Monolithic vs. Microservices in the Cloud :- Review on Performance, Scalability, and Fault Tolerance

Ashutosh Dongre (22bcs020), Atharva Attarde (22bcs021), Aniket Shelke (22bcs010) and Abhijeet Alande (22bcs007)

Indian Institute of Information Technology, Dharwad

Abstract—Cloud computing has really changed how we make and use applications. The way we design these applications (architecture) is very important for how well they work, how much they can grow, and if they can keep working when something breaks. This review looks at two main types of cloud application designs:- monolithic and microservices. Monolithic designs are like everything in one big code base. They are easy to build and use at first. But, it becomes hard to make them bigger or to stop problems in one part from affecting everything else. Microservices designs are made of many small, separate parts that work together. They are better at growing and staying strong when problems occur. However, they can be more complex to manage and run. This review is about the good and bad things of both designs in cloud setups. We will see how well they can handle changes in work, get back to normal after failures, and use computer resources efficiently. We want to understand the good points and bad points of each design. This will help developers decide which design is best for their cloud applications.

Keywords—*Monolithic Architecture, Microservices Architecture, Cloud Computing, Scalability, Resilience, Fault Tolerance, Literature Review*

1. Introduction

Cloud computing is a very important new technology. It has greatly changed how information technology works and how organizations and people use computers and data storage. In the past, companies used their own servers and computer systems. Now, cloud computing provides access to shared computer resources like networks, servers, storage, software, and services. These are available online when needed. This is a major shift from older IT methods. Cloud computing allows organizations to be more scalable, agile, and cost-effective. By simplifying computer system management, cloud computing allows companies to concentrate on their main business activities, encouraging innovation and quick responses in the global market. Santos et al. [8] says "Cloud computing is a revolutionary technology; it has changed how companies and users interact with computing and data storage. Instead of relying on local servers and physical systems, cloud computing provides access to computer resources over the internet. This new approach offers benefits like scalability, flexibility, and reduced costs, making it essential for organizations worldwide." This significant change has led to cloud adoption in many sectors, from finance and manufacturing to education and healthcare. This marks a big change in how technology supports modern operations and service delivery. Driven by business needs like saving costs, increasing speed, faster product launches, and global expansion using cloud has become a key strategy for businesses in all industries and cloud computing industry to reach 1 trillion valuation mark [7].

However, to realize the full potential of cloud computing, the architecture used to implement it is crucial. Simply moving to the cloud does not guarantee success. To truly achieve benefits like scalability, resilience, cost-efficiency, and agility, choosing and implementing well-designed cloud architectures is essential. In today's complex digital world, cloud setups are not simple. They often combine different cloud service types (IaaS, PaaS, SaaS), deployment methods (public, private, hybrid, community), and many interconnected services and microservices. Industry experts emphasize that the architecture is the foundation for successful cloud strategies. Poor architectural decisions can lead to problems such as slow performance, security weaknesses, vendor lock-in, and ultimately, not achieving the expected return on investment. Furthermore, cloud computing presents technical challenges, including ensuring strong security and compliance [4] , managing complex distributed systems integrating with existing older systems and finding skilled cloud professionals. Scalability and resilience are consistently ranked as top priorities for cloud applications .These are fundamental architectural qualities that must be designed into cloud systems from the beginning. Therefore, understanding and prioritizing good cloud architecture is not just a technical detail, but a key strategic need for organizations wanting to succeed in the cloud era. As organizations depend more on cloud for critical operations and data-intensive applications, the importance of cloud architecture becomes increasingly clear. This requires careful planning, expert knowledge, and a deep understanding of how business needs and technical capabilities interact.

1.1. How This Literature Review is Organized

To make this literature review clear and easy to follow, we will organize it in the following way:

1. **Introduction:** We will start by explaining why cloud computing and choosing the right architecture are important. We will also introduce the two architectures we are going to compare: monolithic and microservices.
2. **Monolithic Architecture in the Cloud:** In this part, we will look at monolithic applications when they are used in the cloud. We will study:
   - What are the features of monolithic architecture.
   - How they are usually scaled (mostly by making them bigger - vertical scaling).
   - How reliable they are (resilience).
   - How well they perform in the cloud.
3. **Microservice Architecture in the Cloud:** Next, we will study microservice architectures in the cloud. We will explore:
   - The basic ideas of microservices.
   - How they are usually scaled (mostly by adding more services - horizontal scaling).
   - How they handle failures and stay reliable (resilience mechanisms like fault isolation and separate data).
   - How well they perform in the cloud.
4. **Comparing Scalability:-** In this section, we will directly compare how well monolithic and microservice architectures scale in the cloud. We will look at:
   - Horizontal vs. vertical scaling.
   - Which one is more cost-effective.
   - Which one works better for different types of workloads.
   - We will use studies that tested performance under different loads, like the experiments done by Blinowski and others [2].
5. **Comparing Resilience:-** Here, we will compare how reliable both architectures are in the cloud. We will focus on:
   - How well they handle errors (fault tolerance).
   - How they recover from problems (recovery strategies).
   - How their distributed nature affects system stability.
6. **What Affects the Choice of Architecture:** In this part, we will analyze different things that affect whether you should choose monolithic or microservices. These things include:
   - How big the application is.
   - How the team is organized.
   - What the business needs are.

- What the cloud provider offers.
- We will see how these factors influence the best architectural choice for scalability and resilience in cloud environments.

7. **Conclusion:** Finally, we will summarize the main points of the literature review. We will highlight the good and bad points of monolithic and microservice architectures in the cloud. We will also point out areas where more research is needed in the future.

