity of the application. Due to its simplicity and ease of implementation, monolithic architecture is an excellent choice for smaller and to-the-point applications. Larger applications with sophisticated designs and features on the other hand are likely to be benefitted from the extra flexibility, scalability, and agility that the microservices architecture offers (Cervantes & Kazman, 2016).

This thesis aims to examine the complexities in monolithic architecture design to other contemporary software design architectures. This thesis aims to establish a notion that no matter the changes in the design patterns of software development, monolithic architecture still has a powerful role to play in case of the developments of the application in the current times. It examines the benefits and limitations of this architecture along with contemporary architectures as well as its suitability on developing various kinds of applications. The historical significance of monolithic systems is reflected in their evolution within software development processes. Monolithic systems, which consolidated the entire program into a single deployment unit and codebase, were once the most common architectural approach. It was easier to develop, test, and implement simpler applications with this unified approach. However, monolithic systems have been experiencing more issues as a result of user demands, the exponential growth in data volume, and the requirement for more scalability and agility (Skotnický, 2023).

Architectural environments have undergone a significant shift with the rise in distributed and cloud-native systems in the software industry. Scalability, adaptability, and the capacity to quickly adjust to evolving requirements and technology breakthroughs are essential for modern applications. Whether monolithic systems can satisfy these requirements in this case has been questioned. However, with the

pros of monolithic architecture, there are many more challenges that affect the use and reputation of the monolithic architecture in the current era of modern software development. Hence, it requires a careful consideration and evaluation of their suitability and sustainability in designing the software projects over an extended period of time (Bellemare, 2020).

In the present times, the software industry is dynamic and constantly evolving, causing modern approaches to software development closely examining the use of “outdated" monolithic architecture as the factors such as durability, scalability, and agility are gaining higher importance rather than simplistic design patterns. Even though quite a large number of software developers still have a liking to the monolithic approach to software design, the use of the modern architectural design patterns is becoming more popular due to the high importance placed on the factors discussed above. This study, in contrast to other models, looks at the benefits and drawbacks of monolithic systems within the framework of modern development (Martin, 2017).

Traditional frameworks, such as the monolithic model, need to be reviewed in light of the rapid advancements in programming improvement techniques. Although these unified, integrated frameworks ease of use and quick development have generally been advantages, it is becoming more and more obvious that these frameworks have limitations of their own. Microservices and cloud-based models have gained popularity as substitutes because it is more robust, flexible, and adaptable. Nevertheless, it is unclear whether monolithic architecture is still reasonable given the current state of technology and, if so, under what conditions.

The cutoff points of monolithic models become more obvious given the demands of modern software, which is a significant test. Fast innovation times and early ease of use are two advantages of these frameworks; nonetheless, their centralization limits their adaptability, execution, and flexibility. These issues may make it more difficult for businesses to provide programming that satisfies the needs of modern customers and adjusts to shifting economic conditions.

Therefore, considering recent developments, it is imperative to critically assess the feasibility of monolithic systems. Despite the technical aspects of the plan, this evaluation should consider the organization new requirements, the salient features of the application under development, and the available resources. Businesses can select the optimal architecture for their specific development projects by carefully balancing the benefits and drawbacks of unique designs in comparison to monolithic systems.

2 DIFFERENT DESIGN ARCHITECTURES

The term software architecture defines the outline used in the creation, structurization, organization and behavior of the system. The software architecture acts as a blueprint before creating a system. This basic structure of a software system plays a major impact on the system capacity to scale, sustain and adjust to the changing requirements of the system. The components such as modules, layers, and interfaces play a crucial role in guaranteeing the smooth operations and distinct roles while creating a system. Several architectural patterns help build a software, with each architecture having its own set of benefits and drawbacks which provide a broad framework for arranging various components in an SDLC environment. Architectures such as monolithic, modular, and microservices play a significant role in SDLC as it impacts the system security, scalability, performance, and dependability eventually (Dooley, 2011).

In modern times, the foundation of any system architecture is scalability, which allows it to effectively manage growing loads on the system. Use of a scalable architecture minimizes the bottlenecks and maximizes resource consumption in the system. Along with scalability, scalable architecture helps to accommodate increasing demands using techniques like vertical as well as horizontal scalability. In addition to scalability, maintainability–which emphasizes clean code, modularization, and documentation is highly in demand which promotes the ease of understanding, modification, and extension of the application is crucial for the long-term success of the application. By providing support to open standards and flexible design patterns, an adaptable architecture provides further improvement in resilience and makes it possible to integrate modern technologies and changing business needs with ease (Kleppmann, 2023).

System design architecture also incorporates security considerations along with making the system faster. To protect against the potential threats, weakness, and failures, the architectural design patterns help incorporate strong security features such as rate limitation, encryption, as well as authentication. It also helps to include fault tolerance and reliability measures which are placed in the system to guarantee continuous functioning of the application in the event of hostile assaults, possible attacks, software bugs, or hardware malfunctions. Thus, a well-designed system architecture is an essential component to software engineering excellence as it provides not only an immediate functionality of the system, but also establishes the foundation for expansion and innovation in the future as per the need. Any software system architecture choice determines its fundamentals, including its structure, organization, and behaviour. A system capacity for scalability, maintainability, and adaptability is determined by the architecture choice (Abott & Fisher, 2017).

2.1 Monolithic Architecture

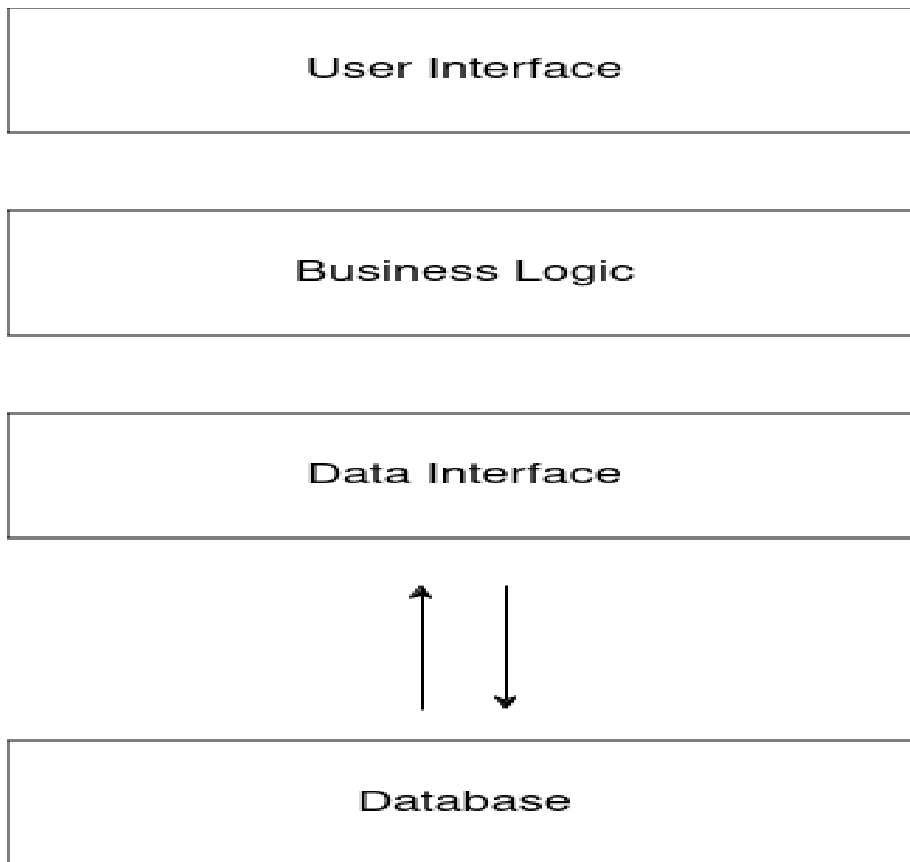


Figure 1 Monolithic Architecture (Ingeno, 2018)

