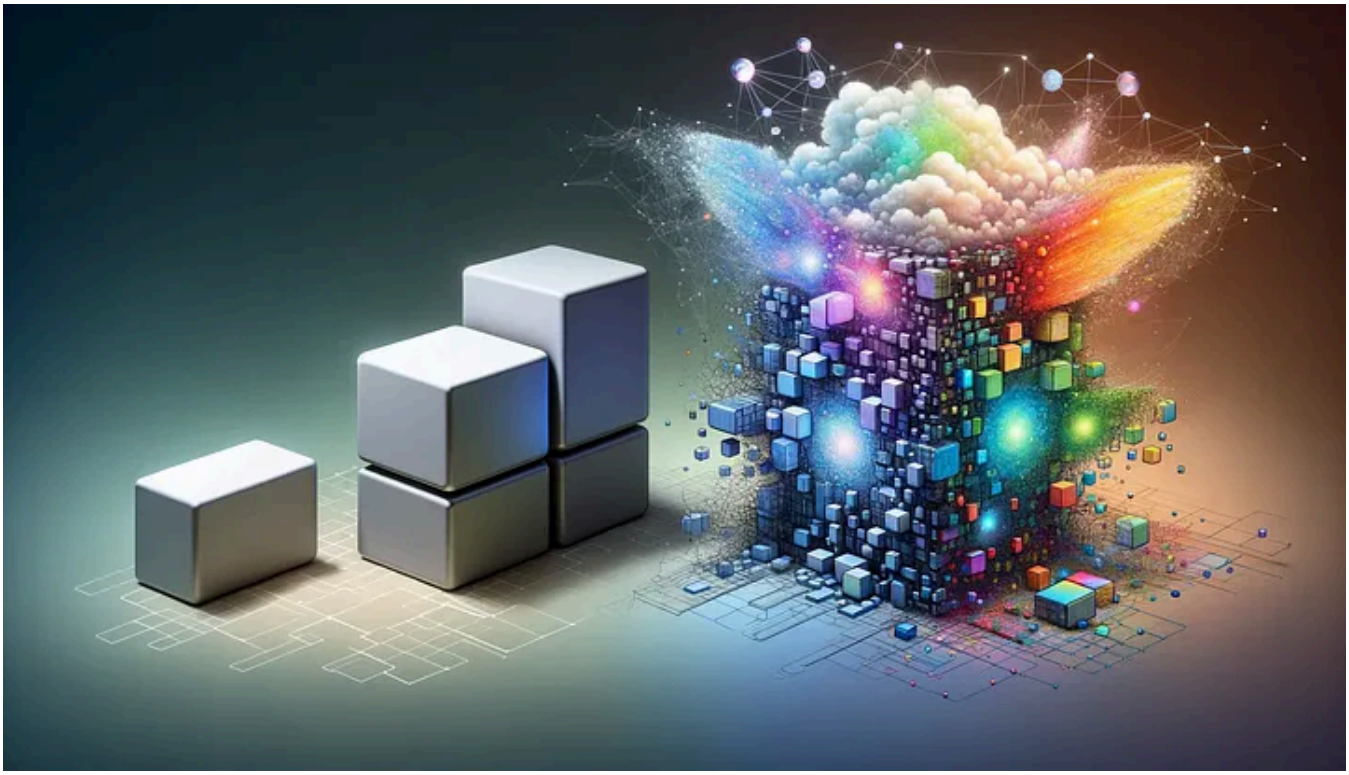
Open in app ↗

Search



# Evolution of Software Architecture : From Monoliths to Microservices and Beyond

Pier-Jean Malandrino
Published in Scub-Lab
17 min read · Nov 27, 2023

Listen Share More

In the vast and ever-evolving domain of software development, the architecture of software systems stands as a pivotal aspect, shaping not only how applications are built and maintained but also how they adapt to changing technological landscapes and business needs. This paper embarks on an exploratory journey through the evolution of software architecture, tracing its progression from the early days of monolithic designs to the contemporary era of microservices and serverless architectures. We delve into the fundamental shifts in architectural patterns, examining how each has been influenced by and has responded to the

advancements in technology, the growing complexity of applications, and the evolving requirements of businesses.

Our exploration begins with monolithic architectures, the bedrock of early software development, characterized by their unified and indivisible nature. We then transition to modular designs, heralding a new era of software architecture that emphasizes separation of concerns and encapsulation. Following this, we explore the emergence of Service-Oriented Architecture (SOA), a paradigm shift that underscores service reuse and interoperability. The narrative progresses to the rise of microservices architecture, a fine-grained approach building on the principles of SOA but with a greater emphasis on independence and scalability. Our journey extends to the realm of serverless computing, a paradigm that further abstracts and simplifies (or not ?) architectural complexities.