1.2. What This Literature Review is About

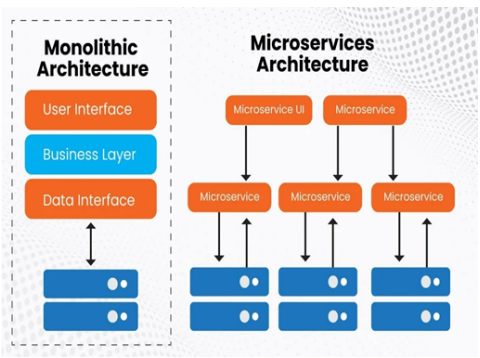
This literature review will mainly compare Monolithic and Microservice architectures. We will look at these two architectures specifically when they are used in cloud environments. More and more companies are using cloud computing for their applications. So, it's important to understand which type of architecture works best in the cloud. As Blinowski and others [2] have said, "Microservices are getting a lot of attention from researchers and companies, especially for cloud setups." So, in this review, we will study how well these two architectures perform and what their features are when used in the cloud.

We will mainly focus on two important things: Scalability and Resilience.

Scalability means how well a system can handle more work when you add more resources. In cloud environments, the workload can change a lot, and many users might access the system at different times. Microservices are often praised for their "horizontal scalability". This means you can easily add more services to handle more load. Monolithic applications are usually scaled "vertically", which means you make the existing system bigger and more powerful. We want to understand which way of scaling works better and is more cost-effective in the cloud.

Resilience means how well a system can keep working and recover from problems or failures. Cloud computing systems are often spread out and can have problems. As noted in [5], microservice applications are loosely connected, which makes them more fault-tolerant. If one part fails, the whole system may still work. In this review, we will look at how resilient each architecture is in the cloud. We will consider things like how well they isolate problems, how they use backup systems (redundancy), and how they recover when things go wrong.

This literature review is important because we need to make good decisions about architecture in the cloud era. While many people think microservices are the best modern approach, Blinowski and others [2] point out that "for systems that don't have many users and can be scaled up by making them bigger, we haven't really studied if moving to microservices is better. And the research we have is not clear". This review wants to answer the main question: "Which architecture – monolithic or microservices – is better for scalability and resilience in cloud environments? And when are these advantages most obvious?" By looking at existing research, this review aims to give a clearer idea of the pros and cons and best practices for choosing between monolithic and microservice architectures for applications in the cloud.



1.3. Monolithic Architecture Vs Microservices Architecture

Monolithic Architecture:-

What it is:- A monolithic architecture is like building an application as one big piece. All parts of the software are put together into a single unit. This is the traditional way of designing software.

What it's like:

- **One Big Code:** All the software code is in one place.
- **Shared Database:** All parts of the application use the same database to store information.
- **Parts Talk Inside:** The different parts of the application talk to each other within the same system, meaning they are closely connected.

Good and Bad Points:

- **Good:**
  - Easier to build and get started, especially for smaller applications.
  - Simpler to deploy or put online.
- **Bad:**
  - Hard to grow and handle when the application becomes bigger and more complex.
  - Difficult to maintain and update as it gets larger.

Examples: Older types of business applications, like early online shopping websites and content management systems, often used monolithic architecture.

Microservices Architecture:

What it is: Microservices architecture is a way of building an application as a collection of small, independent services. These services are loosely connected and can be developed, deployed, and scaled separately.

What it's like:

- **Separate Parts for Different Jobs:** Each service is designed to do one specific business task.
- **Own Data Storage:** Each service manages its own database. This makes them more independent.
- **Different Data Tech Possible:** Services can use different types of databases depending on what they need. This is called "Polyglot Persistence".
- **Talking Through APIs:** Services communicate with each other using simple and standard methods like HTTP/REST or messaging systems. These are called APIs.

Important Ideas for Microservices:

- **12-Factor App Methodology:** These are good rules for building applications that are scalable and easy to maintain. They focus on things like keeping settings separate from code, making applications portable, and using continuous deployment [12].
- **Cloud-Native Principles:** These are ideas for designing applications to work well in cloud environments. They emphasize scalability, resilience (being able to recover from problems), and easy management [10].

Examples of Companies Using Microservices:

- **Netflix:** Netflix changed from a monolithic system to microservices to handle more users and be more reliable.
- **Amazon:** Amazon uses microservices to be more flexible in development and manage their huge product catalog.

2. Scalability in Cloud Architectures

2.1. Vertical Scaling of Monolithic Architectures

What it is: Vertical scaling, also called "scaling up," is like making your existing computer server stronger. You add more power to it, like more CPU , more memory (RAM), or more storage (hard disk). For

monolithic applications, this means making the single server where your application is running more powerful.

Good Things (Advantages):

- **Simple:** It's easy to upgrade one server. You don't need to do complicated setup.
- **Data is Consistent:** Because everything is on one server, it's easier to keep data consistent and avoid problems with data syncing.
- **Cheaper for Medium Size:** For applications that are not too big, vertical scaling can be cheaper than horizontal scaling.

Bad Things (Limitations):

- **Limited by Hardware:** You can only make a single server so powerful. There's a limit to how much you can upgrade one machine.
- **Single Point of Failure:** If this one server fails, the whole application stops working. This is a big risk.
- **Not Flexible Enough:** Vertical scaling might not be quick enough to handle sudden increases in users or demand. It's not very good at adapting to quick changes.

2.2. Horizontal Scaling of Monolithic Architectures

What it is: Horizontal scaling, also called "scaling out," is like adding more computer servers to share the work. You run multiple copies of your application on different servers and distribute the workload among them. For monolithic applications, this means running exact copies of the entire application behind a load balancer (which directs traffic to different servers).