In a monolithic architecture, the entire application is built as a single, indivisible unit. All components and modules are tightly interconnected and run in a single process. Monolithic architectures are characterized by a unified codebase and a single deployment unit (Awati & Wigmoret, 2022). The entire program is created, implemented, and maintained as a single, coherent unit inside a single codebase under the monolithic architecture paradigm. As shown in FIGURE 1, this architecture components are connected, frequently using the same database and codebase. But because growing a monolith requires reproducing the entire application, which can be laborious and resource-intensive, tight coupling can pose scalability issues. Because of its intrinsic structure, monolithic architecture may have trouble expanding and meeting growing demands, even though it is easy to design and maintain.

The benefits of monolithic architecture in software development are numerous and help to explain its popularity. Its simplicity is one of its main advantages, as the uniform codebase streamlines the deployment and development procedures. A more efficient development cycle is made possible by having all the components in one location, which makes testing and debugging easier to handle. Since the codebase in the monolithic architecture is unified, it has the tendency to speed up the process for initial development, used largely for smaller projects. Monolithic architecture is highly efficient and simple while managing smaller software projects, hence a primary choice for these projects as these benefits are highly appealing, beneficial, and important to those systems (Newman, 2023b).

While monolithic architecture offers quite a lot of positivity, there also exists some disadvantages for using this architecture in a larger project. Scalability poses a significant challenge in this architectural pattern as the entire application must be scaled together, which generally leads to increased resource consumption. Also, it is generally inefficient and burdening in the larger projects with huge codebases. Also, if the applications grow and become increasingly complex, maintainability becomes increasingly tiresome and burdening as large monoliths often prove difficult to manage and update over time due to a large codebase. Monolith architecture is also coherent with limited flexibility regarding the tech stacks and available scalability options. As a result, teams often find themselves constrained with the choices made in the early SDLC process, particularly regarding the choice of stack and architecture patterns. These constraints potentially work to hinder adaptations to evolving needs and advancements in technology. These disadvantages underscore the importance of careful consideration when opting for a monolithic architecture, particularly considering long-term scalability and maintenance concerns. These disadvantages underscore the importance of careful consideration when opting for a monolithic architecture, particularly in light of long-term scalability and maintenance concerns (Skotnický, 2023).

2.2 Layered Architecture

The design pattern known as "layered architecture," or "n-tier architecture," divides the software system into several layers, each of which is in charge of a certain set of functionalities. Each of these layers only interacts with the layers directly above and below it, forming a hierarchical structure. This division of responsibilities aids in the application modularity, maintainability, and scalability (Bass et al., 2022). The presentation layer, sometimes referred to as the user interface layer in a layered architecture, is in charge of communicating with people and displaying information to them. User interfaces like those found on websites, mobile apps, and desktop programs usually make up this layer. Its main goal is to

give consumers an easy-to-use and aesthetically pleasing interface through which it may engage with the system (RIchards, 2015).

The primary business logic and application rules are contained in the business logic layer, sometimes referred to as the application layer. This layer manages the information flow between various system components, processes and manipulates data, and puts business rules into practice as shown in FIGURE 2. It encourages reusability and maintainability by encapsulating the business logic in a manner that is not dependent on any particular user interface or data storage technology (Fowler, 2015).

There are various advantages to using layered architecture in software development. It encourages the division of responsibilities, which facilitates system comprehension, upkeep, and modification. Additionally, because each layer can be written and tested separately, it makes component reusability easier. Because individual layers of a layered architecture can be scaled independently to accommodate changes in workload or user base, layered architecture also supports scalability. Layered architecture does, however, have several disadvantages. Strict component layering, particularly in big systems, can result in higher overhead and complexity. Because each layer adds a layer of indirection that might affect system performance, it may also lead to performance problems. Furthermore, adjustments to one layer could necessitate adjustments to adjacent layers as well, creating a close interaction and possible dependence between layers. (Software design Patterns, 2022).