Throughout this exploration, we also address a critical aspect of modern software architecture — the escalating infrastructure costs associated with increasingly decentralised systems that become acceptable with the emergence of powerful modern infrastructure. This paper aims not only to provide a historical perspective on software architecture but also to highlight the importance of context-driven decision-making in the face of these evolving paradigms. By understanding the strengths, limitations, and suitable application contexts of each architectural style, we can better navigate the complex landscape of software development, ensuring that our architectural choices are both technologically sound and aligned with the strategic goals of the organizations and projects they serve.

In sum, this paper offers an overview of the evolution of software architecture, illuminating the path from monolithic simplicity to the complex yet flexible world of microservices and beyond, while emphasizing the need for thoughtful, context-aware architectural decisions in the face of ever-changing technological and business environments.

## What is Software Architecture ?

In the context of this paper, *"Software Architecture"* refers to the fundamental organization of a software system, encompassing its components, the relationships between them, and the principles guiding its design and evolution. It is a blueprint that defines the structure, behavior, and more importantly, the logical integrity of the software, ensuring that it meets both technical and business requirements.

Software architecture goes beyond the mere selection of technological tools; it involves strategic decision-making about how to best structure and interconnect various parts of an application to achieve desired performance, scalability, maintainability, and other critical attributes. This includes considerations like how data flows through the system, how components communicate, how they are deployed, and how they can be scaled and maintained over time.

In essence, software architecture is about creating a cohesive and coherent framework that not only supports the functional needs of the software but also aligns with broader business goals and adapts to the ever-evolving technological landscape. It is the foundation upon which software's reliability, efficiency, and adaptability are built, and it plays a crucial role in determining a software project's long-term success and viability.

## The Early Days — Monolithic Architectures

### Timeframe:

Roughly from the 1960s to the late 1980s.

### Key Contributor

Monolithic architectures have been the default since the early days of computing, so it's challenging to attribute them to a single person. However, companies like IBM were instrumental in defining early software architecture through their development of mainframe computers in the 1960s and 1970s.

### What is Monolithic Architecture?

Monolithic architecture represents an early approach in software design where an application is developed as a single, indivisible unit. It combines various components such as the user interface, business logic, and data access layers into a unified codebase. This structure, prevalent in the initial phase of software development, necessitated deploying the entire application as one cohesive entity.

### Link with Early Software Development Technology

During the early stages of software engineering, technology was markedly different, characterized by simpler and less powerful hardware, limited networking capabilities, and nascent development tools. Applications were generally smaller and less complex, managed by smaller, centralized teams. The monolithic architecture fit well within this context, where modular programming and distributed systems were still in their infancy.

Security concerns were also not a big issue because there were fewer people using it and the data was not as valuable as it is today.

#### Benefits of Monolithic Architecture

The monolithic approach offered several advantages in its time:

**Simplicity in Development and Deployment:** The unified codebase and development environment simplified the development, testing, and deployment processes, particularly important when tools and operational expertise were limited. It is also necessary that we are not talking about a “web application”, which means that you would have to deploy directly on devices, taking care of OS management, etc.

**Performance Efficiency:**

*Adapted to Limited Resources:* Monolithic architectures were efficient because they were designed to operate within the constraints of less powerful hardware. They had to ensure that systems worked effectively with limited memory and CPU resources.

*Simpler Applications:* Applications were simpler and less demanding, so a single, unified application structure was sufficient and practical for the available technology.

#### Trade-offs of Monolithic Architecture

Despite its benefits, monolithic architecture came with trade-offs:

**Scalability Issues:** As applications grew in complexity, scaling monolithic applications became challenging.

**Flexibility Limitations:** Implementing new technologies or making significant changes within a monolithic application could be cumbersome.

**Deployment Challenges:** With all components tied into a single unit, even small changes required redeploying the entire application, leading to longer downtime and potential risks.

## The Shift to Modular Design

#### Timeframe

Gaining traction in the 1970s, with a significant rise in the 1980s and 1990s.

### Key Contributor

The concept of modularity in software design gained traction in the 1970s with the publication of David Parnas' seminal paper on modular program structure in 1972, which laid the theoretical foundation for this approach.

### What is Modular Design in Software Architecture?

Modular design in software architecture is a forward-thinking approach that revolutionizes the structure and development of software systems. It involves breaking down a system into distinct, manageable modules, each responsible for a specific functionality. This philosophy is anchored on principles like modularity, encapsulation, and separation of concerns. Modularity divides software into smaller parts, encapsulation conceals each module's internal operations, and separation of concerns ensures each module uniquely addresses an aspect of the software's functionality.