Challenges:

- **Scaling Everything:** Even if only one part of your application is getting very busy, you have to scale the entire application. This is not efficient.
- **Wasting Resources:** You might end up using too many resources for parts of the application that are not busy, just to handle the busy parts.
- **Complicated to Set Up:** Managing multiple copies of the application is more complex. It's harder to keep data consistent and manage user sessions across different servers.

Worse Compared to Microservices (Limited Benefits Compared to Microservices):

- **Microservices Scale Better:** Microservices allow you to scale only the specific parts of the application that need more resources. This is much more efficient.
- **Microservices are More Reliable:** If one microservice has a problem, it usually doesn't affect other microservices. This makes the whole system more reliable.

2.3. Horizontal Scaling of Microservices

Microservices are designed to scale out easily. This means you can add more copies of individual services to handle more work. This allows for good resource management and flexibility, so the system can handle different amounts of load efficiently. Tools like Kubernetes and Docker are very important for managing this scaling. They automatically handle deploying, scaling, and load balancing microservices that are packaged in containers. These platforms make sure that when demand increases, each microservice can scale out by adding more copies, keeping performance good.

2.4. Vertical Scaling of Microservices

Vertical Scaling in Microservices: While microservices are mostly scaled horizontally, sometimes vertical scaling (making a single service instance more powerful) can be useful. For example, services that keep track of state (data) or have specific resource problems might get better performance by vertical scaling. However, this is not as common because microservices are designed to be distributed, and scaling out is usually preferred over scaling up.

2.5. Scalability Problems and Things to Consider for Microservices

Communication Overhead: Because microservices are made of many separate parts, they need to communicate with each other. This communication can slow things down and create bottlenecks in the network, which can affect the overall system speed. Using good communication methods and reducing the number of times services need to communicate directly (synchronously) is important to reduce these problems.

Complexity of Management: Managing many microservices is complex. It requires good management platforms, detailed monitoring, and advanced deployment processes. As the number of microservices grows, the complexity also increases. This means you need advanced tools and practices to make sure everything runs smoothly.

Data Management Problems: Making sure data is consistent across different services can be difficult. Using "eventual consistency" models (where data becomes consistent over time, not instantly) and having good ways to synchronize data are important to keep the system working correctly.

[6]:- This research paper talks about a smart auto-scaler that helps to use resources efficiently in microservice applications. It helps to improve how resources are allocated by monitoring each service individially .

3. Resilience and Fault Tolerance in Cloud Architectures

3.1. Resilience Challenges in Monolithic Architectures

Inherent Vulnerabilities: Monolithic architectures often present single points of failure, where a malfunction in one component can cascade, affecting the entire system. This tight coupling makes fault isolation challenging, leading to potential widespread outages.

Deployment and Update Challenges: Updating monolithic applications can necessitate redeploying the entire system, increasing downtime risks and expanding the "blast radius" of potential failures.

3.2. Resilience Strategies for Monolithic Architectures

Redundancy and Load Balancing: Implementing redundancy and load balancing can enhance availability. Hosting monoliths across multiple availability zones in the cloud can improve reliability by ensuring that the application remains operational if one or more data centers go down.

Disaster Recovery Planning: Establishing robust disaster recovery protocols is crucial. This includes regular backups, failover mechanisms, and comprehensive incident response strategies to mitigate downtime during unforeseen events.

3.3. How Reliable Microservices Are in Cloud Environments

Microservices have become a popular way to build modern applications that can grow easily, are reliable, and work well, especially in cloud environments. One of the main reasons people like them is that they are more reliable than older, monolithic

applications. This reliability comes from the basic design principles of microservices, which focus on keeping problems separate, being decentralized, and letting each service work on its own. In cloud environments, where things can change quickly and failures are expected, these features are very important for keeping applications running and making sure users have a good experience.

Keeping Problems Separate and Being Decentralized Makes Microservices Reliable

A key idea in microservices architecture, and the main reason for its reliability, is fault isolation. As noted in [5], "if one microservice fails, only that service will stop working in the whole system. The other microservices will continue to work, so the system can tolerate failures." This is very different from monolithic applications, where, as the paper notes, "in a monolithic system, if one service fails, everything stops working." This ability to isolate problems is a big advantage, because it stops small failures from causing big, system-wide crashes.

Microservices achieve fault isolation in a few important ways:

- **Independent Processes and Deployments:** Each microservice runs as a separate process, often inside a container. And each service is deployed or put online independently. The principle of "Independently Deployable" [5] emphasizes that "if we need to change or update a microservice, we can deploy it on its own, without affecting other microservices and without stopping the whole application." This separation at the process and deployment level means that if one service has a problem, it is less likely to directly affect other services. If a microservice has an error or crashes, because they are in containers and deployed independently, it's easy to quickly restart or replace them without stopping other parts of the system.
- **Loosely Connected and Clear Interfaces:** Microservices are designed to be loosely connected. They mostly talk to each other through well-defined APIs, often using simple methods like HTTP REST, as mentioned in the principle of "Smart endpoints and dumb pipes" [5]. This loose connection reduces how much services depend on each other. While services need to communicate to do bigger business tasks, using clear interfaces and keeping direct dependencies to a minimum limits how failures can spread. If a microservice becomes unavailable, other services that need to talk to it can be designed to handle this problem gracefully. They might use techniques fallback plans, instead of crashing because of the failure.
- **Decentralized Data Management:** The principle of "Decentralized Data Management" [5] also helps with fault isolation. As the paper explains, "each microservice must manage its own data. This means that when designing a microservice, it must connect to its own database or storage." This stops a single point of failure in the data storage from affecting the whole application. If the database for one microservice has issues, it mainly affects just that service. Other microservices with their own separate databases can continue to work. This decentralized data approach also limits data problems or errors from spreading across the system if there is a failure in one area.