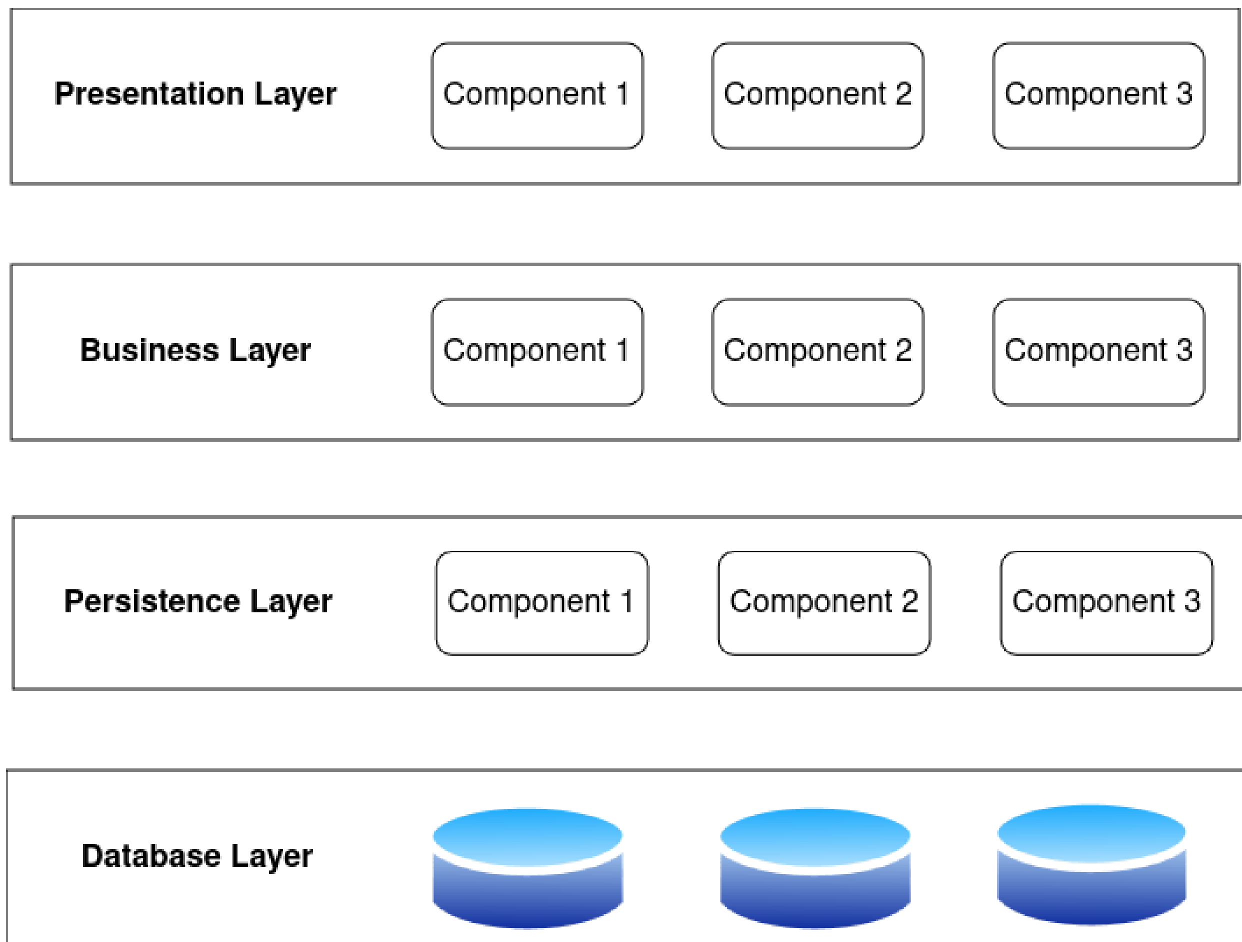


Figure 2 Layered Architecture (Richards, 2015)

2.3 Client-server architecture

In a client-server architecture, service providers—also referred to as servers—and service requesters—also referred to as clients—share duties or workloads. As shown in FIGURE 3, with this design, the two entities are clearly distinguished from one another. Clients make requests for resources or services, and servers respond to those requests. Requests are sent by clients to servers, which reply to them, and communication between clients and servers usually takes place over a network (Steen & Stuart, 2017).

Clients in a client-server architecture are often end-user gadgets like PCs, tablets, and smartphones. To use the resources or services the server offers, it communicate with it. Clients are in charge of establishing contact with servers, submitting requests for information or actions, and handling server responses. Conversely, servers are specialized hardware or software programs that offer resources or services to users. Servers are in charge of receiving requests from clients, processing them, and responding with the

proper information. Typically, it carry out operations like data processing, computation, or storage and manage resources like databases, files, or computing resources (Berson, 1996).

The varieties of client-server architecture can be further classified according to the type of communication and the functions of servers and clients. For instance, in a two-tier architecture, the display and business logic levels are handled by a single server with direct client-server communication. Clients interface with an application server in a three-tier design, which then acts as a mediator between clients and different servers for data storage, presentation, and business logic (Kleppmann, 2023).

The scalability and flexibility of client-server architecture are two of its main advantages. It is simpler to grow the system by adding more servers or clients as needed when tasks are divided across clients and servers. Furthermore, because servers may respond to requests from several clients at once and maximize resource usage, client-server architecture enables improved resource management. Client-server design does, however, come with several drawbacks. For clients and servers to communicate, a dependable network infrastructure is needed, and any interruptions in network connectivity may affect the system availability and functionality. Due to the possibility of unwanted access or data breaches, client-server communication over a network raises additional security concerns (Berson, 1996).

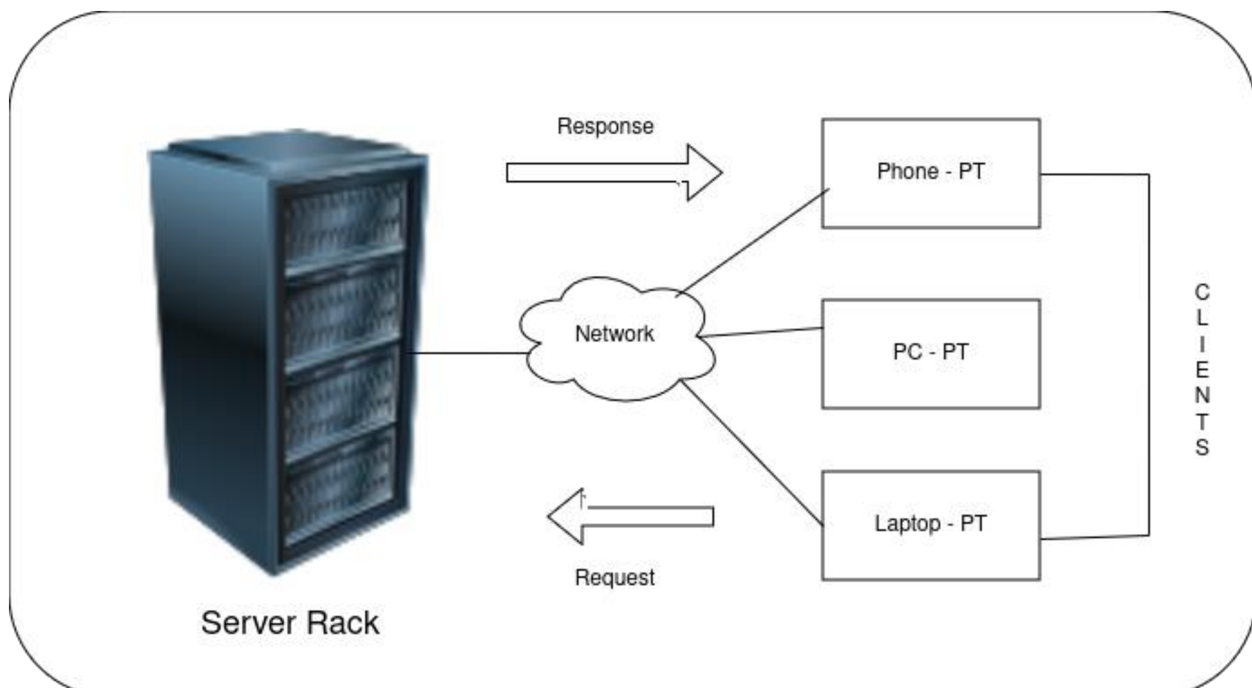


Figure 3 Client-server architecture (Richards, 2015)

2.4 Microservices Architecture

In a microservices architecture, an application is built as a collection of small, independent services, each representing a specific business capability. These services communicate with each other through well-defined APIs. Microservices promote loose coupling and allow each service to be developed, deployed, and scaled independently. Microservices also facilitate better fault isolation, as issues in one service are less likely to affect the entire system. Additionally, the use of microservices enables teams to adopt different technologies and frameworks suited to the specific requirements of each service, fostering innovation and agility within the development process. (Fowler 2015).