### Evolving Software Complexity and Practices

The move toward modular design was primarily driven by the growing complexity of software systems and the inherent limitations of monolithic architectures. As software applications grew in size and complexity, the need for more manageable, maintainable, and scalable architectures became evident.

### Early Software Engineering Practices

During this period, the focus was on improving software design principles and methodologies. Concepts like structured programming and later object-oriented programming (which became widely adopted in the 1980s and 1990s) played a significant role in shaping modular design. These practices emphasized breaking down software into manageable, logically distinct components, thus paving the way for modular architectures. It is important to understand that most of the concepts that seem obvious today had to be invented, conceptualised and tools created to implement them.

### Benefits of Modular Design

**Enhanced Maintainability:** The modular structure makes it easier to update and maintain different parts of the software independently.

**Improved Scalability:** Modules can be scaled independently, allowing for more efficient resource utilization and system growth.

**Increased Flexibility:** Modular designs allow for easier integration of new technologies and quick adaptation to changing requirements.

**Simplified Debugging and Testing:** Isolated modules simplify the process of identifying and fixing issues.

#### Trade-offs in Adopting Modular Design

**Complexity in Integration:** Ensuring seamless interaction between different modules can be complex.

**Overhead in Communication:** Communication between modules, especially in distributed systems, can introduce latency and complexity.

**Design Challenges:** Designing a system with well-defined, independent modules requires careful planning and expertise.

In conclusion, the transition to modular design has been a crucial evolution in software architecture, addressing the challenges of increasing complexity and maintenance demands. This shift has laid the groundwork for more robust, scalable, and maintainable software architectures and represents a significant milestone in the maturation of software development practices.

## Service-Oriented Architecture (SOA)

#### Timeframe

SOA emerged in the late 1990s and early 2000s, a period that coincided with the rapid expansion and democratization of the internet.

#### Key Contributors

The concept of “Service-Oriented Architecture” was popularized by Gartner in a 1996 report. During the early 2000s, IBM and Microsoft were notable advocates, integrating SOA principles into their products and services.

#### What is Service-Oriented Architecture (SOA)?

SOA is a software design paradigm that focuses on service reuse and interoperability, essential in a networked world boosted by the internet’s growth. It represents an architectural pattern where applications are built to provide services to other applications across a network, utilizing communication protocols. SOA’s hallmarks include reusability, loose coupling, interoperability, and discoverability, facilitating diverse services to interlink and address various business needs.

#### Link with Technological Developments

SOA’s ascendancy is linked to business needs, technological advancements, and the widespread adoption of the internet. As businesses sought agility and flexibility to

quickly respond to market changes, SOA provided an apt framework. The internet's democratization played a pivotal role, offering a global platform for interconnected services and applications. This technological backdrop made SOA a natural fit, enabling seamless integration of diverse systems and promoting a collaborative IT environment in an increasingly internet-centric world.

#### Benefits

**Enhanced Business Agility:** Leveraging the internet's reach, SOA enables swift adaptation to new opportunities and challenges by reconfiguring services.

**Cost-Effectiveness:** By reusing services, SOA cuts down repetitive software development costs.

**Simplified Maintenance and Scalability:** SOA's modular nature eases maintenance and enhances scalability, aligning with the dynamic nature of the internet.

#### Trade-offs

**Complexity in Design and Governance:** The challenge lies in designing and managing SOA frameworks, requiring robust systems.

**Legacy System Integration:** Incorporating older systems into an SOA framework, especially in an internet-driven environment, often requires substantial changes.

**Cultural Shifts:** Adopting SOA demands a shift from traditional, isolated approaches to collaborative, networked mindsets, mirroring the interconnectedness fostered by the internet.

## The Rise of Microservices

#### Timeframe

Gaining momentum in the early 2010s.

#### Key Contributor

Dr. Peter Rogers is credited with first using the term *"micro web services"* during a cloud computing conference in 2005. The term "Microservices" gained widespread attention in 2011 at a workshop for software architects, where many were experimenting with this style of architecture.

#### What is Microservices Architecture?

Microservices architecture represents an evolutionary development in software architecture, building on SOA principles but introducing finer granularity in service decomposition. It involves breaking down applications into small, independently

deployable services, each running unique processes and communicating typically via HTTP-based APIs.

#### Link with Cloud Computing, DevOps, and Containerization

The ascent of microservices is closely tied to advancements in cloud computing, DevOps, and containerization. Cloud computing offers scalable infrastructure ideal for deploying microservices, while DevOps practices align with the philosophy of rapid, reliable delivery of complex applications. Containerization technologies like Docker and Kubernetes provide consistent environments for development and simplify service management.

#### Benefits

**Improved Scalability:** Microservices allow for scaling individual components of an application rather than the entire application, facilitating efficient resource usage.

**Enhanced Flexibility and Agility:** The modular nature of microservices enables faster development cycles and the ability to update or add new features without overhauling the entire system.