Decentralization is another important factor that makes microservices reliable. The paper [5] talks about "Decentralized Governance" and "Decentralize All the Things". This means that microservices and the teams that manage them are independent. This decentralization is not just about management; it's also part of the technical design. There isn't one central part that controls everything. If that central part fails, it could crash the whole system. Because microservices are spread out, the system as a whole is stronger against local problems. Teams can make their own decisions about which technologies to use, how to scale, and how to deploy each microservice. This allows for more flexibility and reliability that is specifically designed for each service's needs.

Because of fault isolation and decentralization together, microservices can achieve graceful degradation. Graceful degradation means that if some parts of the system fail, the system can still keep working, although maybe with fewer features or in a limited way. In a microservices setup, if a less important microservice stops working, the main parts of the application can often continue to function. For example, imagine an online store built with microservices. If the service that recommends products fails, users might not see personalized suggestions. But they can still browse products, put items in their shopping cart, and buy things because the product catalog, shopping cart, and order processing services are working independently. This ability to keep working even when some parts fail ensures that users can still use the application, even if some features are temporarily unavailable. This reduces disruption and keeps the business running.

Netflix is a well-known example of resilience engineering. They intentionally cause failures in their live system (using tools like Chaos Monkey) to find weaknesses and make their microservices platform stronger.

3.4. Resilience Patterns and How to Make Microservices More Reliable

Microservices are naturally designed to be reliable. But to make them even better at handling problems, we use special techniques called resilience patterns. These patterns help to reduce failures, stop problems from spreading, and ensure the system is always available in cloud environments.

Important Resilience Patterns in Microservices

- **Circuit Breaker Pattern:** Imagine a circuit breaker in your house that trips when there's too much electricity. The circuit breaker pattern in microservices does something similar. It stops requests going to a service that is failing. It detects problems and stops sending requests to the failing service until it recovers. This helps to prevent system-wide failures and gives the failing service time to get back to normal before it is used again.
  - *Example:* Netflix uses something called Hystrix, which is a well-known example of the circuit breaker pattern [1].
- **Retry Mechanisms and Fallbacks:**
  - *Retry Mechanisms:* If a request fails temporarily, retry mechanisms try sending the request again. This helps to handle short-term problems.
  - *Fallbacks:* If a service is unavailable, fallbacks provide alternative responses. Instead of just failing, the system can provide a default answer or use a backup plan. This makes the system more available even when there are temporary issues.
  - *Example:* If a payment service fails to process a payment, it might try to resend the transaction a few times before giving up and reporting an error.
- **Bulkhead Pattern:** It separates different parts of the system so that if one service fails, it doesn't affect other services. By giving separate resources to different parts, the bulkhead pattern makes sure that problems in one area don't crash the whole system.
  - *Example:* In an online store, separating the inventory system from the order processing system means that if there's a problem with the inventory, it won't stop people from placing orders.
- **Health Checks and Monitoring:** They help to find problems early and start automatic fixes. Monitoring tools constantly watch how services are performing and find potential problems before they affect users.
  - *Example:* Kubernetes, a tool for managing containers, automatically does health checks and restarts containers that are failing.

Service Mesh for Microservices Resilience Service mesh technologies, like Istio and Linkerd, make microservices more reliable by managing how services communicate with each other. They provide features like: Dynamic Traffic Routing, Observability, and Security Policies.

- **Istio:** A service mesh that helps control traffic, inject failures for testing (fault injection), and use circuit breakers for microservices [9].
- **Linkerd:** A service mesh that improves reliability by automatically retrying failed requests, setting timeouts, and using smart load balancing that considers latency (delay) [11].

Conclusion By using resilience patterns and service mesh technologies, microservices architectures can handle failures, recover quickly, and stay highly available in cloud environments. These techniques make sure that distributed applications remain strong even when things go wrong unexpectedly.

3.5. Resilience Challenges and Considerations for Microservices

Even though microservices have many good things, making them reliable is more complex. Because they are spread out, they have new challenges compared to monolithic applications.

Complexity of Making Microservices Reliable: It's much harder to build reliability into microservice systems than into monolithic systems. As noted in [3], "The way services interact with each other is complicated and difficult to fully understand". This difficulty comes from several things:

- **Spread Out Nature (Distributed Nature):** Microservices work across networks. Networks can be slow, get divided (partitions), and fail. These network issues are just a normal part of microservice systems.[3] mention that parts of the application "used to talk to each other directly inside the system, but now they are separated and need to use APIs to exchange information". Because they rely more on network communication, they must be good at handling network problems.
- **Services Depend on Each Other (Inter-Service Dependencies):** Services need each other to work correctly. This creates complicated chains of dependencies. If one service fails, it can cause problems in other services that depend on it. As highlighted in [3], "services need to trust other parts...and depend on them to achieve what they need to do."
- **Different Technologies (Diverse Technologies):** Microservices are often built using different technologies and programming languages. This makes it harder to ensure reliability consistently across the whole system. The SockShop example in [3] shows this, with services in NodeJS, Java, and Go.
- **Watching and Getting Alerts (Monitoring and Alerting):** To make microservices reliable, you need good systems to watch them and send alerts when something goes wrong in a spread-out environment. Huang et al. [3] look at Kubernetes probes, which are used for monitoring, and find that they are not always good at finding all types of failures.
- **Putting Systems Online (Deployment Strategies):** Good strategies for deploying or putting microservices online are important to reduce downtime and keep the system always available. Examples are rolling updates, blue/green deployments, and canary releases.
- **Network Divisions (Network Partitions):** Parts of the network can get cut off from each other. This means services can't communicate with each other. This is a basic problem in spread-out systems, and you need ways to handle data consistency and synchronization even when this happens (eventual consistency).
- **Data Not Matching Up (Distributed Data Inconsistencies):** Keeping data consistent across many services and databases becomes more difficult. Often, "eventual consistency" models are used, where data becomes consistent over time, but you need to carefully think about how to sync data and handle possible conflicts.
- **Complex Failure Modes:** The way services interact can lead to new and complicated problems that are hard to predict and fix [3] indirectly talk about this by testing different failure scenarios to see how the system behaves when things go wrong.
- **Temporary Problems (Transient Faults):** Short-term network problems, temporary lack of resources, and other short issues are more common in spread-out environments. Reliability mechanisms need to be able to handle these temporary problems smoothly without causing big disruptions.