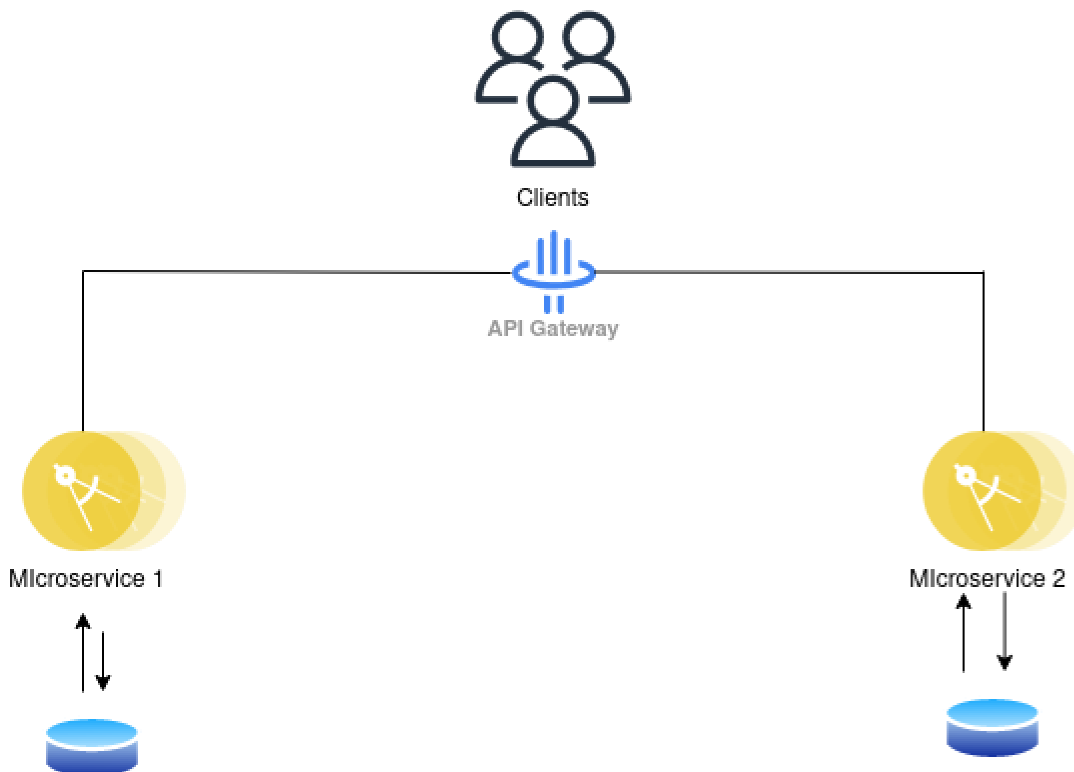


Figure 4 Microservices Architecture (Newman, 2023)

Microservice architecture consists of a lot of benefits, causing it to be one of the most popular software engineering architectural patterns currently used. It allows each component to act independently with other services, which operates a service to act as a standalone component that is developed and deployed separately with other components, offering it as one of its primary advantages. This strategy makes

upgrade and maintenance easier while also enabling quick service creation and deployment. Furthermore, its ability to manage data in a decentralized manner has enabled the microservices architecture to have a dedicated database or data storage platforms. Decentralization of databases and services reduces the possibility of system failures by increasing the flexibility and resilience of the system as a whole. Microservice design also offers high scalability as it offers each individual service to scale independently based on demand as shown in FIGURE 4. Considering all these benefits, it is essential to recognize the complexity that comes with the microservices design (Newman, 2023).

Microservices architecture offers several advantages which address the limitations of a monolithic system. One such advantage is service independent, where each service operates as an autonomous unit which is developed and deployed separately with communication to other relevant services. This approach enables each team to focus on individual services, which facilitates faster development cycles and codebase maintenance as per the need in the future. Since the microservice architecture embraces decentralized data management, it grants each service to have its own data storage or a database. This system of distributed data model enhances flexibility and resilience within the system. Also, microservices architecture promotes scalability by allowing each service to be scaled independently based on demand. This system of granular scalability enhances resource utilization and enables organizations to effectively handle varying workloads. (Davies & Kim, 2019). These advantages altogether enhance resource utilization, leading to collectively contribute to the appeal of the microservices architecture, offering greater agility, flexibility, and scalability, compared to traditional monolithic architectural design.

Although microservices architecture has many benefits, there are also important drawbacks that need to be considered. A prominent disadvantage is the inherent difficulty of distributed system management. It can be difficult and resource-intensive to coordinate communication and ensure that different services interact seamlessly, necessitating careful planning and execution. Furthermore, compared to typical monolithic systems, the initial development of a microservices design involves higher upfront expenses. Additionally, sustaining a microservices-based program presents operational problems. To guarantee that each service and the system as a whole run well, proper orchestration and monitoring are crucial. Microservices deployment becomes more complex as a result of the need for strong tooling and experience to manage these operational tasks effectively. It takes a lot of effort and money to set up the infrastructure, define service boundaries, and put communication protocols in place (Newman, 2023).

2.5 Event-Driven Architecture

The event-driven architecture (EDA) bases communication and information flow between program components on events. As shown in Fig 5, instead of direct method calls or synchronous requests, components in this architecture communicate asynchronously through the generation, detection, and response to events. Events are occurrences or notifications that signify a modification in the system state or a noteworthy action that has been carried out in an event-driven architecture. The system itself, user interactions, system operations, and external services are some of the sources from which these events can originate. These events are published by components—also referred to as event producers—to a central event bus or message broker. Other elements, referred to as event consumers or subscribers, indicate interest in particular event categories and sign up to receive alerts when such events happen (Bellemare, 2020).

Event consumers have the option to respond to events by carrying out pre-arranged. However, there are drawbacks to event-driven architecture as well. These include managing the complexity of event-driven processes, handling event ordering and consistency, and guaranteeing message delivery (Jain, 2021). While designing Event-driven systems, that need to be carefully planned, implemented, and monitored in order to handle the problems that might arise in the future, and be dependable and effective for tasks altering their state, or initiating other processes. This idea allows the system to be more flexible, scalable, and modular, thanks to the repeating loosely coupled and decoupled communication approach between the events that occur during the execution of the application. Event-driven architecture is highly effective in situations where the systems need to be highly scalable, flexible, and responsive to adhere to the changing circumstances. Since the event driven architecture allows the components to the events as soon as it occur, it makes it possible for the systems to manage distributed as well as real-time data processing, which in turn helps to handle complex problems more successfully (Bellemare, 2020).

