DOI 10.47176/sjis.8.1.11

Original Article

SRPH

SJIS, 2026; 8(1): 11-18

# Comprehensive Analysis and Comparison of Software Architecture Patterns and Their Applications in Software Development

Yazdan shadab

Faculty of IT & Computer Engineering
Urmia University of Technology, Urmia
Iran.

***Corresponding Author:**
yazdan8686@gmail.com

**Received:** 22 December, 2025
**Revised:** 15 February, 2026
**Accepted:** 21 March, 2026
**Published:** 25 March, 2026

**ABSTRACT**

The ultimate guide to software architecture patterns ensures and facilitates a successful software development process. Why? Because the decision to implement one architecture pattern over another in a development project can change everything in today's dynamic world of software development. This article provides a comprehensive guide for selecting the appropriate architecture pattern for projects with varying requirements. First, it offers a clear explanation of the functionality, advantages, and disadvantages of several architecture patterns, including MVC, microservices, event-driven architecture, layered architecture, and others, analyzing these patterns through a comprehensive comparison based on learned literature. Next, it examines the real-world application of these patterns by assessing significant computational projects to determine who uses such patterns, for what purposes, and how theoretical evaluation can be translated into practical implementation to identify the best pattern for a project .Finally, the article presents the outcomes of comparing various architecture patterns and provides the final insights that software developers and architects need for successful advancement.

***Keywords:*** software architecture, software architecture patterns, Microservice, Layered architecture

## Introduction

Software is not only being utilized more extensively but also plays a greater role in enhancing the productivity of organizations and individuals. As projects become more complex and users increasingly seek to incorporate diverse perspectives, the need in the software creation domain is to establish faster pathways to achieve goals, enabling quicker development while ensuring the final outcome maintains quality and reliability. In this context, the approach of software architecture patterns has emerged as one of the key tools in the design and implementation of software systems.

Software architecture is the first step in the development which tells how to solve the problem. It shows how to build the system hence it can tell us about the behaviour of the system prior to the actual implementation [1].

With the rapid progress of technology and its necessary complications, software has come to make one of the main pillars of the technological foundations in our modern societies. Software development has long since ceased to be a matter of creating simple programs, as the act itself has evolved into a complicated, many-sided process, requiring more and more comprehensive ways of thinking about both design and implementation. In this context, the choice of the correct pattern can make the difference between success and failure for a whole project.

A high level explanation of software design plays a crucial and fundamental role in understanding and managing large and complex software systems. In fact, the qualitative attributes of large software systems (such as maintainability, reliability, usability, performance, flexibility, ...) are defined and constrained by software architecture. Software architecture plays a significant role in achieving the system's qualitative attributes, and in this regard, the evaluation of architecture for its effectiveness in meeting the desired quality requirements is of utmost importance in the early stages [2].



[ DOI: 10.47176/sjis.8.1.11 ]

Publisher: Scientific Research Publishing House (SRPH), Shirvan, Iran
http://srpub.org
Email: sjis@srpub.org

Architecture patterns help define the basic characteristics and behavior of an application. For example, some architecture patterns naturally lend themselves toward highly scalable applications, whereas other architecture patterns naturally lend themselves toward applications that are highly agile. Knowing the characteris tics, strengths, and weaknesses of each architecture pattern is necessary in order to choose the one that meets your specific business needs and goals [3].

Patterns of software architecture offer molds and reversible solutions that allow developers to efficiently and effectively solve the design problems of software systems in a more manageable way. Not only can these patterns help to refine a software structure and its organization, but they can also speed up development and facilitate comparison and analysis of approaches.

Selecting the right architectural pattern can make a crucial difference in the outcome of a project. Software architecture patterns as a set of principles for structuring the structure and behavior of systems assist software developers and architects to design robust, scalable, maintainable, and other systems based on previous experience. Because modern software projects are so diverse and complex, a thorough understanding of the pros and cons of every architecture pattern and of how it works in practice is needed.

With the wide variety of architecture patterns available, such as MVC, microservices, layered architecture, serverless, and others, developers face the challenge of selecting the best approach for the specific needs of their project. There are strengths and weaknesses in every pattern and which may need to be carefully analysed and contrasted. This is especially crucial in projects that require high scalability, durability, and continuous update capabilities, where a precise understanding of these patterns is vital.

