IJARSCT

ISSN (Online) 2581-9429

International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)

International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal

Impact Factor: 7.53

Volume 4, Issue 3, June 2024

Comparative Analysis of Monolithic vs. Distributed Architecture.

Amey Arun Padvekar and Vikaskumar Badriprasad Gupta
Students, Master of Computer Applications (MCA)
Late Bhausaheb Hiray S. S. Trust's Institute of Computer Application, Mumbai, India

Abstract: *This article compares Monolithic and Distributed structures, highlighting their strengths, weaknesses, and best uses. Monolithic architecture offers a simple, easy-to-develop, and efficient deployment solution for high-performance, low-complexity situations, but struggles with maintenance and scalability as applications grow. Distributed architecture, with its flexible, scalable, and fault-tolerant microservices and service-oriented designs, is ideal for large, complex systems, allowing for independent development, deployment, and scaling of services. However, it brings challenges in service orchestration, data consistency, and network latency. According to an analysis of case studies and performance metrics, the research suggests that monolithic architectures work well for small to medium-sized programs with limited scalability requirements, while distributed architectures are better suited for large, dynamic environments with a strong emphasis on fault tolerance and scalability.*

Keywords: Monolithic Architecture, Distributed Architecture, Microservices, Scalability, Fault Tolerance, Performance, System Complexity, Software Development, Deployment, Maintainability

I. INTRODUCTION

In software development, architectural considerations are crucial to an application's success and longevity. The two main architectural paradigms, monolithic and distributed, each have their own set of advantages and problems, and we provide a comparative analysis of both architectures to explain their strengths, limitations, and pitfalls, as well as their optimal application scenarios. A monolithic architecture is distinguished by a single, consistent code base in which all components and functionalities are integrated. This traditional approach has been the backbone of software development for decades since it is simple to conceive, create, and deploy. Monolithic systems are frequently selected due to their efficiency and simplicity of testing. However, when systems grow in complexity and size, monolithic architectures can become onerous, and difficult to maintain, scale, and isolate faults.Distributed architectures, such as microservices and service-oriented architecture, promote the separation of applications into smaller, independent services. Each service focuses on a specific business function and may be built, launched, and scaled automatically. This modularity boosts flexibility, encourages continuous delivery, and enhances fault tolerance. However, the distributed nature of this architecture raises issues such as service orchestration, inter-service communication, data consistency, and network latency. This article seeks to provide a thorough knowledge of the context in which each architecture succeeds by delving into performance measures, case studies, and real-world examples.

The analysis takes into account system complexity, scalability needs, development and deployment methods, maintainability, and fault tolerance. This comparison study seeks to serve as a thorough guide for software architects and developers, assisting them in making educated selections that are consistent with their strategic goals and operational requirements for their projects.

II. BACKGROUND

2.1 Monolithic Architecture

A. Definition and explanation

A monolithic architecture is a traditional software development approach where a single code base handles multiple business function.All the software components in a monolithic system are interdependent due to the data exchange

IJARSCT
ISSN (Online) 2581-9429
International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
Impact Factor: 7.53
Volume 4, Issue 3, June 2024

mechanisms within the system. This setup makes modifying a monolithic architecture quite restrictive and time-consuming because even small changes can impact large parts of the code base.

B. Historical Context

Monolithic architectures have been around since the early days of computing. While it's difficult to credit one individual with their creation, companies like IBM played a crucial role in shaping early software architecture, particularly with their development of mainframe computers in the 1960s and 1970s. Back then, Networking capabilities were limited, and development tools were just beginning to emerge. Applications were generally smaller and less complex, and they were managed by small, centralized teams. In this environment, monolithic architecture was a natural fit, as concepts like modular programming and distributed systems were still developing.

C. Traditional use cases

Monolithic architecture is traditionally used in scenarios where system complexity is relatively low, and performance is a critical factor. Common use cases include:

- Enterprise Applications: Many business applications, such as Enterprise Resource Planning (ERP) systems and Customer Relationship Management (CRM) systems, have historically been developed as monolithic applications due to their straightforward deployment and maintenance.
- Desktop Applications: Software that runs on individual desktops, like word processors and accounting software, often adopts a monolithic structure.
- Early Web Applications(e.g. Fig. 1): Initial web applications, such as content management systems and e-commerce platforms, were frequently built as monoliths to simplify development and deployment.