The ability of the event-driven architecture to manage microservices as well as distributed systems is one of the major advantages. The notion of event propagation allows the system components that are divided among several nodes or services to communicate with one another. Since the components can be added or removed dynamically via this architecture without affecting the system, this architecture highly encourages scalability as well as fault tolerance. Also, since its components are loosely connected,

along with being easily replaceable without affecting other components of the system, it is concluded that the event-driven architecture design encourages modularity and reusability of the components. Since the majority of the components can function independently without affecting other components, even if the other components malfunction or fail, it renders the system more resilient to faults (Jain, 2021).

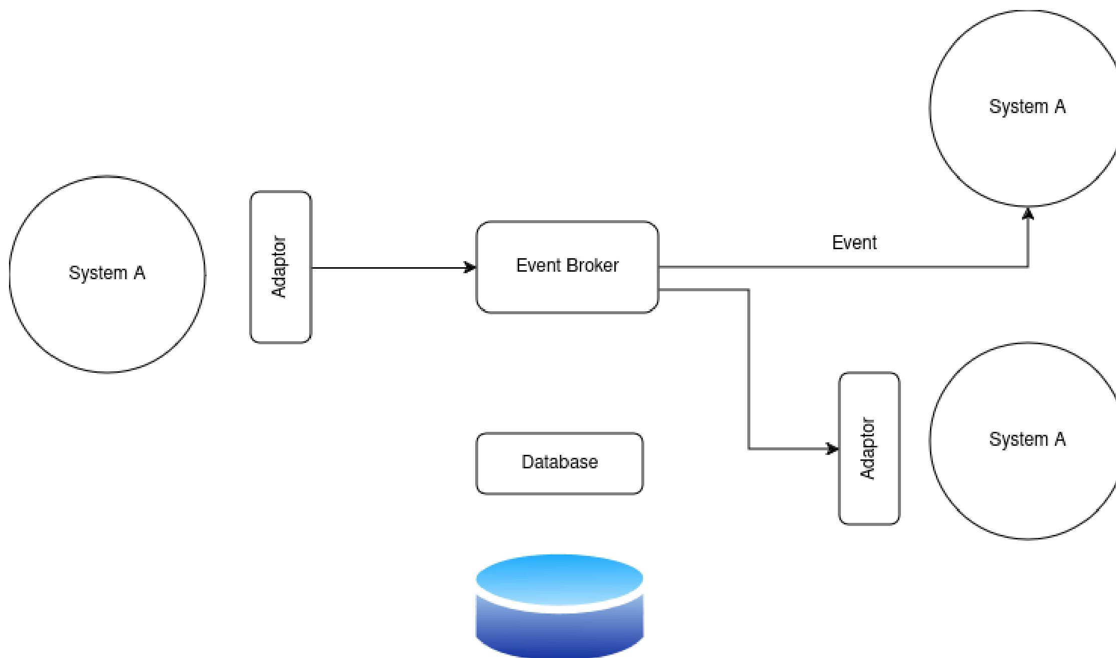


Figure 5 Event Driven Architecture (Bellemare, 2020)

2.6 Modular Architecture

Modular architecture is a design approach that structures a software application as a collection of independent, interchangeable modules. Each module has a well-defined set of functionalities and responsibilities and communicate with each other through well-defined interfaces. This approach offers several advantages over monolithic architecture, where the entire application is built as a single, indivisible unit. In the context of software development, modular architecture refers to a design process that divides a system into distinct, connected, yet independent components known as modules. Each module has a distinct set of responsibilities and features, and all communicate with one another through interfaces that are well defined with their responsibilities and communication approach to other modules. This approach offers several benefits over monolithic design, when the entire application is built as a single, indivisible unit. Modular architecture enhances maintainability, adaptability, and scalability by breaking down a complex system with a large codebase into smaller, easier-to-manage components. In this architectural

pattern, the core components of SDLC such as testing, development, and maintenance of the system are supported by the unique functions that each module serves (Hagel, 2020).

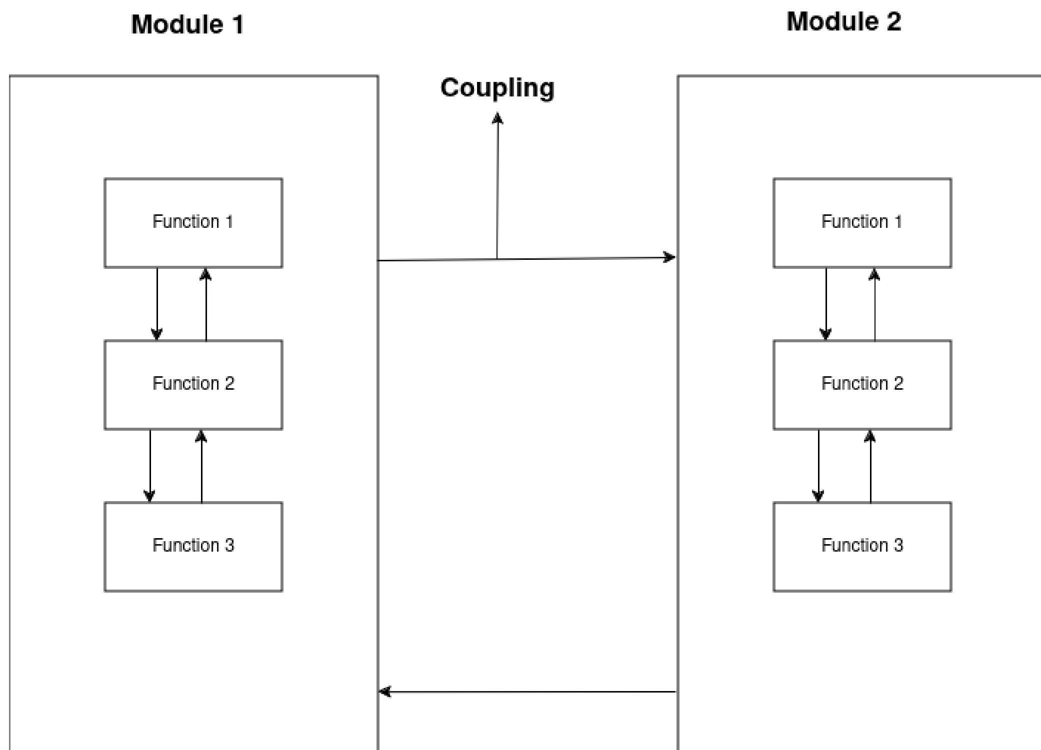


Figure 6 Modular Architecture (Cervantes & Kazman, 2016)

Fundamentally, decomposition is the act of dividing the system into more manageable, smaller modules according to certain functionalities, as illustrated in FIGURE 6. Teams may concentrate on specific components thanks to this modular structure, which expedites the development and maintenance process. Furthermore, a key component of modular architecture is interchangeability of components, which allows the upgrade or replacement of modules without causing any system disruptions. This feature improves flexibility of the system and makes it easier for the product to evolve smoothly with the passage of time. Furthermore, modules can be reused in different areas of the program or even in different applications, which makes reusability a crucial factor, allowing seamless and easier software development. Along with these benefits, encapsulation of each component allows each module to encapsulate its functionality, which minimizes dependencies on other modules or components. This idea of encapsulation of the modules allows improved code maintainability, which in turn enhances the system stability altogether (Rengaiah, 2014).

