Software Architecture Evolution: Patterns, Trends, and Best Practices

Nithish Nadukuda

Paypal Inc.

Email: ninadukuda@gmail.com

Abstract

Software Architecture is a fundamental aspect in software development, influencing the structure, scalability, and maintainability of applications. This article considers the evolution of software architecture, emphasizing key patterns, trends, and best practices that have emerged over time. From monolithic architectures to microservices and from client-server to cloud-native approaches, various architectural paradigms are examined alongside their advantages and challenges. Additionally, the article discusses emerging trends such as serverless computing, event-driven architecture, and containerization, showcasing their impact on modern software systems. Best practices for designing resilient, scalable, and flexible architectures are presented, stressing the importance of modularity, decoupling, and automation. By understanding the evolution of software architecture and embracing current trends and best practices, developers can construct robust, adaptable, and future-proof software systems.

Keywords: Software architecture, monolithic architectures, microservices, client-server, cloud-native, serverless computing, event-driven architecture, containerization, modern software systems, resilience, modularity, automation.

1. Introduction

Software architecture stands as the cornerstone of system development, delineating the structure, organization, and behavior of software components within an application. Over time, as technological landscapes evolve and business needs morph, software architecture undergoes a continuous process of evolution to stay aligned with these changes. This introduction delves into the significance of software architecture in system development and dissects the evolutionary trends shaping the field.

Importance of Software Architecture in System Development

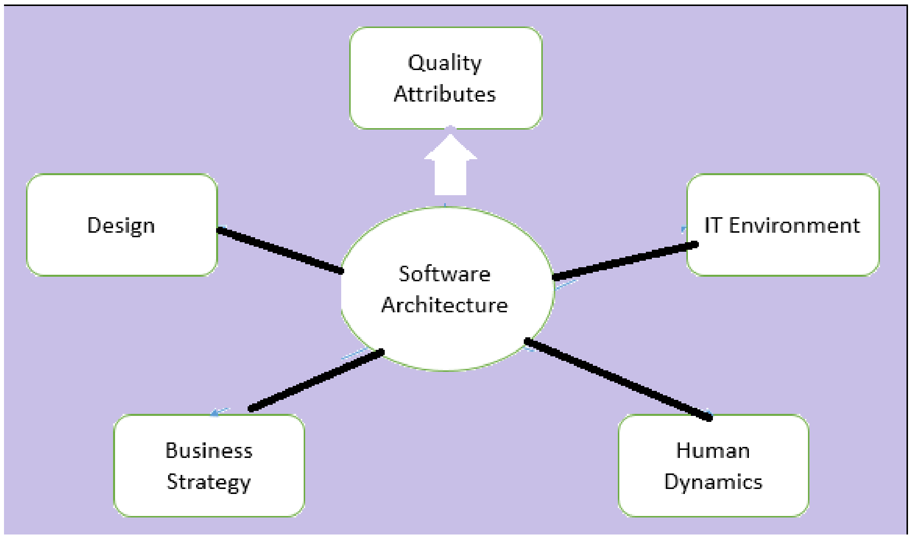
Foundation for System Design: Software architecture lays the groundwork for designing intricate software systems. It offers a blueprint that guides developers in structuring and implementing software components efficiently, ensuring coherence and clarity in the overall system design.

Scalability and Maintainability: Well-crafted software architecture fosters scalability and maintainability within systems. By delineating clear boundaries between components and adhering to modular design principles, architectures can scale gracefully as demands grow and remain manageable over time, easing maintenance efforts and reducing technical debt.

Risk Mitigation: Robust software architecture acts as a bulwark against technical risks by identifying potential pitfalls early in the development lifecycle. By conducting thorough architectural assessments and risk analyses, organizations can predictively address issues, safeguarding projects from costly setbacks and delays.

Alignment with Business Goals: Software architecture serves as a conduit for aligning technical decisions with overarching business objectives. By encapsulating business logic and requirements within architectural blueprints, organizations can ensure that software systems are purpose-built to deliver tangible value and meet the needs of stakeholders effectively.