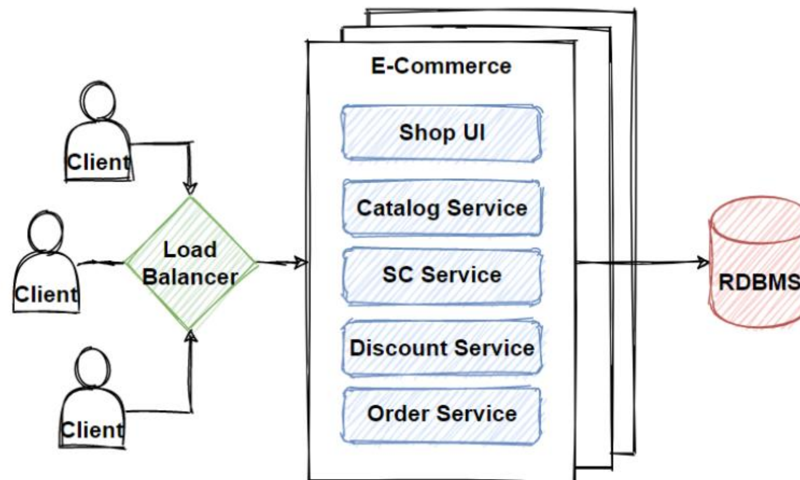

Fig. 1. Monolithic Architecture

2.2 Distributed Architecture

A. Definition and explanation

A microservices architecture, also simply known as microservices, is an architectural method that relies on a series of independently deployable services. These services have their own business logic and database with a specific goal. Updating, testing, deployment, and scaling occur within each service. Microservices decouple major business, domain-specific concerns into separate, independent code bases. Microservices don't reduce complexity, but they make any

IJARSCT
ISSN (Online) 2581-9429
International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
Impact Factor: 7.53
Volume 4, Issue 3, June 2024

complexity visible and more manageable by separating tasks into smaller processes that function independently of each other and contribute to the overall whole.

Adopting microservices often goes hand in hand with DevOps, since they are the basis for continuous delivery practices that allow teams to adapt quickly to user requirements.

B. Evolution

The shift towards distributed architectures began in the late 1990s and early 2000s, driven by the need for greater scalability and flexibility in software systems. The rise of the internet and the proliferation of web-based applications exposed the limitations of monolithic architectures, particularly in terms of scalability and agility.

SOA Emergence: Service-oriented architecture emerged as a response to these limitations, providing a more modular approach to software design. However, SOA's complexity and the overhead associated with ESBs led to mixed success.

Microservices Rise: In the early 2010s, microservices gained popularity as a more granular and flexible approach to distributed architecture. Pioneered by companies like Netflix and Amazon, microservices enabled organizations to rapidly scale and deploy new features, significantly improving their ability to respond to market changes.

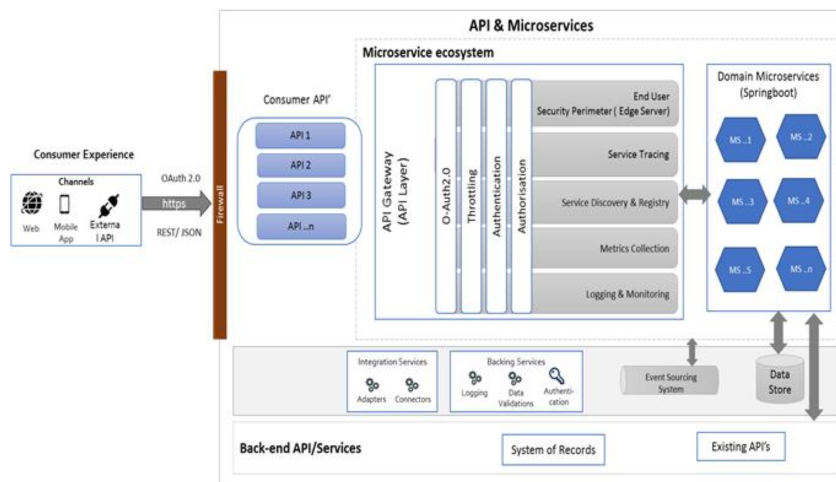

Fig. 2. Microservices Architecture

2.3 Traditional use cases

Distributed architectures are suitable for a wide range of scenarios, particularly those involving large-scale and complex systems. Common use cases include:

- Large-Scale Web Applications: Applications with high traffic volumes, such as social media platforms, streaming services, and e-commerce websites, benefit from the scalability and fault tolerance of distributed architectures.
- Cloud-Native Applications: Software designed to leverage cloud computing resources often adopts a distributed approach to maximize elasticity and resilience.