**Resilience and Isolation of Faults:** Failures in one service do not necessarily cause system-wide failures, enhancing overall resilience. This isolation simplifies troubleshooting and recovery.

**Technological Diversity:** Teams can use the best technology stack for each service, leading to optimized performance and ease of integration with various tools and technologies.

**Continuous Deployment and Delivery:** Aligning well with DevOps practices, microservices facilitate continuous integration and continuous deployment, allowing for frequent and reliable updates.

**Improved Scalability with Cloud Compatibility:** Microservices are particularly compatible with cloud environments, benefiting from cloud infrastructure's flexibility and scalability.

#### Trade-offs

**Complexity in Management and Coordination:** Managing multiple services can be more complex than managing a monolithic application. This includes challenges in service discovery, load balancing, and inter-service communication.

**Network Latency and Communication Overhead:** As services communicate over the network, there can be latency issues, especially if not properly managed.

**Data Management Complexity:** Handling data consistency across different services can be challenging, especially when each service has its database.

**Increased Resource Requirements:** While each service might be small, the cumulative resource requirement for running many services can be significant.

**Testing Complexity:** Testing a microservices-based application can be more complex than testing a monolithic application due to the interactions between services.

**Overhead in Deployment and Operations:** The need for automated tools and processes to manage deployment, monitoring, and logging across numerous services can add operational overhead.

**Security Challenges:** Ensuring security across multiple services and their interactions adds complexity, requiring robust security strategies and tools.

**Cultural and Organizational Adjustments:** Adopting microservices often requires changes in team structure and communication, moving towards a more decentralized approach.

## Serverless Architectures — The Next Frontier

#### Timeframe

Began to surface prominently around 2014, following the launch of AWS Lambda.

#### Key Contributor

The concept of “serverless” computing was popularized by Amazon Web Services with the launch of AWS Lambda in 2014, a platform which executes code in response to events without requiring the user to manage the underlying compute resources.

#### What is Serverless Architecture?

Serverless architecture is a transformation in application building, deployment, and management, abstracting away servers and infrastructure management. In this model, developers focus on writing and deploying code without managing underlying hardware. Core components include Function as a Service (FaaS) and Backend as a Service (BaaS).

### Serverless as an Extension of Microservices

Serverless can be seen as an extension of microservices, reducing service granularity to individual functions. It's driven by the need for scalability and cost-efficiency, allowing automatic scaling and ensuring payment only for used resources.

### Benefits

**Cost Efficiency:** Serverless computing typically operates on a pay-as-you-go model, meaning businesses only pay for the resources they use. This can lead to significant cost savings compared to traditional cloud hosting models.

**Scalability:** Serverless architectures can automatically scale up or down based on demand, eliminating the need for manual scaling. This makes it ideal for applications with variable workloads.

**Simplified Operations:** The abstraction of servers and infrastructure management reduces operational complexity. Developers can focus on writing code without worrying about server maintenance and configuration.

**Faster Time-to-Market:** The ease of deploying applications in a serverless environment leads to quicker development cycles, facilitating faster time-to-market for new features and updates.

**Enhanced Productivity:** By offloading infrastructure concerns, developers can focus more on the business logic and user experience, enhancing overall productivity.

**Event-driven and Modern Workflow Compatibility:** Serverless architectures are inherently event-driven, making them well-suited for modern application workflows like IoT applications, real-time file processing, and more.

### Trade-offs

**Vendor Lock-in:** Most serverless services are provided by specific cloud providers. This can lead to vendor lock-in, restricting flexibility and potentially complicating future migrations.

**Cold Start Issue:** Serverless functions may experience a 'cold start' — a delay during the initial execution when the function is not already running, affecting performance.

**Limited Control:** The abstraction that simplifies operations also means less control over the underlying infrastructure, which can be a disadvantage for certain specific requirements.

**Debugging and Monitoring Challenges:** Traditional debugging and monitoring tools are often less effective in a serverless environment due to its distributed and ephemeral nature.

**Security Concerns:** Security in serverless architectures can be complex, as it requires a different approach to traditional server-based environments. Ensuring secure function execution and data transfer is crucial.

**Time and Memory Constraints:** Serverless functions typically have time and memory limits, which might not be suitable for long-running processes or applications requiring significant computational resources.

**Complexity in Large-Scale Applications:** While serverless is excellent for microservices and small, independent functions, managing a large-scale application with numerous functions can become complex.

## Tools adapt to architectures that adapt to the underlying network

It's fascinating to observe how the evolution of software architecture is characterized by a trend towards decentralization, where the overarching theme is 'decoupling.' This shift is a response to the advancements in network capabilities and hardware. As networks become more capable and efficient, they demand enhanced interoperability and interconnectedness. In this context, microservices and serverless architectures represent the latest step in addressing these evolving constraints.