The purpose of this article is to present a systematic and comparative evaluation of different architecture pattern solutions, so as to give practical suggestions to software developers and architects. Focusing on each of these patterns and from real-world applications, we identify the driving and failing factors for their success and failure.

The main goal of this study is to evaluate the strengths and weaknesses of each of these patterns and assess their applications in various software development contexts. For this purpose, we first describe the important features of each pattern and then we compare them using the selection criteria of, for example, performance, adaptability, and complexity. The findings of this study may help developers and software designers to make more informed decisions for their projects and have a potentially beneficial impact on the development process and the quality of software products. Architecture of a software system should only explain how to achieve the requirements of the customer. It should not disclose any implementation detail[1].

Architecture should be prepared by keeping the view that software system should be flexible to incorporate future changes and maintenance requirements [4].

Architecture should be prepared in such a way that the future product based on it should contain all the functional and non-functional requirements of the customer [4].

Architecture of a software system should try to predict all possible risk that can be projected to the system in future [5].

Architecture of a software system should support optimum utilization system resources [1].

**Architectural patterns**

Software architecture patterns are standard and reusable solutions that help software engineers address common issues in the design of software systems more effectively. These patterns serve as roadmaps for improving the quality and efficiency of systems, and they can lead to reduced design complexity, as well as increased stability and maintainability of software. Software architecture patterns function as key tools in the software development process and have multiple applications that contribute to enhancing the quality and performance of systems. Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice [6].

Architectural patterns help developers design system structures in a way that reduces complexity. By using standard patterns, developers can leverage previous experiences. By separating concerns and decomposing system components, architectural patterns contribute to improved software maintainability. This enables developers to implement changes easily without affecting other parts of the system. Architectural patterns like microservices and SOA allow developers to scale systems efficiently. These patterns enable organizations to quickly respond to changing market demands. With architectural patterns, development team members can collaborate more easily. They help create a common language and better understanding of the system's structure. Architectural patterns enable developers to test system components independently, making troubleshooting and software quality assessment easier. These patterns also allow developers to integrate new technologies into existing systems, helping organizations stay at the forefront of innovation. In recent years, various architectural patterns have been documented by software architects, each developed based on specific requirements. Among the most comprehensive of these are the patterns listed below and shown in Fig 1.





[ DOI: 10.47176/sjis.8.1.11 ]