III. METHODOLOGY

The comparative analysis of Monolithic and Distributed architectures is conducted using a structured framework that assesses multiple dimensions of software architecture. This framework includes criteria such as performance, scalability, maintainability, fault tolerance, and development complexity.

Literature Review:

Grzegorz Blinowskiet al. [1], compares the Monolithic and Microservice Architecture.Microservices-based architecture has grown in popularity due to its benefits, including improved availability, fault tolerance, and horizontal

IJARSCT
ISSN (Online) 2581-9429
International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
Impact Factor: 7.53
Volume 4, Issue 3, June 2024

scalability, as well as increased software development agility. The key lesson is on a single machine, a monolithic performs better than its microservice-based counterpart.

Microservices and monolithic architectures are compared in terms of performance in Omar Alstudy Debagy's [3], to ascertain how these architectures, perform in various scenarios using various testing setups. This paper concludes that monolithic applications and microservices can perform similarly when the application is under normal load. A monolithic application may perform marginally better than a microservices application under a light load of less than 100 users.

The authors of the work by Konrad Goset al. [4] compared the Monolithic and Microservices architecture using the Gatling load testing tool. The tests were performed on PC with Ubuntu18.04.2 LTS operating system. The applications were deployed with Docker. This paper compares the monolithic and microservices architecture on different parameters like Architecture performance, and response time by sending several HTTP GET and POST requests. This paper also describes the pros and cons of Monolithic and Microservice sarchitecture.

Case Studies: Review documented case studies of real-world applications.

Performance Testing:

Monolithic E-commerce:

Configuration:

- Threads (Users): Start with 50 and scale up to 1000.
- Ramp-Up Period: 30 seconds.
- Loop Count: 10.

Steps:

- Prepare JMeter Test Plan:
- Create a new Test Plan in JMeter.
- Add a Thread Group with the configuration mentioned above.
- Add HTTP Request samplers for endpoints /products, /products/:id, and POST /products.
- Add Listeners like View Results Tree, Summary Report, and Aggregate Report.
- Run the Test Plan:
- Save and execute the test plan.
- Monitor response times, throughput, and error rates using JMeter listeners.Gradually increase the number of threads to test higher loads.

Microservices E-commerce :

Configuration:

- Threads (Users): Start with 50 for each service and scale up to 500.
- Ramp-Up Period: 30 seconds.
- Loop Count: 10.

Steps:

- Prepare JMeter Test Plan:
- Create a new Test Plan in JMeter.
- Add two Thread Groups, one for the Product Service and one for the Order Service, each with the configuration mentioned above.
- Add HTTP Request samplers for endpoints /products, /products/:id, POST /products for Product Service, and /orders, /orders/:id, POST /orders for Order Service.
- Add Listeners like View Results Tree, Summary Report, and Aggregate Report to each Thread Group.
- Run the Test Plan:

IJARSCT

ISSN (Online) 2581-9429

International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)

International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal

Impact Factor: 7.53

Volume 4, Issue 3, June 2024

- Save and execute the test plan.
- Monitor response times, throughput, and error rates using JMeter listeners.

IV. COMPARATIVE ANALYSIS

4.1 Performance and Scalability

Monolithic:

- Performance Characteristics: Monolithic designs provide efficient performance for small to medium-sized systems due to reduced overhead and streamlined communication between components.
- Scalability Limitations: As applications grow, monolithic systems encounter scaling issues. Scaling necessitates duplicating the entire system, which leads to inefficiencies. High resource consumption and difficulty handling high volumes of traffic might cause performance bottlenecks.

Distributed:

- Performance Characteristics: Distributed designs, like microservices, improve performance at scale. Each service is fully scalable to suit demand.
- Scalability: Distributed systems provide excellent scalability. They enable horizontal scaling by adding extra instances of specialized services rather than duplicating the entire program, hence optimizing resource usage and boosting overall performance under heavy traffic.

4.2 Development and Deployment

Monolithic:

- Development Processes: Though simple at first, development in monolithic systems can get complicated as an application expands. Recompiling and redeploying the entire application could be necessary if changes are made to a single system component.
- Deployment Processes: As a monolithic program grows in size, deployment gets more complex. Updates need a complete redeployment of the program, which increases the likelihood of deployment failures and lengthens downtime.