Concurrently, as software architecture has adapted to these contextual constraints, tools have evolved in tandem to support these architectural changes. The rise of DevOps, along with tools such as Docker and Kubernetes, aligns perfectly with the needs of microservices architecture. Similarly, the advent of Agile methodology coincided with the internet revolutionizing business and software development. This shift was essential to meet the new demands of a rapidly changing market and user behavior, necessitating reduced time-to-market for software products.

## The Dangers of Blindly Following Architectural Trends

In the rapidly evolving field of software architecture, it's tempting for architects and developers to gravitate towards the latest trends and practices. However, this inclination, if not tempered with critical evaluation, can lead to significant pitfalls. While architectural styles such as microservices, serverless computing, and others have evolved to address specific constraints and requirements of modern software development, the blind adoption of these trends without a thorough understanding of their implications can be detrimental.

### Context-Driven Decision Making

Every architectural decision should be driven by the specific context and requirements of the project at hand. For example, while microservices offer scalability and flexibility, they also introduce complexity in deployment and management. Similarly, serverless architectures, despite their cost-efficiency and scalability, might not be suitable for all types of workloads, especially those requiring long-running processes. The key is to understand that there is no one-size-fits-all solution in software architecture. What works for one project or organization might not be appropriate for another.

### The Pitfalls of Trend-Driven Architecture

The trend-driven approach to software architecture often overlooks critical factors such as organizational readiness, team skill sets, and the actual needs of the business or application. This can lead to several issues:

**Overengineering:** Implementing a complex architectural style like microservices for a simple application can lead to unnecessary complexity and resource drain.

**Mismatch with Business Needs:** Choosing an architecture style that doesn't align with business goals or operational capabilities can hinder rather than help the organization.

**Skill Gaps:** Adopting a new architectural trend without having the necessary expertise in the team can lead to poor implementation and maintenance challenges.

**Increased Costs and Maintenance Overheads:** Without proper consideration, the adoption of a new architecture can lead to increased costs, both in terms of development and ongoing maintenance.

### Balancing Innovation and Pragmatism

While it's important to stay abreast of new trends and technologies, architects must balance innovation with pragmatism. This involves a careful assessment of the

benefits and drawbacks of a particular architectural style in the context of the specific requirements, capabilities, and constraints of the project and organization. It also entails a commitment to continuous learning and adaptation, ensuring that decisions are made based on a solid understanding of both the current technological landscape and the specific needs of the business.

## Increasing Infrastructure Costs in Decentralized Architectures

As we delve deeper into the realm of decentralized architectures, like microservices and serverless computing, a critical aspect that emerges is the escalating infrastructure costs. This phenomenon is a direct consequence of the shift from monolithic to more fragmented, distributed systems.

### Why Infrastructure Costs are Rising in Decentralized Systems

**Greater Complexity in Deployment and Management:** Decentralized architectures typically involve a multitude of services, each potentially running on separate instances or containers. Managing such a distributed setup requires sophisticated orchestration and monitoring tools, which adds to the infrastructure overhead.

**Resource Redundancy:** In a microservices architecture, each service might need its own set of resources, including databases, caching, and networking capabilities. This redundancy can lead to increased usage of computational resources, thereby raising costs.

**Network Traffic and Data Transfer Costs:** The inter-service communication in decentralized systems often occurs over the network. As the volume of this inter-service traffic increases, so does the cost associated with data transfer, especially in cloud-based environments where network usage incurs charges.

**Scaling Costs:** While decentralized architectures offer better scalability, the cost of scaling numerous small services can be higher than scaling a single monolithic application. Each service may need to be scaled independently, requiring more compute and storage resources.

**Maintenance and Monitoring Tools:** The need for specialized tools to monitor and maintain a distributed system also contributes to higher costs. These tools are essential for ensuring system health and performance but come with their own set of licensing and operational expenses.

### Strategies to Mitigate Infrastructure Costs in Decentralized Systems

**Efficient Resource Utilization:** Leveraging containerization and orchestration tools like Kubernetes can optimize resource usage, ensuring that services use only what they need and scale down when not in use.

**Cost-effective Service Design:** Designing services to be lightweight and resource-efficient can significantly reduce costs. This includes optimizing the codebase and choosing the right set of tools and technologies that balance performance with cost.

**Intelligent Scaling Strategies:** Implementing auto-scaling policies that accurately reflect usage patterns can prevent over-provisioning of resources, thus reducing costs.

**Monitoring and Optimization:** Continuous monitoring of resource usage and performance can help identify and eliminate inefficiencies, leading to cost savings.