Fig 1: Software Architecture



Evolutionary Trends in Software Architecture

Shift from Monolithic to Microservices: The paradigm shift towards microservices architecture marks a departure from monolithic application design. By decomposing large, monolithic systems into smaller, independently deployable services, microservices architecture promotes agility, fault isolation, and scalability, enabling organizations to respond swiftly to evolving business requirements.

Adoption of Cloud-Native Architecture: Cloud-native architecture embodies a holistic approach to application development and deployment, leveraging cloud computing resources and services to maximize agility, scalability, and cost-efficiency. By embracing cloud-native principles such as containerization, infrastructure as code, and continuous delivery, organizations can capitalize on the scalability and flexibility offered by cloud platforms.

Rise of Serverless Computing: Serverless computing heralds a paradigm shift in application development, abstracting away infrastructure management and allowing developers to focus solely on writing code. By leveraging serverless platforms, organizations can streamline development workflows, reduce operational overhead, and achieve rapid time-to-market for new features and functionalities.

Embrace of Event-Driven Architecture: Event-driven architecture advocates for asynchronous communication between software components, enabling real-time processing and responsiveness to business events. By decoupling components and facilitating loose coupling through event-driven communication, organizations can build highly scalable, resilient systems capable of handling diverse workloads and fluctuating demand.

Containerization with Docker and Kubernetes: Containerization technologies such as Docker and Kubernetes have emerged as game-changers in the realm of software deployment and orchestration. By encapsulating applications and their dependencies within lightweight, portable containers, organizations can achieve consistency across development, testing, and production environments, while benefiting from enhanced scalability, resource utilization, and deployment automation.

2. Literature Review

Nacha Chondamrongkul and Jing Sun (2023)

This work likely discusses automated planning techniques for managing functional changes in software evolutionary architecture. It may delve into methodologies and algorithms aimed at facilitating the evolution of software systems while ensuring architectural integrity and functionality.

S Roselin Mary and Dr. Paul Rodrigues (2012)

This paper considers the evolution and evaluation of software architecture, covering topics such as architectural design principles, patterns, and evaluation methodologies. It may provide insights into the factors influencing software architecture evolution and approaches for assessing architectural quality.

Hongyu Pei Breivold (2009)

This thesis investigates software architecture evolution and its impact on software evolvability. It may discuss theoretical frameworks, models, and techniques for managing architectural changes and improving the adaptability of software systems over time.

Andrade, L.F., Fiadeiro, J.L. (2003)

This chapter considers the use of architecture-based approaches for evolving software systems. It may discuss techniques for leveraging architectural models to guide the evolution process, manage complexity, and maintain system coherence and consistency.

David Garlan and Mary Shaw (1994)

This seminal work provides a comprehensive introduction to software architecture, covering its definition, principles, and significance in software engineering. It may discuss architectural styles, patterns, and approaches for designing and analyzing software architectures.

3. Foundational Patterns in Software Architecture

Software architecture evolution is a dynamic process shaped by the emergence, adoption, and refinement of various architectural patterns. These patterns encapsulate proven solutions to recurring design problems and serve as foundational principles for architecting robust, scalable, and maintainable software systems. Common architectural patterns and their evolution over time, along with their profound impact on software architecture.

Common Architectural Patterns and Their Evolution

Layered Architecture:

Layered architecture organizes software components into horizontal layers, each responsible for a specific set of functionalities. Common layers include presentation, business logic, and data access layers.

The layered architecture pattern has evolved to accommodate modern development practices and architectural styles. Variations such as the Hexagonal Architecture and Clean Architecture have emerged, emphasizing separation of concerns, testability, and dependency inversion. These variations address the limitations of traditional layered architectures and provide more flexible and maintainable solutions for complex systems.

Client-Server Architecture:

Client-server architecture distributes the responsibilities between client devices and server systems. Clients request services from servers, which process and fulfill these requests.