Modular architecture is widely used by software developers due to its many advantages. Teams can improve maintainability and optimize development efforts by breaking the system down into smaller, more manageable modules according to the needs. Since modules are changeable and may be upgraded without affecting the system, flexibility is encouraged, and development can continue even while certain modules are still being developed. Additionally, the ability to reuse modules across different application components or even different projects promotes uniformity, reduces redundancy, and speeds up development processes. Since dependencies are minimized, code maintainability gets better, and it is ensured that every module encapsulates its own functionality (Rengaiah, 2014).

The many benefits that modular design provides highlight its appeal in software development. Teams can optimize development efforts and improve maintainability by breaking down the system into smaller, more manageable modules that are based on specific features. As modules can be swapped out or upgraded without causing any systemic disruption, flexibility is encouraged, and ongoing development is made possible. Additionally, the flexibility to reuse modules in other application sections or even in multiple projects encourages consistency, cuts down on redundancy, and speeds up development cycles. By ensuring that every module encapsulates its own functionality, encapsulation reduces dependencies and enhances system stability and code maintainability. All of these benefits enable modular architecture to provide software solutions that are scalable, flexible, and effective, which is why it is a preferred method in a variety of fields and businesses (Gamma et al., 2021).

Although modular architecture has many strong benefits, there are certain drawbacks that should be considered as well. Potential for greater complexity is one significant disadvantage, especially in bigger systems with many interconnected parts. It might be difficult to manage dependencies and make sure that modules are communicating with each other, which increases development and maintenance costs. Furthermore, because modules may differ in design and operation over time, the flexibility provided by interchangeability and reusability may unintentionally result in lower system cohesion. Moreover, because inter-module connection and data exchange are required, the modular method may result in performance overhead. Furthermore, there might be a lot of work involved in making sure that modules are consistent and compatible with one another, particularly in diverse contexts (Dooley, 2011).

In summary, while monolithic architecture emphasizes simplicity and initial development ease, microservices and modular architectures focus on scalability, maintainability, and flexibility. The choice between these architectures often depends on the specific requirements and goals of the application being developed.

3 COMPARISON BETWEEN MONOLITHIC, MICROSERVICES, AND MODULAR ARCHITECTURE

Table 1 Difference between Monolithic, Microservices, and Modular Architecture

Topic	**Monolithic**	**Microservices**	**Modular**
Structure	Single codebase with all components tightly coupled.	Collection of independent services communicating through APIs.	Application divided into well-defined, interchangeable modules with interfaces.
Development	Simpler initially but becomes complex as the application grows.	Decentralized, promoting independent development and scaling.	Promotes code reuse and modularity, easing maintenance and updates.
Deployment	Entire application deployed as a unit, leading to downtime for updates.	Individual services deployed independently, minimizing downtime.	Individual modules can be deployed independently, offering flexibility.
Scalability	Difficult and inefficient to scale individual components.	Highly scalable due to independent scaling of services.	Moderately scalable with individual module scaling options.
Flexibility	Limited technology stack choices and adaptation options.	Maximum flexibility in technology choices and adaptation.	Offers more flexibility than monolithic but less than microservices.
Complexity	Simpler initially, but complexity increases with size.	Most complex due to distributed nature and service management.	Moderately complex, balancing granularity and maintainability.
Suitability	Small, simple applications, quick development, and ease of maintenance	Large, distributed systems, high scalability, and flexibility needs.	Larger, complex applications where maintainability and moderate scalability are important.

3.1 Structure

A monolithic architecture is defined by a single, coherent codebase that houses the whole application. All of the application components, including the client interface, data access layers, and business logic, are included in this architecture. This structure promotes application development and initial deployment because everything is in one place. However, as the application grows larger, managing and scaling monolithic systems can become more difficult. Any modifications or upgrades to a single program component may necessitate redeploying the entire monolith, which would extend release cycles and increase the likelihood of errors. Furthermore, monolithic designs might impede the adoption of modern ideas or languages because the entire codebase must be updated to accommodate them (Brooks, 1995).

Modular architecture, often known as monolithic modular architecture, aims to address some of the drawbacks of the monolithic approach by breaking the program up into smaller, independent modules. Each module, which can be developed, deployed, and scaled independently, handles a certain set of functionalities. The ability to modify a single module without impacting the entire program allows for more flexibility and agility in the development and deployment processes. The modules in a modular design may still be dependent on one another or have conflicts because it share the same resources and execution environment. Coordinating the integration and communication between modules may also be challenging, particularly as the number of modules rises (Addison-Wesley, 2000).

Microservices design builds on the concept of modular quality by breaking the software up into several loosely connected services, each running in a different, isolated environment. Every microservice has a distinct set of duties and communicates with other services through well-defined APIs. When this method is contrasted with monolithic and modular systems, it has several benefits. Microservices enable better scalability and fault separation since each one may be scaled and deployed independently. It also offer more technological flexibility because each service can be created with the best possible technology stack. However, data integrity, communication, and service discovery become more complex with microservices; also, managing a vast number of services can be challenging (Fowler, 2015).

3.2 Development

Software engineering development includes a range of approaches and techniques designed to handle the challenges involved in creating and managing large-scale applications. Different development methodologies arise in the setting of monolithic, modular, and microservices architectures, each with its own set of benefits and problems. Teams that use a monolithic development approach work together on a single, unified codebase that holds all of the application components. This method makes it simpler for developers to collaborate and communicate with one another, which speeds up communication and streamlines the setup process. However, as coordination gets more and harder as the codebase gets bigger and more complicated (Brooks, 1995). Changes made to one area could unintentionally impact others, creating conflicts or dependencies and possibly lengthening release cycles.

A monolithic modular architecture that uses modular development divides the program into smaller, self-contained modules. In "Design Patterns: Elements of Reusable Object-Oriented Software," Erich Gamma and associates address modularity principles, which are consistent with this methodology (Addison-Wesley, 2000). Modular development facilitates faster iterations and parallel development by allowing teams to work independently on particular modules or functionalities. It is possible for each module to have its own development lifetime, which allows for updates or revisions without affecting the application as a whole. Despite these advantages, managing module integration and communication can be difficult, requiring clear interfaces and protocols (Ingeno, 2018).

Martin Fowler and James Lewis describe in great depth the microservices development method, which is based on discrete services that stand in for discrete business functionality. Every service operates independently, each with its own codebase, development lifecycle, and deployment pipeline. As each team may concentrate on developing its own components, the decentralized method promotes creativity and innovation by releasing teams from architectural constraints so it can concentrate on delivering services. Things get more complicated, though, as teams have to manage dependencies and guarantee data consistency, connectivity, and service discovery (Fowler, 2015b). Careful coordination and efficient communication protocols are necessary to address these issues and guarantee system scalability and dependability. This facilitates and accelerates software development.