4. Comparative Analysis: Scalability and Resilience Trade-offs

From a perspective of critical software engineering requirements of scalability and resilience, there is a trade-off while deciding between monolithic and microservices architecture. There is no architecture superior to another, but their strengths and weaknesses are evident in different contexts to satisfy different demands. It requires a deep understanding of these trade-offs before coming to a conclusion while choosing the optimal architecture, considering the specific needs and constraints of the application, the views of the development team and the environment in which the application is operated. In this section we focus on the detailed comparative analysis of monolithic and microservices architectures, focusing specifically on their scalability and resilience characteristics, drawing insights from the performance evaluation conducted by Blinowski et al[2], and established architectural principles.

4.1. Direct Comparison Table: Monolithic vs. Microservices for Scalability and Resilience

4.2. Trade-offs and Contextual Suitability

When Monolith Systems Might Be Better:

Microservices are good, but monolith systems are still a good choice in many situations. This is because we need to think about things like how well the system can grow, how reliable it is, and other important things like how fast we can build it and how easy it is to manage.

For normal applications that have typical amounts of work and are not too big, monolith systems can grow enough and be reliable enough. They don't need the extra complexity that comes with microservices. A study by Blinowski and others [2] says that for systems with "a few thousand users at the same time" that can be made bigger by using more powerful servers, it's not clear if moving to microservices is really better. The research on this is not clear. So, for most applications, a monolith system can grow big enough and handle enough users to meet the needs.

Also, if it's more important to build something quickly and simply, rather than having the most reliable and scalable system, then a monolith system is the way to go. This is because monolith systems are easier to build, set up, and test. This means you can get your product to market faster. For small teams or projects with short deadlines, the extra work of building and managing a microservices system might be too much compared to the benefits you get in terms of growth or reliability.

Monolith applications have some advantages when you can make them bigger by using more powerful servers (vertical scaling). Blinowski and others [2] said that "making a server more powerful is cheaper than using many servers" in the Azure cloud. If an application can work well just by using a stronger server, and if this server is not too expensive, then making a monolith system bigger this way is easier and possibly cheaper than changing to a complex microservices system.

Table 1. Comparison of Monolithic vs. Microservices Architectures for Scalability and Resilience

Feature/Aspect	Monolithic Architecture	Microservices Architecture
Scalability - Horizontal Scaling	To scale horizontally, the whole application is copied. Scaling happens for everything, even if only some parts are busy. This can waste resources because everything gets copied even if not needed. Managing and setting up large monoliths becomes hard.	Designed for horizontal scaling of separate parts. Each service can be scaled by itself based on how busy it is. This saves resources by only scaling what's needed. Special systems are required to manage the complexity of many scaled services.
Scalability - Vertical Scaling	This means making the server that runs the monolith stronger (more CPU, RAM, storage). It is easy to do at first. Simple to start with, but performance is limited by the most powerful server you can get. Vertical scaling can become very expensive for high performance.	Individual services can be made stronger by giving them more resources. While horizontal scaling is more common for microservices because they are distributed, vertical scaling can be used to improve a single service before scaling it horizontally.
Scalability - Elasticity	Less flexible. Scaling is usually "all or nothing", meaning you scale the whole monolith even if only a few parts are overloaded. Scaling up or down does not respond quickly to changes in demand in monoliths.	Very flexible and can change dynamically with workload. Services can automatically scale up or down based on current needs. This saves resources and costs. Special systems often provide automatic scaling for microservices.
Scalability - Resource Efficiency	Can waste resources, especially when scaled horizontally. Copying the whole monolith means some copies are not fully used or even idle. This wastes resources and increases costs in cloud environments where you pay for what you use.	Can be very resource-efficient when scaled. Scaling parts separately means resources are mainly used for busy services. Reduces waste by not copying parts that are not busy. This efficiency is important for saving money in dynamic clouds.
Scalability - Complexity	Scaling is simple at the beginning, especially vertical scaling. But, managing and setting up scaled copies of very large monoliths, especially for horizontal scaling, can become very complex over time. Releasing updates can take longer and be riskier as the monolith gets bigger.	Makes scaling infrastructure complex. Needs strong systems to manage services (like Kubernetes), ways to find services, load balancing, and advanced monitoring tools. Complexity moves from scaling the application itself to managing the infrastructure.
Resilience - Fault Isolation	Limited fault isolation because parts of the monolith are tightly connected. A problem in one part can affect the whole application's function or availability. Errors can easily spread within the single codebase.	Good fault isolation is a key advantage. Services are separate and loosely connected. A failure in one service is less likely to directly affect others or the whole application. Keeping problems contained improves the stability of the entire system.
Resilience - Handling Component Failures	When something fails, you often have to restart the entire monolith application. This can cause service downtime and longer recovery times, affecting availability. Fixing and recovering large monoliths can be complex and slow.	Failure of one service can be handled separately. Individual services can be restarted, redeployed, or switched to backup copies without impacting other parts. Automatic recovery and management systems help in faster recovery and better availability.
Resilience - Implementation Complexity	Simpler to add basic resilience features within a single codebase and application. Resilience logic is in one place and easier to understand in a monolith. But, complex resilience methods for distributed systems are not naturally suited for monoliths.	More complex to add resilience methods that work across different services. Requires using techniques like circuit breakers, bulkheads, retries, fallbacks etc. Needs skills in distributed systems and careful handling of network problems and delays.
Resilience - Monitoring and Management	Monitoring and management are initially in one place because the whole application runs as one unit. But as the monolith grows, monitoring and tracking problems and performance issues within the complex code can become very hard. Collecting and connecting logs can be difficult.	Monitoring and management are naturally distributed. Needs advanced tools and ways to track the health, performance, and connections of many services. Requires distributed logging, tracing, and alerting systems to see the whole system's behavior. Setting up monitoring infrastructure is more complex at first.
Resilience - Overall Robustness	Overall robustness depends a lot on how stable and fault-tolerant the single application copy is. There is one single point of failure at the monolith level. Robustness is achieved by writing good code, thorough testing, and having backup infrastructure at the server level.	Can achieve higher overall robustness because of built-in fault isolation, redundancy, and ability to handle failures in single services separately. Robustness depends on how well each service is made resilient and how effective the distributed resilience methods are. Requires careful design.