The advent of web-based applications and the proliferation of mobile devices have led to the refinement of client-server architecture. Architectural patterns such as Representational State Transfer (REST) and GraphQL have emerged to facilitate scalable and efficient communication between clients and servers. These patterns promote loose coupling, scalability, and resource efficiency, enabling the development of modern, distributed systems that can seamlessly adapt to evolving user demands and technological advancements.

Component-Based Architecture:

Component-based architecture decomposes systems into reusable and replaceable software components, promoting modularity and reusability.

The evolution of component-based architecture has seen the rise of frameworks and platforms that facilitate component-based development, such as Java Enterprise Edition (Java EE), .NET Framework, and Spring Framework. Additionally, microservices architecture can be viewed as an evolution of component-based architecture, with services serving as independent, self-contained components. Microservices architecture offers finer-grained modularity, enhanced scalability, and better support for continuous deployment and delivery, enabling organizations to build agile, resilient, and scalable systems in today's fast-paced, dynamic environments.

Microservices Architecture:

Microservices architecture decomposes applications into a set of small, independently deployable services, each responsible for a specific business capability.

Microservices architecture has evolved to address challenges such as service discovery, communication, and orchestration. Patterns such as service mesh, event sourcing, and CQRS (Command Query Responsibility Segregation) have emerged to address these challenges and enhance the resilience and scalability of microservices-based systems. These patterns provide solutions for managing service dependencies, ensuring data consistency, and optimizing communication between services, enabling organizations to build distributed, resilient, and scalable architectures that can meet the demands of modern, cloud-native applications.

Impact of Design Patterns on Software Architecture

Modularity and Reusability:

Design patterns promote modularity by encapsulating related functionalities within cohesive units, enabling easier maintenance and evolution of software systems.

By facilitating the reuse of proven design solutions, design patterns contribute to the development of scalable and maintainable software architectures. They enable developers to encapsulate common functionalities, such as object creation, delegation, and composition, within reusable components, promoting code reuse and reducing redundancy.

Flexibility and Extensibility:

Design patterns provide flexible and extensible solutions to common design problems, allowing software architectures to adapt to changing requirements and accommodate future enhancements.

Patterns such as the Strategy Pattern and Adapter Pattern enable the pluggability of algorithms and components, promoting flexibility and extensibility in software architectures. They provide mechanisms for encapsulating algorithmic variations and adapting interfaces, allowing systems to accommodate new functionalities and integrate with external services or components seamlessly.

Scalability and Performance:

Certain design patterns, such as the Observer Pattern and the Proxy Pattern, can enhance the scalability and performance of software systems by promoting asynchronous communication and lazy loading of resources.

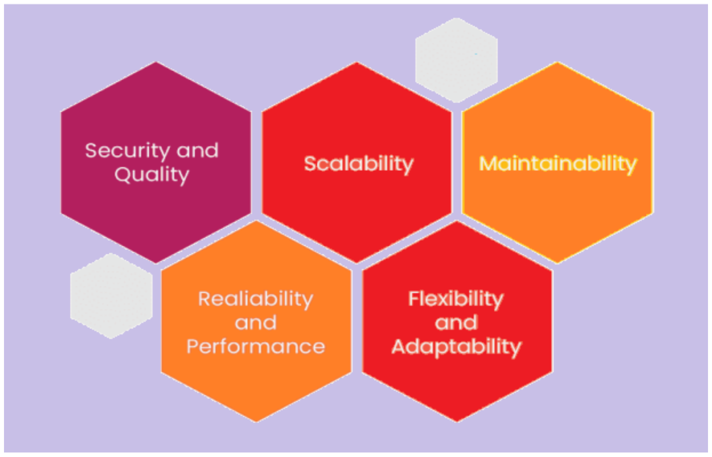
By decoupling components and promoting loose coupling, design patterns facilitate the development of scalable and resilient architectures capable of handling varying workloads and demands. Patterns such as the Observer Pattern enable event-driven communication between components, allowing systems to react to changes and events asynchronously, while the Proxy Pattern enables the lazy initialization and caching of resources, optimizing resource utilization and improving system performance.