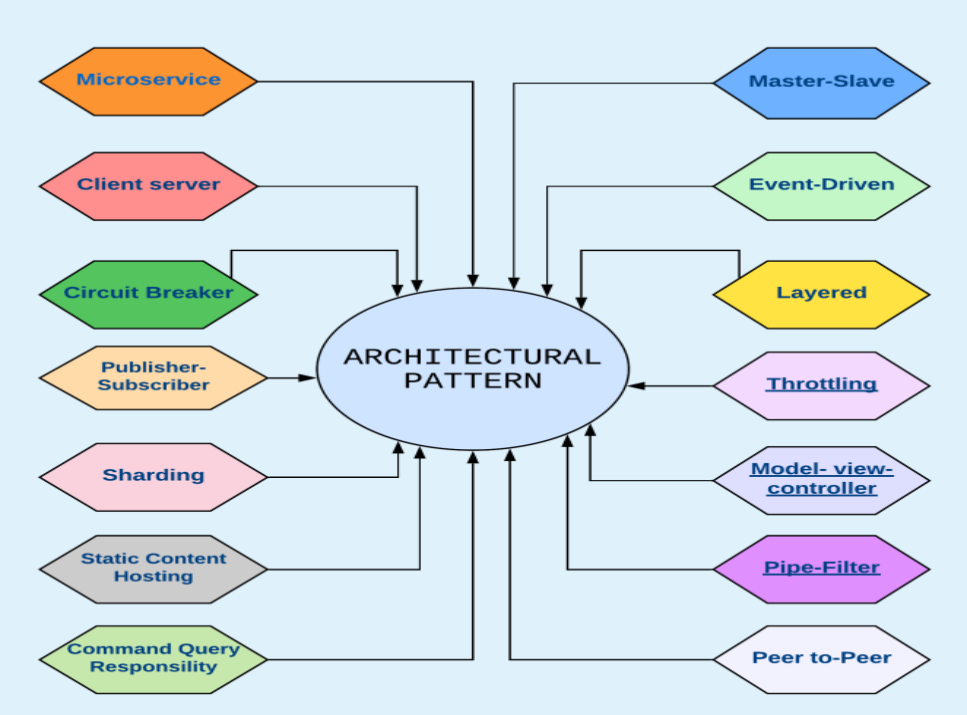


**Fig1.** A number of architectural patterns

As shown in (Fig 1), there are several types of architectural patterns, which do not encompass all the available patterns. In this article, we focus on five of them: Layered, Microservices, Event-Driven, MVC, Pipe & Filter, and provide a comparison between these six architectural patterns.

The use of architectural patterns not only reduces design complexity but also increases the stability and maintainability of systems [7].

Once you need the pattern, you have to figure out how to apply it to your circumstances. A key thing about patterns is that you can never just apply the solution blindly, which is why pattern tools have been such miserable failures [6].

## Layered architecture patterns

Layered architecture is one of the popular patterns in software design that helps in separating responsibilities and organizing system components. This pattern allows developers to divide complex systems into distinct layers, where each layer has its own specific tasks. This type of architecture is particularly useful in large and complex systems that require scalability and easy maintenance.

The most common architecture pattern is the layered architecture pattern, otherwise known as the n-tier architecture pattern. This pattern is the de facto standard for most Java EE applications and therefore is widely known by most architects, designers, and devel opers. The layered architecture pattern closely matches the tradi tional IT communication and organizational structures found in most companies, making it a natural choice for most business application development efforts [3].

Layered software architecture is one of the most common and widely used architectural patterns in the world of software development. Its simplicity and modular structure have made it highly popular among developers. This architecture is particularly well-suited for projects with high complexity and a need for continuous maintenance and expansion.

The layered architecture divides the system into a set of layers, each responsible for specific tasks, typically interacting only with adjacent layers. For example, it often includes layers such as Presentation, Business Logic, and Data Access. These layers are generally described as follows fig2.

*Presentation Layer*

This layer is responsible for displaying information to the user and receiving input from them. The presentation layer serves as the main interface between the system and the user, enabling interaction and providing access to the desired information.

The presentation layer includes managing all of the user interactions, whereas the responsibilities of the data layer include managing the persistence of data [8].





[ DOI: 10.47176/sjis.8.1.11 ]

Imagine an online store as an example. The presentation layer handles the display of products, shopping cart, user information, and other store-related details. It also receives user inputs, such as selecting products, entering account details, and making payments, and forwards this information to the underlying layer.

*Business Logic Layer*

This layer contains the system's business logic and rules. It is responsible for all data processing, calculations, decision-making, and other business-related operations. Acting as the core of the system, this layer performs the primary business functions.

*Persistence Layer*

The Persistence or Data Access layer focuses on managing the storage and retrieval of data from databases or other storage locations. This layer interacts with the database and handles data management. It enables the business logic layer to access data without concerning itself with the database's implementation details.
In general, this layer is responsible for reading and writing data, managing database connections, and abstracting the database's implementation details.
*Database Layer*

This layer consists of the database itself and the data storage structures. It directly works with the data and typically includes Database Management Systems (DBMS).

*Advantages and Disadvantages*

Layered architecture brings many advantages, including isolation from issues, testability, scalability, and simplification. But there are also limitations, for example overseparation may sometimes result in increased complexity.
The layered architecture pattern can slow down the overall communication between different components of the system. Additionally, determining the exact number of layers required for a system can be challenging [9]

As previously mentioned, and shown in Fig 2, the layered architecture typically consists of four distinct yet interconnected layers: Presentation, Business, Persistence (Data Access), and Database, each with its specific responsibilities.
However, this is not always the case. The number of layers and components or modules can vary depending on the specific requirements.
In some cases, the business layer and persistence layer are combined into a single business layer, par ticularly when the persistence logic (e.g., SQL or HSQL) is embed ded within the business layer components. Thus, smaller applications may have only three layers, whereas larger and more complex business applications may contain five or more layers [3