**Hybrid Architectural Approaches:** Sometimes, a hybrid approach that combines elements of both monolithic and microservices architectures can offer a more cost-effective solution. This approach allows for leveraging the benefits of microservices where it makes sense while maintaining simpler, cost-effective monolithic structures for other parts of the application. This is the actual rise of the “Modulith” approach.

As decentralized architectures continue to gain popularity, understanding and managing the associated infrastructure costs becomes increasingly important. By adopting strategic approaches to resource utilization, service design, and scaling, organizations can enjoy the benefits of decentralized systems while keeping infrastructure costs in check. This balance is crucial for achieving not just technological efficiency but also financial prudence in the modern era of software development.

## Conclusion

The comprehensive exploration of software architecture from monolithic structures to microservices and serverless paradigms underscores a dynamic and ever-evolving field. This evolution, closely mirroring advancements in technology and shifting business needs, highlights a continual movement towards decentralization and flexibility in software design. The journey from the unified simplicity of monolithic systems to the granular specificity of serverless computing illustrates a keen response to the growing complexity of applications and the pressing need for scalable, efficient solutions.

However, this progression is not without its challenges. As architectures have become more decentralized, there has been a corresponding rise in infrastructure costs and operational complexities. This necessitates a balanced approach, blending innovation with pragmatism. The adoption of any architectural style — be it microservices, serverless, or others — should be a deliberate choice, driven by the specific context and requirements of the project, rather than a reflexive following of prevailing trends.

Moreover, the shift in architectural paradigms also brings to light the significance of supportive tools and methodologies. The synergy between evolving architectures and advancements in tools like Docker, Kubernetes, and Agile practices reflects a cohesive maturation of the software development ecosystem. It is a testament to the industry's adaptability and responsiveness to the changing technological landscape.

In navigating this complex terrain, architects and developers must exercise discernment, aligning architectural choices with strategic business objectives and operational capabilities. Balancing innovation with a practical understanding of costs, benefits, and organizational readiness is crucial. As the field continues to advance, the focus should remain on creating architectures that are not only technologically robust but also pragmatically aligned with the unique needs and goals of each project.

In conclusion, the evolution of software architecture is a journey of adaptation and refinement. It is a reflection of the software industry's relentless pursuit of solutions that are more scalable, resilient, and aligned with the rapid pace of technological change. By understanding and thoughtfully applying these architectural paradigms, we can continue to forge software that is not only functionally superior but also strategically advantageous in an increasingly complex and interconnected digital world.

*I am the CTO and Head of an architectural unit in a digital company. I participate in the development of technological strategy, design solutions, and lead R&D projects.*

*Thank you for reading! If you enjoyed this article, please feel free to* 👏 *and help others find it. Please do not hesitate to share your thoughts in the comments section below.*

## Scub Lab

*Thank you for being a part of our community! Before you go:*

- *Be sure to **clap** and **follow** the writer!* 👏
- *You can find even more content at **lab.scub.net*** 🚀
- *Sign up for our **free weekly newsletter**.* 🗞️
- *Follow us on **Twitter**(X), **LinkedIn**, and our **site web**.*

Programming | Technology | Software Development | Software Engineering | Software Architecture

Edit profile

## Written by Pier-Jean Malandrino

3.5K Followers · Editor for Scub-Lab

CTO & Head of Architecture at a digital firm, I drive technological strategy, design innovative solutions, & lead R&D projects.

### More from Pier-Jean Malandrino and Scub-Lab

Pier-Jean Malandrino in Scub-Lab

## Data Lake vs. Data Warehouse

Understanding Their Core Differences, Benefits, and Trade-offs

4 min read · Mar 11, 2024

20 1


Boris Bodin in Scub-Lab

## What is it ? — Gherkin for test naming

Learn how using Gherkin syntax for test naming improves test readability and maintenance in SpringBoot/Angular applications.

· 9 min read · Nov 13, 2023

5


Boris Bodin in Scub-Lab

## Focus on: Factory Pattern

Demystifying the Builder Pattern: A practical guide for software architects and developers.

· 11 min read · Dec 11, 2023

6

 Pier-Jean Malandrino in Scub-Lab

## Architecture : The cheat sheet

This paper presents a concise summary of various software architecture patterns, methods and models.