3.3 Deployment

Upgrades and updates are often introduced as one incorporated bundle the use of a single system. Since the code for the whole application is contained in a single codebase, changing the monolith additionally requires updating the whole software. This can result in lengthy deployment times and excessive danger due to the fact any errors or issues with the deployment procedure can affect the complete application. Approaches to monolithic systems have a tendency to be blue-inexperienced or canary releases to reduce downtime and prevent dangers because it assists, perceive and connect bugs faster(Brooks 1995).

On the other hand, deployment techniques in modular design awareness on propagating updates or changes inside every person module. The method to modular architecture layout lets every module to have its personal deployment techniques, permitting teams the power to optimize precise capabilities or functionality without affecting the complete system. This modular approach speeds up iterations and allows for more exact control over the deployment process. User feedback may be gathered, and updates can be released gradually with the use of strategies like A/B testing and feature toggles. However, there can be difficulties in overseeing the implementation of several modules and guaranteeing their smooth integration, especially when the quantity of modules grows (Newman, 2015).

In microservices architecture, deployment procedures focus on starting each service separately from the others. Teams can deliver updates to individual microservices without affecting the system as a whole because each microservice operates independently and has its own versioning system and deployment procedure. Deployment strategies including rolling updates, canary releases, and blue-green deployment are frequently used to enable upgrades that go smoothly and cause the least amount of disturbance. Furthermore, to handle scaling and automate deployment, microservices architectures frequently make use of containerization and orchestration tools like Docker and Kubernetes (Lukša, 2023). Nevertheless, deployment strategies need to take into consideration the additional complexity that comes with micro-services, such as the difficulties with communication, data consistency, and service discovery (Leszko, 2017).

3.4 Scalability

Scalability can be a challenge in monolithic design because application components are closely interrelated. Replicating the client interface, business logic, and data access layers is frequently necessary to scale the complete monolith. This can be wasteful and resource-intensive, particularly if just some application components need more resources. Additionally, because expanding resources for the entire application might not be feasible or cost-effective, monolithic architectures may find it difficult to manage sudden spikes in traffic or increasing demand. The load can be distributed more equally by using horizontal scaling, which is accomplished by running many instances of the monolith behind a load balancer (Abbott & Fisher, 2017). However, this tactic might be limited by the monolith architecture and infrastructure.

On the other hand, a modular architecture improves scalability by breaking the software up into smaller, independent components. More efficient use of resources is possible since each module can be scaled independently according to its own resource requirements. Teams may maximize resource consumption by scaling only the modules that demand more resources thanks to granular scalability. Additionally, serverless computing and microservices are two techniques that modular systems can use to further improve scalability. Teams may scale individual components independently when modules and services are decoupled, allowing for more flexible and responsive scaling strategies (Fowler, 2015). However, maintaining the scalability of several modules and guaranteeing their smooth interaction could be difficult, especially as the number of modules rises.

By design, microservices architecture provides the most scalability. With independent resource management, scalability, and deployment features, every microservice functions as an independent entity. Teams may expand individual services independently in response to demand thanks to this decentralized architecture, which makes for extremely effective resource use. At the microservice level, it is simple to implement auto-scaling, vertical scaling, and horizontal scaling, which allows the system to dynamically adjust to variations in workload or traffic. Furthermore, the administration of scalable microservices systems is made easier by containerization and orchestration technologies like Docker and Kubernetes. To guarantee scalability without compromising performance or reliability, microservices add more complexity to service discovery, communication, and data consistency. This complexity must be effectively managed (Leszko, 2017).

3.5 Flexibility

In monolithic design, the tightly coupled nature of application components may constrain flexibility. Making updates or changes to a single component of the program often entails altering the entire codebase, which can be laborious and error prone. Moreover, early technology decisions made during development may impose limitations on monolithic architectures, requiring substantial rewrites to switch frameworks or technologies. (Evans, nd) This rigidity can hinder innovation and adaptation to changing market trends or business needs.

A modular design addresses these limitations by dividing the application into manageable, standalone components, thereby enhancing flexibility. Agile development practices are facilitated as modules can be created, implemented, and updated independently. Teams can experiment with modern technologies or frameworks for specific modules without impacting the entire program, enabling quicker iterations and adaptation (Martin, 2017). Additionally, modular designs promote maintainability and code reuse through the sharing of modules between teams or projects. However, ensuring smooth interoperability across modules may require concise planning and design (Wolff, 2017).

Microservices architecture offers the highest degree of flexibility by design. Each microservice operates as an independent entity with its own codebase, technology stack, and development lifecycle. This decentralized approach empowers teams to select the best technologies for each microservice, fostering creativity and adaptability. Autonomous development, deployment, and scaling capabilities enable quick iterations and responsiveness to change needs. Also, microservices help facilitate experimentation as well as A/B testing as it helps to deploy the changes to individual services without impacting the entire system. However, the management of a large number of microservices as well as ensuring consistency and unification across the system requires a powerful governance and monitoring procedures to prevent complexity (Gamma et al., 2021).

3.6 Complexity

In monolithic layout, as the scale and complexity of the application develop, so does the complexity. Understanding and coping with the interconnections among various modules will become difficult when the entirety is tightly integrated inside a single codebase. Updates or changes in a single part of the utility

can impact different areas, affecting improvement efforts. Debugging and troubleshooting problems in a monolithic codebase require large navigation through more than one layer of code, making it onerous and tougher to start work on. As a result, adapting to modern technology or evolving business enterprise necessities without developing complexity turns into complicated (Evans, nd).

On the other hand, by segmenting the utility into extra plausible, standalone modules, modular layout seeks to lessen some of the complexity seen in monolithic structures. Although this improves maintainability and versatility, it moreover presents greater communication and module integration difficulties. Careful planning and format are critical to assure interoperability and coordinate interactions among numerous factors. Keeping tune of module dependencies and versioning can similarly complicate the improvement approach and raise the hazard of conflicts or inconsistencies. To maintain system coherence, strict governance and oversight strategies are consequently required (Evans, nd).

Microservices architecture introduces a separate set of complexities due to its loosely coupled nature. Issues arise concerning data consistency, communication, and service discovery despite offering greater flexibility and scalability. Coordinating interactions among multiple microservices, each with its own codebase and deployment pipeline, becomes challenging. Establishing reliable communication lines and solid backup procedures becomes essential in the microservice architecture. Ensuring data consistency across remote transactions and managing eventual consistency add considerably more complexity to the system. Microservices offer benefits in terms of flexibility and scalability, but handling their innate complexity requires planning and consideration (Evans, nd).

3.7 Suitability

Often, small to medium-sized initiatives with easy necessities for scalability generally benefit from monolithic structure. Monolithic architectures simplicity and ease of development are positive for projects with accurately described scopes and solid tech stacks. This ease of use not only speeds up development but also makes debugging and maintenance simpler, which is helpful for teams with limited resources or deadlines (Awati & Wigmore, 2022). Furthermore, since monolithic designs want less overhead in terms of deployment and management, it will be more sensible for projects with tight deadlines or useful resource constraints.