However, if you choose a monolith to handle growth and be reliable, you lose some ability to easily grow in many directions (horizontal scaling) and to keep problems in one part of the system from affecting other parts (fault isolation). Monoliths can be scaled horizontally, but it's not as easy or efficient as with microservices. Monoliths are also tightly connected. This means if one part fails, it can cause the whole application to fail. For example, if there's a memory problem in one part of a monolith, it can crash the entire application. Then you have to restart everything, even the parts that were working fine.

When Microservices Systems Are Better:

Microservices become necessary when you really need to grow a lot and be very reliable. This is especially true for applications that have very changing and unpredictable workloads. For example, e-commerce websites get huge traffic during sales, or social media apps have different numbers of users at different times of the day. Microservices are great in these situations because they can easily grow horizontally. You can increase the number of servers for only the parts of the application that are experiencing high demand. This uses resources efficiently and keeps the application responsive even when there are many users.

Microservices are used for applications that need to grow to a very large scale and adjust to changing demands. Like global streaming services or big online games that need to support millions of users and lots of data. Microservices give you the structure to divide the work across many servers and automatically increase or decrease server capacity based on how much traffic there is at any given moment.

Also, microservices are used for applications that must be very reliable and keep failures from spreading, especially for critical systems like banking systems or air traffic control. Because microservices are separated, if one service fails, it doesn't bring down the whole system. For example, if the payment service in a microservices e-commerce site fails, customers can still browse products. This limits the impact of the failure.

Finally, large, complex applications built by teams in different locations or different departments benefit from microservices. Microservices are designed to be modular, which is perfect for teams working independently. Teams can work on their own services, update them frequently, and not depend too much on other teams. This independence and speed are important for fast innovation and continuous updates in big software projects.

The main problem with microservices is that they are much more complex to build, set up, and run. It's harder to manage a system with many different services compared to one monolith application. Developers need to spend more time dealing with the problems that come with distributed systems. Operations teams need special tools and skills to manage microservices efficiently. This complexity can lead to higher initial costs, longer development times for some features, and a steeper learning curve for teams used to monolith systems.

4.3. Cost Considerations for Growth and Reliability

The costs of choosing between monolith and microservices systems for growth and reliability are complicated and depend a lot on the specific situation, the cloud platform you use, and how you operate the systems.

For monoliths, making the server more powerful (vertical scaling) is the main thing that affects cost when you want to grow. At first, it seems simple, but the cost of vertical scaling usually increases sharply. Moving to more powerful servers in the cloud often involves big jumps in cost between server levels. Also, vertical scaling has a limit based on the most powerful hardware available. If you need to grow beyond that limit, you might have to completely redesign your system. Furthermore, monolith systems often have a problem of "over-provisioning." This means you have to get servers that are powerful enough to handle the highest expected load for the entire application, even if most parts of the application are not using that much power most of the time. This can waste resources and increase costs.

Microservices, while they might save money in the long run by using resources better, can be more expensive to set up initially. Building a microservices system usually means using more virtual servers than a monolith. Each service (or group of services) is often set up separately. The cost of the system that manages microservices (like Kubernetes) and other supporting systems (like service mesh, monitoring tools) adds to the initial cost. The potential cost savings come from using resources efficiently and being able to grow in small steps. By automatically scaling only the services that are busy and scaling based on actual demand, microservices can manage resources dynamically, avoiding waste during quiet periods. The study by Blinowski and others [2] shows that making servers more powerful might be cheaper in some cases in the Azure cloud. But this doesn't mean that microservices can't save money in the long run by growing horizontally and using resources optimally. To get these cost benefits from microservices, you need good automatic scaling policies, effective ways to allocate resources, and maybe use serverless technologies for services that are triggered by events.

Besides infrastructure costs, operational costs are also very different. Running a distributed microservices system is usually more complex and requires advanced skills in areas like tracking requests across services (distributed tracing), collecting logs from different services, and monitoring service performance. This means more operational work and the need for experienced DevOps teams. In comparison, monoliths are easier to operate at first. But managing and fixing problems in large and complex monoliths can become more difficult over time, potentially leading to higher operational costs in the long run due to longer times to fix problems and slower release cycles.