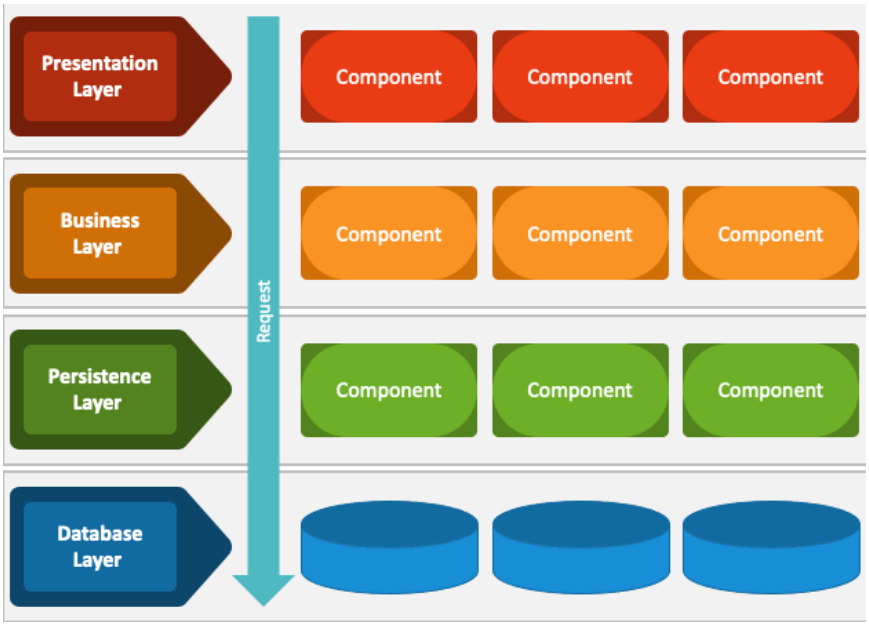


**Fig2**. Layers of layered architecture



[ DOI: 10.47176/sjis.8.1.11 ]

**Microservice**

Microservice Architecture (MSA) is an architecture for software that breaks down an entire system into a number of independent services. All services are developed, deployed, and scaled in isolation and are designed to talk to each other via network protocols including HTTP. This architecture enables greater scalability, faster deployments, and flexibility in updating and maintaining systems. Yet, the complexity and the required sophisticated infrastructure for services' coordination and monitoring are critical challenges of microservices. Large organizations, including Netflix, Amazon, and Uber, adopt microservices architecture for the performance and scalability of their systems.

The Microservices Architecture pattern enforces a level of modularity that in practice is extremely difficult to achieve with a monolithic code base. Consequently, individual services are much faster to develop, and much easier to understand and maintain. this architecture enables each service to be developed independently by a team that is focused on that service. The developers are free to choose whatever technologies make sense, provided that the service honors the API contract [10].

The broadest definition of microservices architecture is an approach to developing a single application as a set of small, independently running services. These services communicate with each other through lightweight mechanisms, often via HTTP or APIs. Unlike monolithic architectures, microservices enable independent scaling and can be developed using different technologies, fostering flexibility and growth in the development process. There is no consensus on the relationship between MSA (Microservices Architecture) and SOA (Service-Oriented Architecture). Some proponents of MSA argue that it is a new architectural style, while some supporters of SOA believe that MSA is simply an implementation approach of SOA [11].

Service-Oriented Architecture (SOA) and Microservices Architecture (MSA) are all service-oriented architectures. SOA is composed of a large number of interrelated services, which usually have a unified database, whereas MSA is composed of unrelated services, each with its own database, with the purpose of achieving scalability and agility. MSA provides advantages, such as scalability, flexibility in development and fault tolerance, but has challenges, including management complexity in the sense of networking, coordination, and communication among the network. Finally, the architecture selection actually relies on the particular requirement of the project.