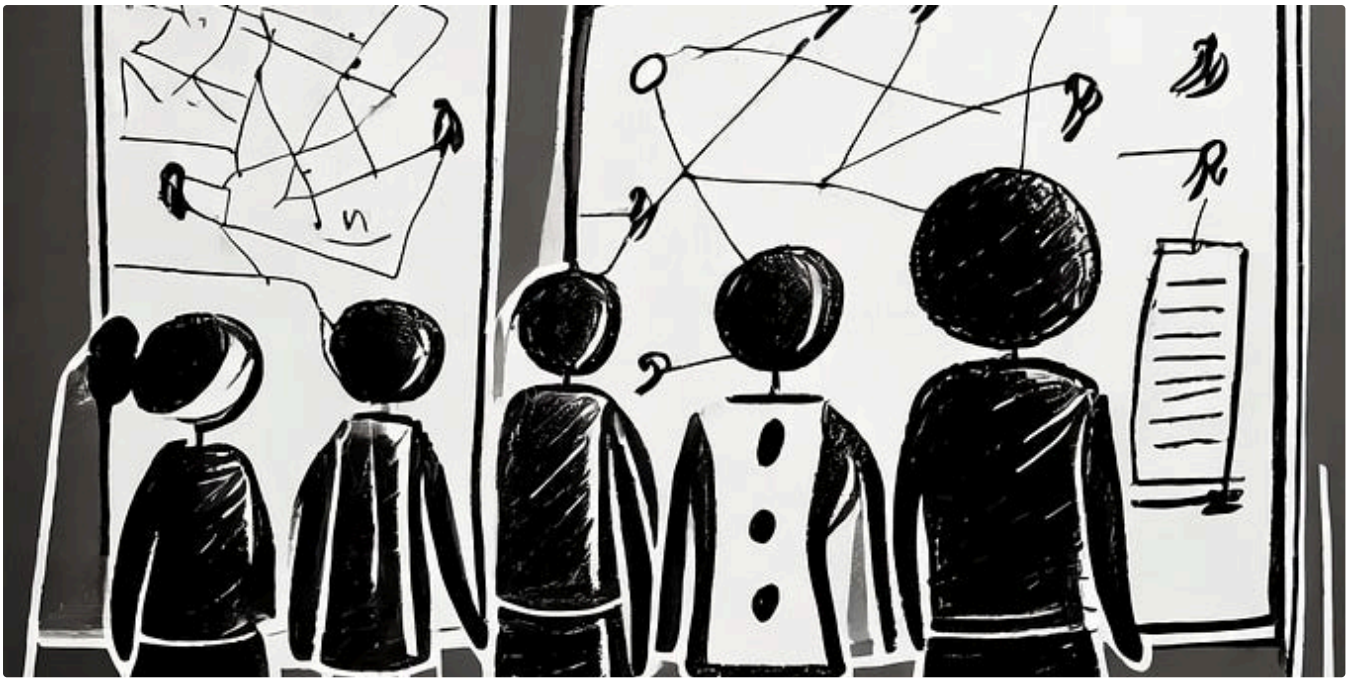
5 min read · Dec 25, 2023

1.92K 6

See all from Pier-Jean Malandrino

See all from Scub-Lab

## Recommended from Medium



Daniel Moldovan

## How to be a staff engineer. Or team lead. Or principal engineer. Or manager. Or whatever ...

Many wonder what does it take to become a senior software engineer. Or staff engineer. Or principal software engineer. Or manager. Or ...

15 min read · Jan 11, 2024

200 1

Andres Felipe Rincon

## Architectural Knockout: Building Your Design System

After the definition of the Architecture Design System (ADS) and its Concepts, it is time to present how to build an ADS using BAU tools...