Finally, the cost of tools is important. Microservices require specific tools for managing the system (Kubernetes), service mesh (Istio, Linkerd), monitoring , and automated pipelines for deploying distributed applications (CI/CD). While there are many free and open-source options, the time to install, set up, and manage these tools needs to be considered as a cost in the overall financial calculation.

In conclusion, comparing monolith and microservices systems shows a complex set of trade-offs between reliability and scalability. Monoliths offer simplicity and faster development for applications that are not too big and have predictable workloads. Microservices are better for applications that need extreme scalability, flexibility, and reliability, but they are more complex. Also, cost is another layer of complexity. Monoliths can be cheaper when you can grow by making servers more powerful, while microservices have the potential to save costs in the future by using resources more efficiently at large scale. Ultimately, the best system choice depends on the specific situation, considering the needs for growth and reliability, team skills, development priorities, and long-term operational needs.

5. Gaps in Research and Future Directions

The world of software systems is always changing, and microservices are a new and popular way to build them. However, research shows that we don't know enough about microservices yet. There are big holes in our knowledge about microservices. This makes it hard for researchers to study them and for companies to use them effectively. Even though this paper compares monolith and microservice systems [2] and gives us some good information, it also shows that we need to keep doing more research in this area.

5.1. What Research is Missing

Several important things missing in current research on microservices. First, most of the new microservice ideas are not tested well in real-world situations. They are at a low stage of "Technology Readiness Level" (TRL). This means we need more practical research

to see if these ideas actually work. This is especially important for tools we use to compare different systems (benchmarking). Blinowski et al. [2] mention that when researchers looked hard online, they only found two example systems to test with . And found only one application, Acme Air, that could be used for benchmarking. Because we don't have enough good, up-to-date tools to compare systems, it's difficult to do good and reliable research on this topic.

We need more real-world examples (case studies) and evaluations. w don't have enough proof from real companies, especially smaller ones. Blinowski et al. [2] also point out this problem: "Small companies might think they will get the same benefits as big companies like Netflix or Amazon if they switch from a monolith to microservices. But this might not be true. We don't really know if microservices are helpful for systems that don't have many users at once and can be made bigger by using stronger servers. The research we have is not clear." This means we have a big gap in research: we need to understand when microservices are actually useful, and for what types of companies, not just for the famous tech giants.

Third, we don't have standard ways to test and compare systems (benchmarking techniques). Blinowski et al. [2] say, "Because people test systems in different ways and under different conditions, the results are confusing. Some tests show microservices are faster, and others show monolith systems are faster." Because tests are not done the same way, it's hard to compare research results and make general conclusions. Also, their paper [2] focuses on speed and growth, but we also need to compare how cost-effective these systems are, especially when using different cloud services. Their study looks at costs in Azure cloud only, and it says, "Our cost analysis is only for Azure prices. If you want to use other cloud services, you need to check their prices carefully before assuming our results will apply." Research should look at more than just one cloud service to give us a better overall idea of how costs are affected when we choose different system designs in different cloud environments.

Blinowski et al. [2] highlight, we also really need more research on how to test if microservices are reliable (resilience testing). Microservices are spread out across different parts, which makes it tricky to make sure they can handle problems and keep the system stable. In the introduction, we said that microservices are supposed to be better at "availability, fault tolerance, and horizontal scalability." However, papers like [2] don't always go deeply into testing these aspects comparatively. We need to find good ways to create problems on purpose (like in [3]) and test how well microservice systems can handle them compared to monoliths across various scenarios. This is important for building systems that don't fail easily.

Finally, we don't have enough clear best practices and advice on when to choose monolith or microservice systems, especially based on what the application needs for growth and reliability. Blinowski et al. [2] conclude by saying, "Microservices are not always the best choice. Monolith systems might be better for simple, small systems that don't need to handle many users at once. We hope our findings help companies avoid using microservices just because they are popular, especially if they can get better results by making their monolith systems stronger." We need to turn research findings into practical advice for people in the industry. This will help bridge the gap between what researchers are finding and what companies are actually doing.

5.2. What Research Should Do Next

Based on these gaps, here are some specific things future research should focus on:

1. Create open-source, frequently updated, and standard sets of tests (benchmarking suites). These should be used to compare how well microservice systems perform in terms of growth, speed, and reliability compared to monolith systems. These tests should be done on major cloud platforms like AWS, GCP, Azure, etc. The tests should be like real-world applications in terms of workload and complexity and should be updated regularly. This will address the need for standard testing methods and the lack of ready-to-use tools. The benchmarking application used in [2] ("We also believe that both researchers and practitioners can benefit from our reference benchmarking application... and use it as a starting point for further experimentation.") could be a starting point for creating these standard test suites.
2. Do more real-world research, like detailed case studies in companies. This research should measure the actual costs, performance, and development trade-offs of using monolith versus microservice systems in different types of organizations (not just big tech companies). It should figure out the good and bad points for startups and small businesses, as mentioned as important by [2]. This will address the need for more real-world evaluations and a better understanding of cost-effectiveness.
3. Investigate new and easy ways to use techniques that make microservices reliable (like circuit breakers, bulkheads, retry mechanisms, etc.). This research should look at automation, tools, and system designs that make it less complicated and less work to build reliable microservice systems. This will address the gap in resilience testing and complexity management.
4. Explore system designs that mix parts of monolith and microservice approaches (hybrid architectures). This could give us better growth, reliability, easier development, and cost savings. This might involve finding the most important parts of a monolith application that would benefit from being broken down into microservices, while keeping other parts as a monolith. This will address the lack of more advanced advice beyond simply choosing between monolith or microservices.
5. Conduct research and create automated tools to help software designers decide whether to use monolith or microservice systems. This decision should be based on the application's needs (team size, growth needs, reliability needs, complexity) and the features of the cloud environment (available services, cost models, etc.). These tools could include systems that help with decision-making, cost estimation, and predicting performance. They can help provide clear best practices and guidelines for choosing system architectures.