Maintainability and Testability:

Design patterns promote maintainability and testability by improving the clarity, readability, and organization of software code.

Patterns such as the Singleton Pattern and Factory Pattern encapsulate object creation logic and promote the separation of concerns, making codebases easier to maintain and test. They provide clear and standardized solutions to common design problems, reducing ambiguity and cognitive overhead for developers. By adhering to design patterns, developers can write cleaner, more modular, and more testable code, facilitating easier debugging, maintenance, and evolution of software systems.

Fig 2: Significance of Software Architecture



4. Emerging Trends in Software Architecture

Software development, advancing trends in architecture continually reshape how applications are designed, developed, and deployed. Two significant trends driving this evolution are Microservices Architecture and Cloud-Native Architecture. These trends in detail, featuring their impact on software development and scalability.

Microservices Architecture and its Impact

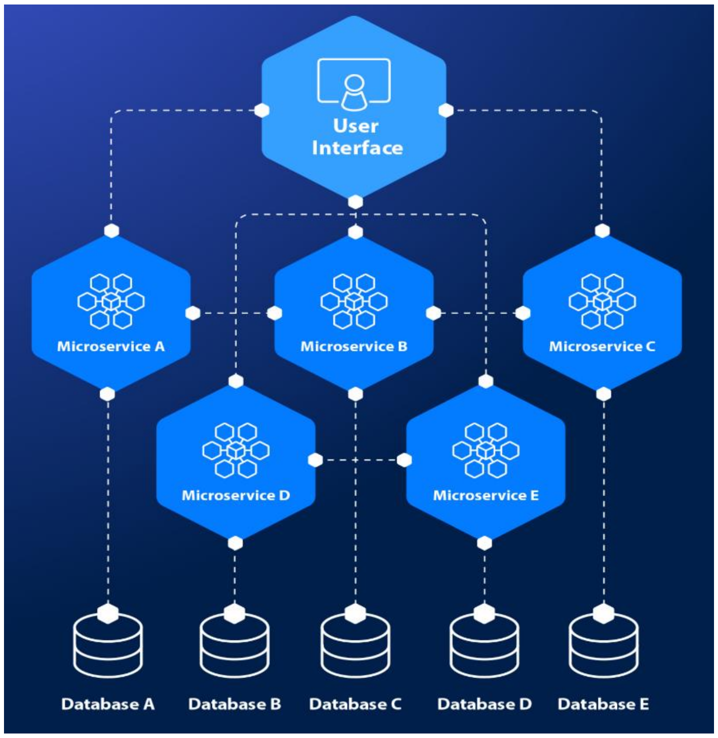
Microservices Architecture:

Microservices architecture is a design approach where an application is composed of loosely coupled services, each running independently and communicating through well-defined interfaces.

Each microservice focuses on a specific business capability, allowing teams to develop, deploy, and scale components autonomously.

Microservices are typically deployed in containers and communicate via lightweight protocols such as HTTP/REST or messaging queues.

Fig 3: Microservices Architecture



Impact on Software Development:

Scalability: Microservices architecture enables horizontal scalability by allowing individual services to scale independently based on demand. This granular scalability ensures efficient resource utilization and optimal performance, even in fluctuating or unpredictable conditions.

Resilience: The decentralized nature of microservices architecture enhances resilience by isolating failures. If one service encounters issues, it doesn't affect the entire system, as other services can continue to function independently. This fault isolation minimizes downtime and improves overall system reliability.

Agility: Microservices promote agility and flexibility in software development. Teams can iterate rapidly on individual services, releasing updates independently without impacting other parts of the application. This agility enables organizations to respond quickly to market changes and deliver new features more frequently.

Technology Diversity: Microservices architecture encourages technology diversity within an application. Each service can be developed using different programming languages, frameworks, or databases, based on its specific requirements. This flexibility empowers teams to choose the best tools for each service, optimizing performance and productivity.