9 min read · Jan 25, 2024

4

### Lists

**General Coding Knowledge**

20 stories · 1210 saves

**Stories to Help You Grow as a Software Developer**

19 stories · 1053 saves

**Coding & Development**

11 stories · 610 saves

**ChatGPT**

21 stories · 630 saves

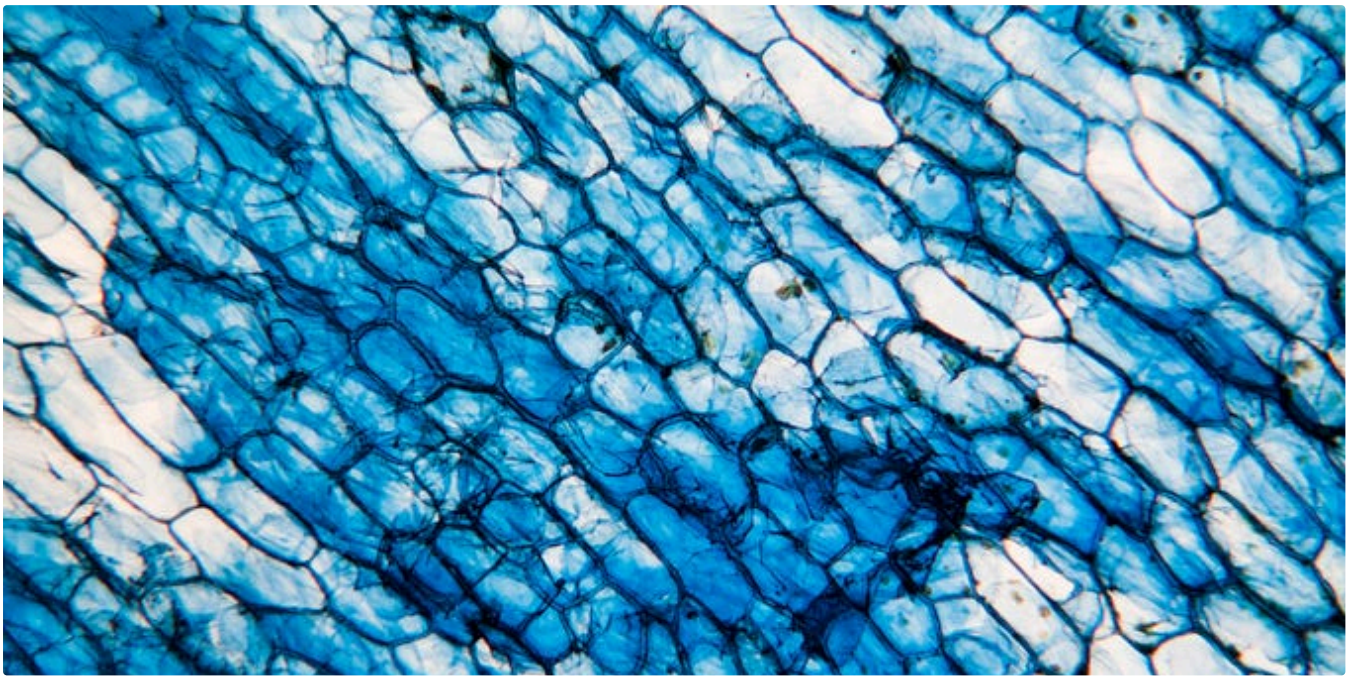
Aashish Peepra

## How to design clean API interfaces

This is going to be a tough read. This contains too much code to read and no pictures. It's not for people with faint hearts. So read if...

7 min read · Mar 3, 2024

525 7



chubernetes

## Evolution of Monolithic Systems

The Entropy of Software in Tech Startups

13 min read · May 5, 2024

152 1

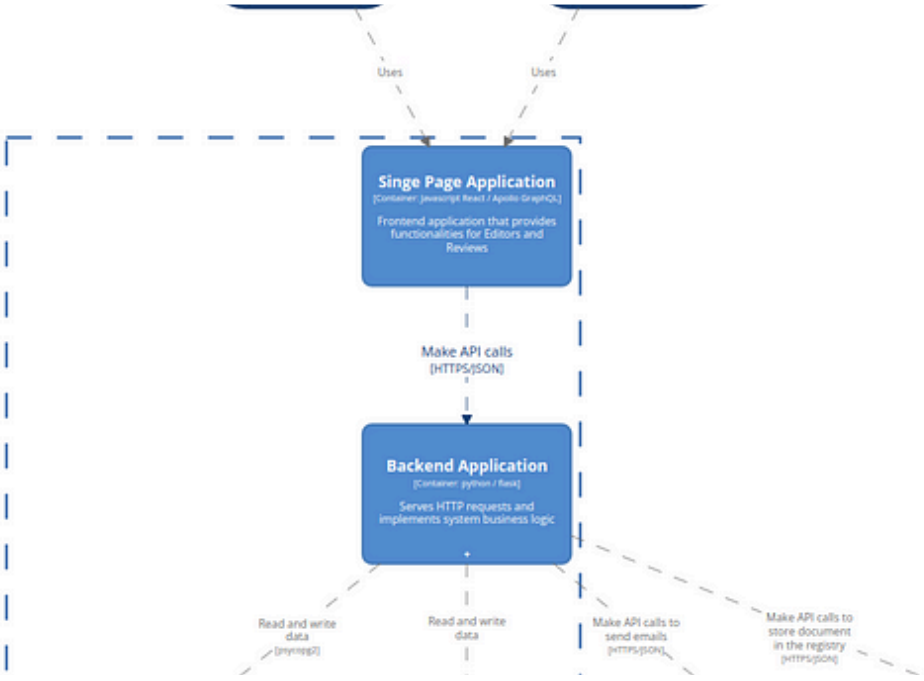

Mario Bittencourt in SSENSE-TECH

# Exploring Advanced Error Handling Patterns with Event-Driven Architecture—Part I

An event-driven architecture (EDA) brings changes to the way we approach error handling. When using the more commonly adopted synchronous...

7 min read · Apr 5, 2024

340 1




Jarek Orzel in Towards Dev

## How to visualize your system architecture using the C4 model?

As a software developer or system architect you often have a task to visualize your existing or potential application architecture for...

5 min read · Jan 5, 2024

 25  1  

See more recommendations