Distributed:

- Development Processes: Distributed architectures facilitate the autonomous creation of services, enabling teams to concurrently work on various components. This leads to distributed development processes. Development cycles can accelerate thanks to this parallelism.
- Deployment Processes: Deployment is less hazardous and more adaptable. Deploying individual services separately can minimize downtime and isolate problems to particular parts of the system rather than the entire infrastructure.

4.3 Maintainability and Flexibility

Monolithic:

- Maintainability: Monolithic architecture reduces maintainability as components become more tightly coupled. Changes in one module can affect others, complicating debugging and updates.
- Flexibility: Monolithic systems lack flexibility. Adapting new technologies or making architectural changes is difficult, and often necessitates extensive refactoring.

Distributed:

- Maintainability: Distributed systems are easily maintainable. Services are decoupled, making it easier to update, test, and debug individual components without disrupting the entire system.
- Flexibility: These architectures are extremely adaptable. Individual services can independently adopt new technologies or methodologies, allowing for greater innovation and adaptation to changing requirements.

IJARSCT
ISSN (Online) 2581-9429
International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
Impact Factor: 7.53
Volume 4, Issue 3, June 2024

4.4 Fault Tolerance and Reliability

Monolithic:

- Fault Tolerance: Fault tolerance in monolithic systems is limited. A failure in one part of the application can cause the entire system to fail, making it less reliable.
- Reliability: Monolithic systems often lack redundancy and are less reliable under failure conditions, as the entire application relies on a single codebase and runtime environment.

Distributed:

- Fault Tolerance: Distributed architectures are designed for fault tolerance. Services run independently, so the failure of one service does not necessarily impact others, enhancing overall system reliability.
- Reliability: With built-in redundancy and isolation, distributed systems are more reliable. They can automatically reroute traffic and maintain functionality even when individual services fail.

4.5 Analysis

Label	Sample	Average	Median	90% line	95% line	99% line	Min	Max	Error	ThroughPut	Received KB/sec	Sent KB/se
Get Products	10000	1157	1246	1496	1537	3817	15	5524	0	133.4044824	8181.673273	16.28472685
Add Product	10000	1099	1236	1482	1511	1559	2	1615	0	133.6683954	31.32853018	32.24227898
Get Order	10000	1099	1239	1482	1513	1560	4	1622	0	133.386688	8014.897053	16.02203381
Add Order	10000	1102	1241	1479	1509	1564	13	1619	0	133.5006542	31.28921582	31.81070275
TOTAL	40000	1114	1241	1485	1515	1595	2	5524	0	530.9758008	16179.6631	95.79861249

Fig. 3. Aggregate Report of Monolithic Architecture

Label	sample	avg	median	90% line	95% line	99% line	min	max	error	throughput	received KB/sec	sent KB/sec
Get Products	18000	358	161	895	1182	2201	2	3506	0	145.3535321	3855.295464	17.74335109
Add Products	18000	303	106	831	1007	1289	1	2480	0	145.4733541	34.09531737	35.08976413
Add Orders	18000	60	32	142	247	442	1	533	0	145.4615981	34.09256206	34.66077143
Get Orders	18000	39	22	96	144	265	2	605	0	145.499224	3891.941866	17.47695757
TOTAL	72000	190	41	671	863	1425	1	3506	0	581.2170038	7808.826275	104.8631264

Fig. 4. Aggregate Report of Microservice Architecture

From Fig. 3 and Fig. 4 we infer that

- The average and median response times for the monolith architecture are generally lower than those for the microservices architecture. However, the 99th percentile response time for the monolith is significantly higher, indicating that in some cases, the monolith experiences very high response times.
- The monolith architecture exhibits a much higher error rate compared to the microservices architecture. This suggests that the monolith is less reliable and has stability issues under load.
- The monolith architecture has a higher throughput than the microservices architecture. This indicates that the monolith can handle a higher number of requests per second, likely due to reduced overhead from inter-service communication in microservices.
- The microservices architecture shows significantly higher data reception rates compared to the monolith. This could be due to the communication between services in a microservices architecture, which typically involves a higher volume of data transfer.
- For specific services, microservices generally have higher average response times but significantly lower error rates. The monolith architecture, while faster on average, has high error rates for all services.