The modular design works exceptionally well for projects that need to be flexible and easy to manage. Breaking down a project into smaller independent modules can be useful for projects whose needs change frequently or require frequent updates. Teams can quickly expand and change to ongoing commercial and enterprise needs thanks to the modular design. Modular architecture can also be used to accelerate development and increase performance in projects that require code reuse or go-team collaboration (Rengaiah, 2014). In essence, projects can improve overall system efficiency and accelerate development cycles by optimizing code reuse through the adoption of modular design principles.

Large operations with strong desires and high scalability requirements are an acceptable quality for microservice systems. The decoupled nature of the microservice architecture can be great for remote clusters or heterogeneous stack operations because it provides additional flexibility in scaling individual components independently and properly handling failures in projects with varying workloads or that require fault tolerance and resilience. Microservice architectures reduce the possibility of cascade failures by isolating components and services, guaranteeing that disruptions are confined, and system integrity is preserved even under trying circumstances. (middleware, n.d.)

4 CONCLUSION

The success and persistence of applications are important architecture-driven factors in the rapid pace of software development. The unitary architectural model has been shown to be a reliable model for many applications, as well as other architectural models. However, as organizations and technologies change, so does the utility of unified systems in modern development contexts. This topic examines the complexity, advantages, disadvantages, and applications of the unitary architecture in modern software development environments.

It is clear from the research that unified systems have many advantages. Small- and medium-sized projects with clearly defined requirements are particularly suited because of their familiarity and simplicity, allowing for rapid development and deployment and, as communication costs decrease, tighter integration in a unified system tends to give the best performance. Moreover, the centralized design of the monolithic architecture facilitates the maintenance and debugging process, which overall enhances the stability of the system. Despite these benefits, monolithic architecture has inherent drawbacks that can be difficult and quick to scale up. Effectively adapting a monolithic architecture can be challenging due to the increasing scalability of the system and changing user requirements. The monolithic design of these architectures makes it difficult for teams to collaborate and deploy quickly because each change or update requires a redeployment of the entire application Also, larger monolithic applications can be complex if necessary if the entire code base is executed twice, there are performance challenges.

In response to these challenges, alternative architecture techniques, such as serverless computing and microservices, have become increasingly popular for around a decade. Due to the decentralized and modularized approach to application design, this architecture offers increased strength, scalability, and flexibility. Microservice architecture addresses many of the shortcomings of the monolithic system by breaking down applications into smaller independent systems. Similarly, serverless computing eliminates infrastructure management, freeing developers to focus solely on application logic. While these options are appealing, it is important to understand that single-occupancy housing still has advantages in some circumstances. Monolithic architecture can provide a practical and cost-effective solution for applications with relatively complex requirements and low scalability requirements. Additionally, there are significant risks and challenges associated with migrating existing monolithic applications to serverless microservice architectures that must be carefully considered.

In summary, evaluating collaborative planning in contemporary development contexts is not simply a hide-and-seek decision. Instead, each architectural design requires a specific business understanding and a detailed understanding of the specifics. Unitary architectures are better for some types of projects because it is simpler, more efficient, and more familiar. But as modern software development demands, developers and architects must be flexible, willing to consider innovative approaches that can transform today dynamic applications Developers negotiate the complexity of software architecture and changes in incoming images are accepted and buildable.

REFERENCE

Abbott, M.L. and Fisher, M.T. 2017. Scalability rules: Principles for scaling web sites. Boston, MA: Addison-Wesley.

Awati, R. and Wigmore, I. 2022. What is monolithic architecture in software?, WhatIs. Available at: https://www.techtarget.com/whatis/definition/monolithic-architecture Accessed: 20 February 2024.

Bass, L., Clements, P. and Kazman, R. 2022. Software architecture in practice. Boston: Addison-Wesley.

Bellemare, A. 2020.Building event-driven microservices: Leveraging organizational data at scale. Beijing: O'Reilly Media.

Berson, A. 1996. Client/server architecture. New York etc: McGraw-Hill.

Brooks, F.P. 1995. The mythical man-month: Essays on Software Engineering. Estados Unidos: Addison-Wesley.

Cervantes, H. and Kazman, R. 2016. Designing software architectures: A practical approach. Boston: Addison-Wesley.

Davis, C. and Kim, G. 2019. Cloud native patterns: Designing change-tolerant software. Shelter Island, NY: Manning Publications.

Design patterns: Elements of reusable object-oriented software 2000. Reading: Addison-Wesley.

Dooley, J. 2011. Software development and professional practice. Berkeley, CA: Apress.

Evans, E. (no date) Domain-driven design: Tackling complexity in the heart of software. Addison-Wesley.

Fowler, M. 2015. Patterns of enterprise application architecture. Boston, MA: Addison-Wesley.

Gamma, E. et al. 2021. Design patterns: Elements of reusable object-oriented software. Beijing Shi: Ji xie gong ye chu ban she.

Hagel, P. 2020. Modular Software Architecture for FPGA based Payload Module Computers. München: Dr. Hut.

How Microservices Architecture Works. Available at: https://middleware.io/blog/microservices-architecture/ Accessed: 20 February 2024.

Ingeno, J. 2018. Software architect's handbook: Become a successful software architect by implementing effective architecture concepts. Birmingham, UK: Packt Publishing Ltd.

Jain, A. 2021. Anti-patterns: Event driven architecture, LinkedIn. Available at: https://www.linkedin.com/pulse/anti-patterns-event-driven-architecture-arpit-jain/ Referenced 20 February 2024.

Kleppmann, M. 2023. Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems. Findaway World.

Leszko, R. 2017. Continuous delivery with Docker and Jenkins: Delivering software at scale. Birmingham; Mumbai: Packt Publishing.

Lukša, M. 2023. Kubernetes in action. Shelter Island: Manning Publications.

Martin, R.C. 2017. Clean architecture: A craftsman's guide to software structure and Design. Pearson.

Newman, S. 2023a. Building microservices: Designing fine-grained systems. Findaway World.

Newman, S. 2023b. Monolith to microservices: Evolutionary patterns to transform your monolith.

Rengaiah, P. 2014. On modular architectures, Medium. Available at: https://medium.com/on-software-architecture/on-modular-architectures-53ec61f88ff4 Referenced 20 February 2024.

Richards, M. 2015. Software architecture patterns. O’Reilly Media, Inc.

Skotnický, A. 2023. Microservices vs. Monolithic Architectures, taikun.cloud. Available at: https://taikun.cloud/microservices-monolithic-architectures-whats-the-difference/ Referenced 20 February 2024.

Steen, M. van and Stuart T.A.S. Andrew. 2017. Distributed systems: Principles and paradigms. Maarten van Steen.