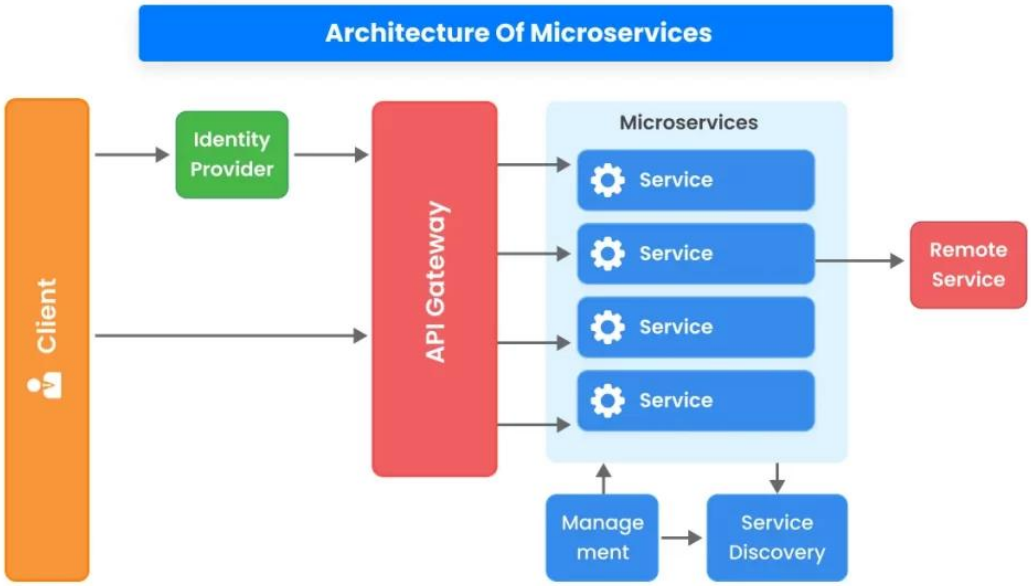


**Fig3**. Microservices architecture pattern

**Event Driven**

Event-Driven Architecture (EDA) is a software architecture design pattern that addresses the creation, identification, uptake, and action to events. There, in this architecture, the system is decomposed in event producers and consumers that are completely independent and only communicate via messaging systems such as Apache Kafka or RabbitMQ.





[ DOI: 10.47176/sjis.8.1.11 ]

This architecture allows streaming, scalable execution, in which components execute asynchronously and respond to either system states or events in the system.
EDA enables systems to respond quickly to changes, scale efficiently, and improve operations through automation. However, it also presents challenges like managing event complexity and maintaining data consistency. This architecture is particularly useful in sectors like finance, e-commerce, and IoT, helping organizations respond effectively to dynamic workloads and real-time requirements.
In this model, messages are referred to as events, which is why it is called event-driven architecture [12].
Producers and consumers do not need to be aware of each other's existence, allowing new components to be added to the system without impacting existing ones. Along with event sourcing, EDA enables repeatability and transparency [13]

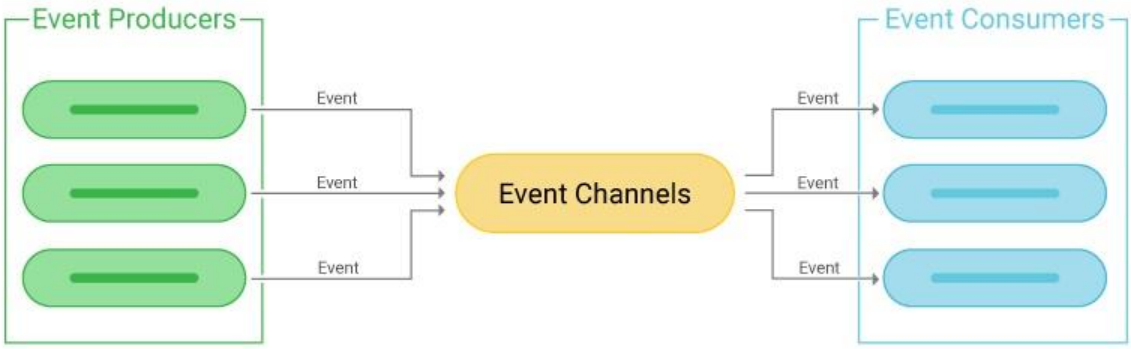


**Fig4**. Event-Driven Architecture design pattern

**Model View Controller**

The Model-View-Controller (MVC) is a software design pattern most widely used for developing a user interface. It separates the application logic into three unified parts – Model, View, and Controller. The Model is the data, the View is the output to the user, and the Controller is the user input to the application which updates the Model and View. This separation indeed makes maintaining and scaling software systems far easier fig 5.