V. CASE STUDIES

Monolithic Architecture Case Study:

Expanding Prime Video's Audio/Video Monitoring Services Problem

Prime Video provides thousands of live streams, and to ensure a consistent user experience, they need a tool to monitor each stream for audio/video quality issues. Their initial architecture, built with distributed microservices and serverless components, had limitations.

key problems included:

- High cost: Orchestration workflows and data transfer between components were both expensive.
- Limited Scalability: The architecture was unable to handle the desired number of concurrent streams (thousands).

Solution

Prime Video's VQA team redesigned the monitoring service from a distributed microservices architecture to a monolithic application.The key steps included:

- Consolidation: All components (media converter, defect detectors, and orchestration) were combined into a single process, which eliminated the need for data transfer between services and simplified the orchestration logic.
- Deployment: The service was deployed on scalable Amazon EC2 and Amazon ECS instances, taking advantage of cost-saving plans.
- Data Transfer: Video frames were kept in memory rather than being temporarily stored on Amazon S3, lowering the cost of Tier-1 S3 calls.
- Horizontal vs. Vertical Scaling: The initial architecture allowed for horizontal scaling of detectors (by adding more microservices), but the new approach has vertical scaling limitations. To address this, the team sets up multiple instances of the service, each handling a subset of detectors. A lightweight orchestration layer distributes customer requests between these instances.

Results

- Cost Reduction: The transition to a monolithic application resulted in a significant cost reduction (over 90%) for infrastructure.
- Improved Scalability: The service can now handle thousands of concurrent streams and has the potential for further scaling.
- Enhanced Monitoring: Prime Video can now monitor all streams, not just the ones with the most viewers, resulting in a better overall customer experience.

Distributed Architecture Case Study:

Distributed System for Netflix Streaming Service

Problem Statement

Netflix, the world's leading streaming service, faced significant challenges due to its monolithic architecture as it rapidly expanded its global subscriber base. The key issues were:

- Scalability: The monolithic system struggled to handle the exponential growth in user demand, especially during peak times when millions of users accessed the service simultaneously.
- Reliability: System outages and downtimes were becoming more frequent due to the single points of failure inherent in a monolithic architecture.
- Global Reach: Delivering a seamless and high-quality streaming experience to users worldwide requires a more robust and distributed infrastructure.

Solution

To address these issues, Netflix decided to transition from a monolithic architecture to a microservices-based distributed infrastructure.

This transition involved several strategic steps:

IJARSCT
ISSN (Online) 2581-9429
International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
Impact Factor: 7.53
Volume 4, Issue 3, June 2024

Microservices Adoption:

- Service decomposition involves breaking down a monolithic application into small, independent microservices, each responsible for a specific functionality (e.g., user authentication, content catalogue, streaming).
- Containerization: Used Docker to containerize microservices, ensuring consistent deployment environments.

Cloud Migration:

- AWS Cloud: Migrated to Amazon Web Services (AWS) to benefit from its scalable and globally distributed infrastructure.
- Elastic Load Balancing: Implemented Elastic Load Balancing to distribute traffic among servers and regions.

Data Management:

- NoSQL Databases: Adopted NoSQL databases like Amazon DynamoDB for high availability and low-latency data access.
- Distributed Caching: Implemented distributed caching systems like EVCache to store frequently accessed data.

Monitoring and Analytics:

- Centralized Logging: Used tools like Elasticsearch, Logstash, and Kibana (ELK Stack) for centralized logging and real-time analytics.
- Monitoring Tools: Deployed tools like Prometheus and Grafana to monitor system performance and health.

Results

The transition to a distributed microservices architecture resulted in significant improvements for Netflix. The key outcomes were:

- Enhanced Scalability: The system can handle millions of concurrent users, including peak traffic periods, without performance degradation.
- Improved Reliability: Eliminating single points of failure improved system reliability, leading to fewer downtimes and a better user experience.
- Global Performance: Netflix's use of CDNs and cloud infrastructure ensures high-quality streaming with minimal latency and buffering for users globally.
- Operational Agility: Netflix's microservices architecture enables faster development cycles, allowing for rapid innovation and risk-free feature deployment.

This case study illustrates how transitioning to a distributed microservices architecture can solve scalability and reliability issues, enabling organizations like Netflix to deliver superior service to a global audience.

Lessons Learned

Monolithic Architecture

- Simplicity and Integration: Monolithic architectures offer simplicity in terms of development and deployment when systems are relatively small and manageable. The tightly integrated nature ensures all components work seamlessly together.
- Scalability Challenges: As applications grow, the scalability of monolithic systems can become a bottleneck. Scaling often involves duplicating the entire application, leading to resource inefficiencies.
- Maintenance Difficulties: Managing a large monolithic codebase can be cumbersome, especially when implementing updates or changes. The risk of unintended consequences increases as the system becomes more complex.