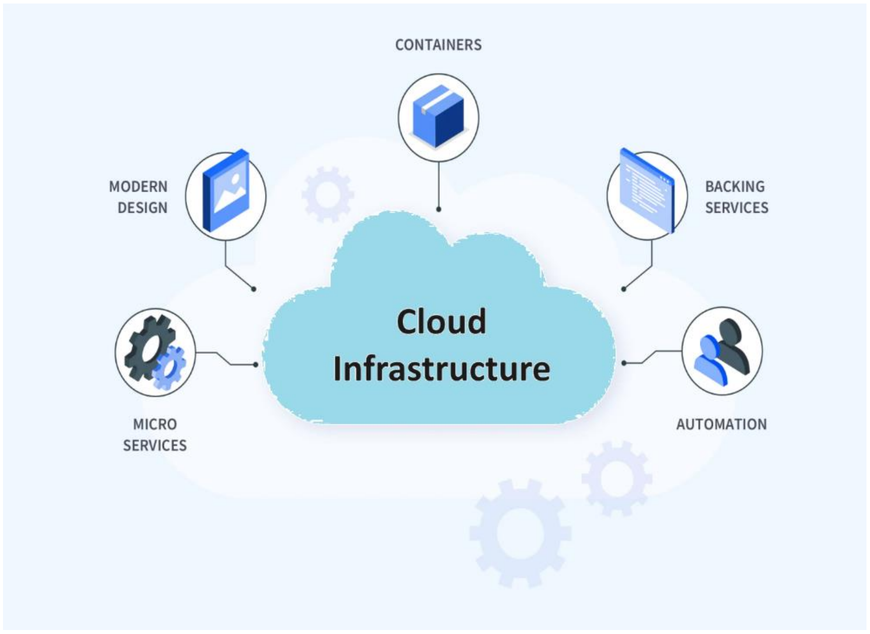
Cloud-Native Architecture and Scalability

Cloud-Native Architecture:

Cloud-native architecture is an approach to building and running applications that fully leverage the benefits of cloud computing platforms.

It emphasizes containerization, microservices, infrastructure as code, and continuous delivery, enabling organizations to develop scalable, resilient, and adaptable systems in cloud environments.

Fig 4: Cloud-Native Architecture:



Impact on Scalability:

Auto-scaling: Cloud-native architectures leverage auto-scaling capabilities provided by cloud platforms to adjust resources dynamically based on workload demand. This automatic scaling ensures optimal performance and cost-efficiency, as resources are provisioned or de-provisioned in real-time as needed.

Elasticity: Cloud-native architectures embrace elasticity, enabling systems to scale resources dynamically in response to changes in workload or traffic. This elasticity ensures that applications can handle sudden spikes in demand without sacrificing performance or reliability, maintaining a seamless user experience.

Microservices and Scalability: Microservices architecture is often a central component of cloud-native architectures. Each microservice can be scaled independently based on its specific resource requirements, enabling fine-grained control over scalability and resource allocation. This flexibility ensures that resources are allocated efficiently, optimizing performance and cost-effectiveness.

Distributed Computing: Cloud-native architectures embrace distributed computing principles, enabling applications to leverage the scalability and fault-tolerance of distributed systems. By distributing workloads across multiple nodes or instances, cloud-native applications can achieve higher levels of scalability and resilience than traditional monolithic architectures.

5. Best Practices in Software Architecture Design

Software architecture design is a cornerstone of building reliable, scalable, and maintainable software systems. By following best practices and principles, architects can ensure that their architectures meet the demands of modern applications. Best practices in software architecture design, emphasizing principles for creating robust and scalable architectures, as well as design patterns for fostering flexibility and maintainability.

Principles for Robust and Scalable Architectures

Modularity:

Break down the system into modular components, each handling a specific functionality or business concern.

Modularity promotes reusability, simplifies maintenance, and facilitates scalability by allowing components to be developed, tested, and deployed independently.