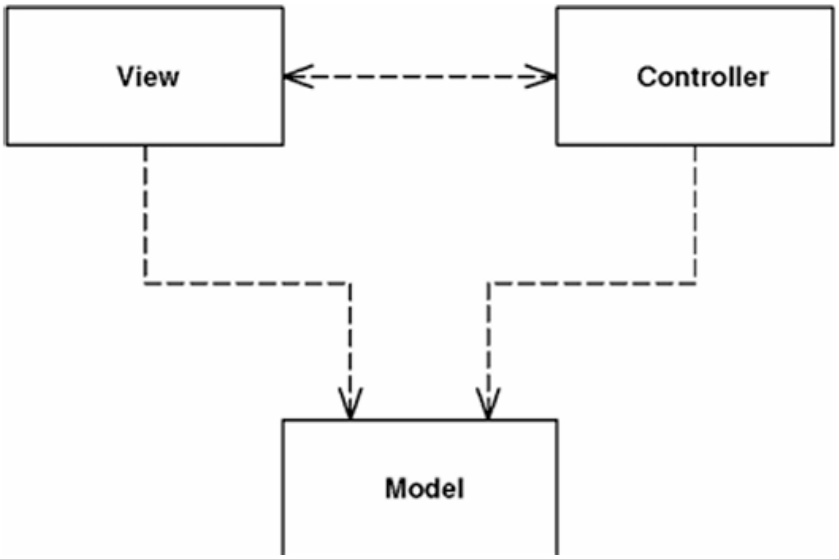


**Fig5**. MVC pattern

View: The View section is responsible for displaying data to the user. It presents the data visually without business logic. Its main role is to offer a user interface and display information.





[ DOI: 10.47176/sjis.8.1.11 ]

Controller: The Controller acts as an intermediary between the Model and the View. It processes user requests, interacts with the Model to retrieve data, and sends it to the View for display. It implements business logic and manages data flow between the Model and View. The model is an object that represents some information about the domain. It's a nonvisual object containing all the data and behavior other than that used for the UI. In its most pure OO form the model is an object within a Domain Model . You might also think of a Transaction Script as the model providing that it contains no UI machinery. Such a definition stretches the notion of model, but fits the role breakdown of MVC [6].

**Pipe And Filter**

The Pipe & Filter architecture pattern is a software design pattern that divides a system into a set of filters, each responsible for a specific task in data processing. These filters are connected to each other through pipes, which transfer the data between them fig6.

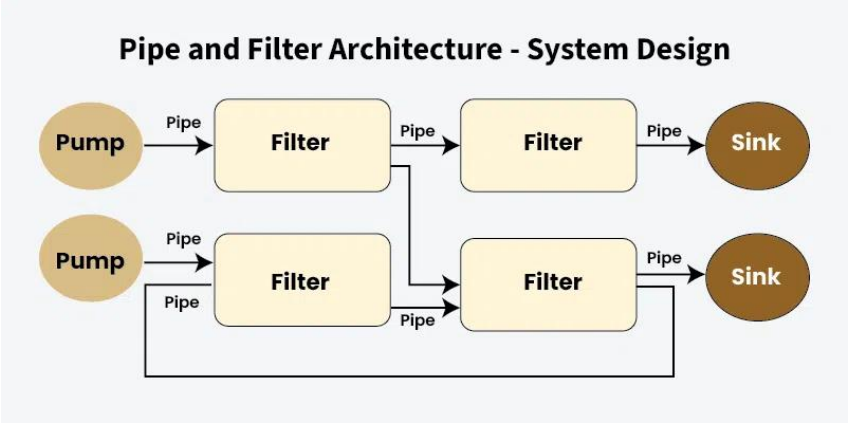


**Fig6**. Pipe & Filter architecture pattern

(The filters are executed independently and in parallel, and the pipes enable asynchronous data transfer. This pattern helps increase the maintainability, scalability, and flexibility of the system.)