Distributed Architecture

- Scalability and Flexibility: Distributed architectures excel in scalability, allowing individual services to be scaled independently based on demand. This flexibility is crucial for large, dynamic applications.
- Fault Isolation: One of the significant benefits of distributed systems is their ability to isolate faults. Failures in one service typically do not impact others, enhancing the overall reliability of the application.
- Complexity in Development and Deployment: While offering numerous advantages, distributed architectures introduce complexity in development, testing, and deployment. Coordinating multiple services and managing their interactions requires sophisticated tools and processes.
- Continuous Deployment: Distributed systems facilitate continuous deployment, enabling faster and more frequent updates. This agility helps organizations respond quickly to market changes and user needs.

By examining these real-world examples and extracting the lessons learned, we gain valuable insights into the practical implications of choosing between monolithic and distributed architectures. These case studies highlight the importance of aligning architectural decisions with organizational goals, system requirements, and the anticipated scale of operations.

VI. CONCLUSION

Summary of Findings

In this comparison of monolithic and distributed architectures, we looked at four key dimensions: performance and scalability, development and deployment, maintainability and flexibility, and fault tolerance and reliability. Monolithic architectures are simple to create and manage, providing reliable performance in less complex systems. However, they face scalability and fault tolerance issues. Distributed architectures, such as microservices and SOA, offer superior scalability, flexibility, and fault tolerance, but they are more complex to develop and deploy.

Recommendations

Choosing the right architecture is heavily influenced by your project's unique requirements and context. A monolithic architecture may be best suited for smaller applications or projects that require rapid development. It streamlines the development and deployment processes, making them easier to manage initially. However, if your application requires high scalability, flexibility to adapt to changing requirements, or high availability and fault tolerance, a distributed architecture is better suited. This method enables independent scaling of services, improved fault isolation, and the ability to deploy updates with minimal downtime.

Future Work

While this analysis provides a thorough overview, several areas require further investigation. Future research could look into the long-term maintenance costs of both architectures, how technological advancements affect these architectural choices, and how to create hybrid models that combine the advantages of both approaches. Furthermore, empirical studies on the performance of these architectures in various industries and application types would be useful for practitioners.

Organizations can make more informed decisions by understanding the trade-offs and strengths of monolithic and distributed architectures, ensuring that their architectural choices align with their strategic goals and operational requirements.

REFERENCES

[1]. G. Blinowski, A. Ojdowska and A. Przybyłek, "Monolithic vs. Microservice Architecture: APerformance and Scalability Evaluation," in IEEE Access, vol. 10, pp. 20357-20374, 2022, doi:10.1109/ACCESS.2022.3152803

[2]. Al-Debagy, O., &Martinek, P. (2018). A Comparative Review of Microservices and Monolithic Architectures. https://doi.org/10.1109/cinti.2018.8928192

[3]. O. Al-Debagy and P. Martinek, "A Comparative Review of Microservices and Monolithic Architectures," 2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI), 2018, pp. 000149-000154, doi: 10.1109/CINTI.2018.8928192.

[4]. K. Gos and W. Zabierowski, "The Comparison of Microservice and Monolithic Architecture," 2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH), 2020, pp. 150-153, doi: 10.1109/MEMSTECH49584.2020.9109514.

[5]. From Monolithic Systems to Microservices: A Comparative Study of Performance by Freddy Tapia 1,2,*ORCID,Miguel Ángel Mora 2ORCID,Walter Fuertes 1ORCID,HernánAules 1,3,*ORCID,Edwin Flores 1ORCID andTheofilosToulkeridis

[6]. Tanenbaum, A. S., & van Steen, M. (2007). Distributed Systems: Principles and Paradigms. Pearson Education. ISBN: 9780132392273.

[7]. Cockcroft, A. (2012). Migrating to Cloud-Native Microservices: A Netflix Case Study. QCon San Francisco. Retrieved from https://www.infoq.com/presentations/Netflix-Architecture/

[8]. Gopalakrishnan, M. (2015). Distributed Systems at Netflix. Communications of the ACM, 58(3), 78-87. doi:10.1145/2677033

[9]. Kolny, M. (2023, March 22). Scaling up the Prime Video audio/video monitoring service and reducing costs by 90% - Prime Video Tech. Prime Video Tech. https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90