Layered Architecture:

Organize the system into layers, with each layer responsible for a specific aspect of functionality (e.g., presentation, business logic, data access).

Layered architecture promotes separation of concerns, making the system easier to understand, maintain, and evolve. It also facilitates scalability and flexibility by providing clear boundaries between different parts of the system.

Loose Coupling:

Minimize dependencies between components to reduce the impact of changes and improve flexibility.

Loose coupling makes it easier to modify and extend the system without affecting other components. It also promotes reusability and testability by enabling components to be replaced or modified independently.

High Cohesion:

Ensure that components within the same module or layer are closely related and serve a common purpose.

High cohesion improves the clarity and maintainability of the system by grouping related functionality together. It also reduces the risk of unintended side effects when modifying or extending components.

Scalability:

Design the architecture to scale horizontally by adding more instances or nodes to handle increased load.

Scalable architectures can accommodate growing user bases and handle spikes in traffic without sacrificing performance or reliability. Techniques such as load balancing, caching, and asynchronous processing can help distribute workload and maintain responsiveness under heavy loads.

Fault Tolerance:

Build resilience into the architecture by anticipating and mitigating failures at various levels of the system.

Fault-tolerant architectures can withstand hardware failures, network issues, and other disruptions without compromising service availability or data integrity. Strategies such as redundancy, failover mechanisms, and graceful degradation ensure that the system remains operational even in adverse conditions.

Design Patterns for Flexibility and Maintainability

Factory Method Pattern:

Define an interface for creating objects, but let subclasses decide which class to instantiate. This pattern promotes flexibility by allowing the creation of objects to be deferred to subclasses.

The Factory Method Pattern enables the system to be extended with new object types without modifying existing code. It also encapsulates object creation logic, making the system more maintainable and modular.

Strategy Pattern:

Define a family of algorithms, encapsulate each one, and make them interchangeable. This pattern allows algorithms to vary independently from clients that use them.

The Strategy Pattern enables dynamic selection of algorithms at runtime, making the system more flexible and adaptable to changing requirements. It also promotes code reuse and maintainability by encapsulating algorithmic logic in separate classes.

Observer Pattern:

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

The Observer Pattern facilitates loose coupling between components by decoupling the sender (subject) from the receivers (observers). It enables objects to communicate without being aware of each other's identities, promoting reusability and maintainability.

Adapter Pattern:

Convert the interface of a class into another interface that clients expect. This pattern allows incompatible interfaces to work together.

The Adapter Pattern enables integration of incompatible components or systems by providing a bridge between them. It promotes interoperability and reusability by allowing existing code to be reused without modification.

Singleton Pattern:

Ensure that a class has only one instance and provide a global point of access to that instance.

The Singleton Pattern ensures that there is only one instance of a class throughout the system, making it useful for managing resources, configuration settings, or shared state. It promotes consistency and centralization, simplifying access to shared resources and reducing memory overhead.

Decorator Pattern:

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

The Decorator Pattern allows behavior to be added or removed from objects at runtime, without affecting other objects of the same class. It promotes flexibility and maintainability by enabling objects to be extended with new functionality without modifying their structure or behavior.

6. Case Studies on Software Architecture Evolution

Real-world case studies provide invaluable insights into the complexities and successes of architectural evolution within organizations. These examples not only demonstrate the challenges faced during transformation but also showcase the strategies employed and the lessons learned along the way. Case studies of architectural evolution and extract key learnings from each scenario.

Real-World Examples of Architectural Evolution

Netflix: Transition to Microservices:

Netflix, faced with scalability and resilience challenges in its monolithic architecture, embarked on a journey to transition to a microservices-based architecture.

Netflix decomposed its monolithic application into hundreds of microservices, each responsible for specific business functions. They utilized containerization technologies like Docker and orchestration platforms like Kubernetes to manage their microservices ecosystem.

This architectural shift empowered Netflix to scale globally, handle millions of concurrent users, and roll out new features seamlessly. Failures in one service no longer caused system-wide outages, significantly improving fault tolerance and resilience.