**comparison**

In this paper, we have discussed five types of architectural patterns, each developed based on diverse requirements. Each of these architectural patterns provides a solution to specific and different needs. Therefore, a general comparison is not accurate because each one was created to address a particular problem or question. Thus, the comparison we will conduct in this paper will be in the form of a table, which is categorized based on quality requirements. It will indicate which pattern provides the best solution for each specific quality requirement Table 1.

**Table 1 .**Comparison of architectural patterns based on quality requirements

| Event Drive | Model View Controller (MVC) | Pipe & Filter | Layered | Microservices | Architecture Patterns | |
|---|---|---|---|---|---|---|
| **** | *** | *** | *** | *** | *** | Quality requirements |
| Medium | Medium | Low | High | Very High | *** | Security |
| High | Medium | High | Medium | Very High | *** | Scalability |
| High | Medium | Medium | Medium | High | *** | Performance |
| Medium | Medium | High | High | Medium | *** | Stability |
| Medium | High | Medium | High | High | *** | Maintainability |
| High | Medium | Medium | High | Very High | *** | Reliability |
| Low | High | Medium | High | Medium | *** | usability |



[ DOI: 10.47176/sjis.8.1.11 ]

**Conclusion**

In this article, we discussed architectural patterns in software architecture and highlighted their importance and effectiveness in reducing complexity, improving testability, and more. We elaborated on five software architecture patterns, which include the following:

- Microservices
- Layered
- Pipe & Filter
- Model View Controller (MVC)
- Event Driven

Finally, we compared these architectural patterns, and the results can be seen in Table 1.

**Acknowledgement**

I would like to express my sincere gratitude to Dr. Rashidi for providing valuable references and guidance throughout the course of this research.

**References**

1. Anubha, Sharmaa. and Manoj, Kumarb. And Sonali, Agarwalc. (2015) "A Complete Survey on Software Architectural Styles and Patterns" Procedia Computer Science,Volume 70.
2. Montaghi, V. A., & Mirian Hossein Abadi, S. H. (2007). A method for comparing software architectures. Proceedings of the Annual National Conference of the Iranian Computer Society.
3. Mark Richards, 2022," Software Architecture Patterns", 2nd Edition,O'Reilly Media, Inc.
4. Lindström, Åsa, et al. "A survey on CIO concerns-do enterprise architecture frameworks support them?" Information Systems Frontiers 8.2 (2006): 81-90
5. Bass, Len, Paul Clements, and Rick Kazman. Software architecture in practice. Addison-Wesley Professional, 2003.
6. Fowler, M. (2012). Patterns of Enterprise Application Architecture. United Kingdom: Pearson Education.
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Germany: Addison-Wesley.
8. Cervantes, H., Kazman, R. (2016). Designing Software Architectures: A Practical Approach. United Kingdom: Pearson Education.
9. Sharma, A., Kumar, M., & Agarwal, S. (2015). A complete survey on software architectural styles and patterns. Procedia Computer Science.
10. Richardson, C. (2017). Microservices: Designing and deploying. NGINX, Inc
11. Li, S., Zhang, H., Jia, Z., Zhong, C., Zhang, C., Shan, Z., Shen, J., & Babar, M. A. (2021). Understanding and addressing quality attributes of microservices architecture: A Systematic literature review. Information and Software Technology, 131, 106449.
12. Fernando, C. (2022). Designing Enterprise Platforms with Event-Driven Architecture Patterns. In Solution Architecture Patterns for Enterprise: A Guide to Building Enterprise Software Systems (pp. 147-179). Springer.
13. Krause, T., Zickfeld, M., Bruchhaus, S., Reis, T., Bornschlegl, M. X., Buono, P., Kramer, M., Mc Kevitt, P., & Hemmje, M. (2023). An Event-Driven Architecture for Genomics-Based Diagnostic Data Processing. Applied Biosciences, 2(2), 292-307.

SJIS



**Citation:** Shadab Y. Comprehensive Analysis and Comparison of Software Architecture Patterns and Their Applications in Software Development. SJIS, 2026; 8(1): 11-18.

https://doi.org/10.47176/sjis.8.1.11



[ DOI: 10.47176/sjis.8.1.11 ]