Amazon: Evolution of AWS Architecture:

Amazon Web Services (AWS) started as a monolithic application but faced scalability challenges as demand grew. AWS transitioned to a service-oriented architecture (SOA) and later embraced microservices to address scalability and maintenance issues.

AWS decomposed its monolithic application into a suite of loosely coupled services, each offering distinct cloud computing capabilities. They leveraged containerization, auto-scaling, and distributed systems principles to build a resilient and scalable infrastructure.

The evolution of AWS architecture enabled Amazon to establish itself as a leader in cloud computing, offering a vast array of services to millions of customers worldwide. The modular architecture allowed for rapid innovation, with teams independently developing and deploying services.

Lessons Learned from Architectural Transformations

Incremental Approach and Rapid Iteration:

Start with small, incremental changes and iterate rapidly based on feedback. Breaking down the transformation into manageable phases reduces risks and enables course corrections as needed.

Alignment with Business Objectives:

Ensure that architectural transformations align with strategic business goals and customer needs. Focus on initiatives that deliver tangible value, such as improving scalability, reliability, or time-to-market for new features.

Investment in Automation and Tooling:

Invest in automation and tooling to streamline the development, deployment, monitoring, and testing of new architectures. Automation reduces manual effort, accelerates delivery, and ensures consistency.

Empowerment of Cross-Functional Teams:

Foster collaboration among cross-functional teams and empower them with the autonomy to make decisions and implement changes. Encourage a culture of innovation, accountability, and shared ownership.

Continuous Learning and Improvement:

Embrace a culture of continuous learning and improvement, where teams are encouraged to experiment, learn from failures, and adapt their approaches based on evolving requirements and technologies.

Data-Driven Decision-Making:

Monitor key metrics related to system performance, availability, and user satisfaction to assess the impact of architectural changes. Use data-driven insights to identify areas for optimization and refinement, ensuring ongoing evolution and alignment with organizational goals.

7. Future Directions in Software Architecture

The landscape of software architecture is in a constant state of evolution, driven by advancements in technology, changing business needs, and emerging paradigms in software development. By anticipating future trends and innovations in software architecture, organizations can position themselves to adapt and thrive in an ever-changing environment. Predicted trends and innovations in software architecture and delves into their implications for future software development practices.

Predicted Trends and Innovations in Software Architecture

Event-Driven Architectures (EDA):

Event-driven architectures enable systems to react to events and triggers in real-time, fostering responsiveness and flexibility. This approach promotes loose coupling between components and facilitates asynchronous communication.

EDA is expected to gain traction as organizations seek to build scalable, resilient, and adaptable systems. By decoupling components and leveraging asynchronous communication, EDA enables applications to handle dynamic workloads and evolving requirements effectively.

Serverless Computing:

Serverless computing abstracts infrastructure management away from developers, allowing them to focus solely on writing application code. With serverless platforms, developers no longer need to provision or manage servers, as the platform automatically scales resources based on demand.

Serverless architectures offer numerous benefits, including reduced operational overhead, rapid scalability, and cost-efficiency. As serverless platforms mature and expand their capabilities, organizations are likely to adopt this approach for building and deploying applications more efficiently.

Edge Computing:

Edge computing brings computation and data storage closer to the data source or end-user, reducing latency and enabling real-time processing of data. Edge architectures are particularly relevant in scenarios where low latency and high throughput are critical, such as IoT, autonomous vehicles, and augmented reality.

Edge computing enables organizations to deliver low-latency and high-performance applications by processing data closer to the point of origin. As the proliferation of IoT devices and edge computing infrastructure continues, organizations will increasingly leverage edge architectures to deliver immersive and responsive user experiences.

AI-Driven Architectures:

AI-driven architectures leverage machine learning and artificial intelligence techniques to optimize system performance, automate decision-making processes, and deliver personalized experiences. These architectures analyze large volumes of data in real-time to derive insights and make informed decisions.

AI-driven architectures have the potential to revolutionize various industries by enabling intelligent automation, predictive analytics, and enhanced user experiences. As AI technologies become more accessible and mature, organizations will increasingly integrate AI capabilities into their architectures to drive innovation and competitive advantage.

Implications for Future Software Development Practices

Agility and Flexibility:

Future software architectures will prioritize agility and flexibility to accommodate evolving business needs and technological advancements. Organizations will adopt iterative development practices, such as Agile and DevOps, to enable rapid experimentation, feedback, and adaptation.

Resilience and Fault Tolerance:

With the increasing complexity and distribution of systems, resilience and fault tolerance will become paramount. Future architectures must embrace fault isolation, redundancy, and graceful degradation strategies to ensure system availability and reliability in the face of failures or disruptions.

Decentralization:

Decentralized architectures, such as blockchain and distributed ledgers, will play a significant role in future software development. These architectures enable peer-to-peer interactions, data integrity, and decentralized governance models, particularly in applications requiring trust, security, and transparency.

Security by Design:

Security will be integrated into the fabric of software architecture from the outset, rather than treated as an afterthought. Future architectures will employ robust security measures, including encryption, authentication, and authorization mechanisms, to protect data and mitigate cybersecurity risks.

Ethical and Responsible AI:

As AI-driven architectures become more prevalent, organizations must prioritize ethical AI practices, transparency, fairness, and accountability. It is crucial to ensure that AI systems uphold societal values, respect user privacy and rights, and mitigate the risks of bias and discrimination.

Continuous Learning and Upskilling:

Software developers and architects must embrace continuous learning and upskilling to stay abreast of emerging technologies and architectural paradigms. Lifelong learning and a growth mindset will be essential for navigating the evolving landscape of software architecture effectively.

8. Conclusion

In conclusion, software architecture evolution is influenced by technology, business needs, and emerging trends. By prioritizing agility, resilience, and security, organizations can adapt to future challenges and drive innovation in software development.

References

Andrade, L.F., Fiadeiro, J.L. (2003). Architecture Based Evolution of Software Systems. In: Bernardo, M., Inverardi, P. (eds) Formal Methods for Software Architectures. SFM 2003. Lecture Notes in Computer Science, vol 2804. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-39800-4_8

David Garlan and Mary Shaw. January 1994. An Introduction to Software Architecture. School of Computer Science, Carnegie Mellon University

Hongyu Pei Breivold, et al. A systematic review of software architecture evolution research. Information and Software Technology. Volume 54, Issue 1, January 2012, Pages 16-40. https://doi.org/10.1016/j.infsof.2011.06.002

Hongyu Pei Breivold. 2009. SOFTWARE ARCHITECTURE EVOLUTION AND SOFTWARE EVOLVABILITY. Malardalen University Press Licentiate Theses. No. 97. https://citeseerx.ist.psu.edu/document?doi=7b9d57d41e8b88ec8f8ca3f9da1a91598c6c0f04&repid=rep1&type=pdf

Jeffrey M. Barnes. Software Architecture Evolution. December 2013. CMU-ISR-13-118. Institute for Software Research, School of Computer Science, Carnegie Mellon University.

Mohsin, A. (2021). A framework to support the stochastic software architecture modeling and analysis for system of systems. https://ro.ecu.edu.au/theses/2472

Qin (2008). Introduction to Software Architecture. In: Software Architecture. Advanced Topics in Science and Technology in China. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-74343-9_1

S Roselin Mary and Dr.Paul Rodrigues, “Software Architecture- Evolution and Evaluation” International Journal of Advanced Computer Science and Applications(IJACSA), 3(8), 2012. http://dx.doi.org/10.14569/IJACSA.2012.030814

Wu, Y., Peng, X., Zhao, W. (2011). Architecture Evolution in Software Product Line: An Industrial Case Study. In: Schmid, K. (eds) Top Productivity through Software Reuse. ICSR 2011. Lecture Notes in Computer Science, vol 6727. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-21347-2_11