By studying these areas, we can move towards a better and more fact-based understanding of monolith and microservice systems. This will ultimately help organizations make smarter and more informed choices about system design that fit their specific needs and limitations.

6. Conclusion

6.1. Key Takeaways

This review looked at how well monolith and microservice systems grow and handle problems in cloud setups. We found that neither system is always better. Which one is good depends on what the application needs, what the business wants, and what is practical to manage.

Monolith systems are built as one big piece. They are easier to start with for building, setting up, and growing at first (vertical scaling). They can be cheaper for applications that don't need to grow too much, especially if you can just use a more powerful server, as suggested by cost analysis in specific scenarios [2]. But, monoliths have problems growing in many directions (horizontal scaling), being flexible, and stopping failures from spreading (fault isolation). When applications get bigger and more complex, these problems become obvious. To grow a monolith, you often have to grow the whole thing, which wastes resources and can cause bottlenecks. Also, if one part of a monolith fails, it can break the whole system.

Microservice systems are made of smaller, separate parts. They are designed to grow easily in many directions (horizontal scaling) and be more reliable [5]. Because they are separated, each part can be set

up and grown on its own. If one part fails, it doesn't have to affect the others as much (fault tolerance [3]). Microservices are good for cloud environments that change a lot, need to be very flexible, and must be available all the time. They can handle changes in workload and keep the business running. However, microservices are more complex to build, set up, and run. You need more advanced tools and skills in distributed systems to build, set up, and watch over a microservice system properly. You also need good ways of working to manage them.

When choosing between these systems, architects need to think about the trade-offs. It's about balancing simplicity with how much the system needs to grow and how reliable it needs to be. Monoliths are easier to start with and faster to build at the beginning. They are good for simpler applications with predictable work. Microservices are more complex to set up at first, but they are much better for large, changing, and very important applications that need to grow a lot and be very reliable. But you will need to spend more money and effort on infrastructure, tools, and people with special skills for microservices.

6.2. Final Thoughts and Implications

Cloud systems are always changing and getting better. Microservices have become a popular way to build systems that can grow. But, this review shows that monolith systems are still useful in some cases, aligning with findings like those in [2]. The discussion about monoliths and microservices is not about picking a winner. It's about making smart choices based on the situation. The things we still need to research, like better ways to compare systems, real-world examples, and tools to help decide, show that we need to keep studying these systems. We need to understand better how they work in practice and what the trade-offs are.

For people who work with cloud systems, this review is important. It shows that you need to really understand how monolith and microservice systems grow and stay reliable. You also need to know how complex and costly they are. Just using microservices because they are new and popular without thinking about what your application really needs can cause problems. It can make things too complex and you might not even get the benefits you expected. On the other hand, if you stick with monolith systems for applications that really need to grow and be reliable, you might limit your growth and be more likely to have failures. Future research should focus on giving practical advice and tools to help architects make these difficult choices. They need to choose the best system that fits the business goals and what the technology can do. Looking into hybrid systems that mix parts of monolith and microservice styles could also be helpful. This might help get the right balance of simplicity, growth, and reliability in different cloud situations.

References

[1] B. Christensen, "Introducing hystrix for resilience engineering", *Netflix TechBlog*, 2012, Accessed: 2025-04-06. [Online]. Available: https://netflixtechblog.com/introducing-hystrix-for-resilience-engineering-13531c1ab362.

[2] G. Blinowski, R. Nowak, and R. Spiewak, "Monolithic vs. microservice architecture: A performance and scalability evaluation", in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2022, pp. 667–671. DOI: 10.1109/ACCESS.2022.3152803. [Online]. Available: https://ieeexplore.ieee.org/document/9717259.

[3] J. Flora, P. Gonçalves, M. Teixeira, and N. Antunes, "A study on the aging and fault tolerance of microservices in kubernetes", *IEEE Access*, vol. 10, pp. 132 786–132 799, 2022. DOI: 10.1109/ACCESS.2022.3231191.

[4] Cloud Security Alliance, *New cloud security alliance report finds cloud services are well rooted in all aspects of financial services*, Accessed: 2025-04-06, Jun. 2023. [Online]. Available: https://cloudsecurityalliance.org/press-releases/2023/06/06/new-cloud-security-alliance-report-finds-cloud-services-are-well-rooted-in-all-aspects-of-financial-services.

[5] V. Velepucha and P. Flores, "A survey on microservices architecture: Principles, patterns and migration challenges", *IEEE Access*, vol. 11, pp. 88 339–88 358, 2023. DOI: 10.1109/ACCESS.2023.3305687.

[6] H. Ahmad, C. Treude, M. Wagner, and C. Szabo, "Smart hpa: A resource-efficient horizontal pod auto-scaler for microservice architectures", in *2024 IEEE 21st International Conference on Software Architecture (ICSA)*, 2024, pp. 46–57. DOI: 10.1109/ICSA59870.2024.00013.

[7] Gartner, Inc. "Gartner forecasts worldwide public cloud end-user spending to surpass $675 billion in 2024". Accessed: 2025-04-06, Gartner, Inc. (May 2024), [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/2024-05-20-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-surpass-675-billion-in-2024.

[8] P. R. Santos and J. Bernardino, "Factors affecting cloud computing adoption in the education context—systematic literature review", *IEEE Access*, vol. 12, pp. 65 739–65 758, 2024. DOI: 10.1109/ACCESS.2024.3400862. [Online]. Available: https://ieeexplore.ieee.org/document/10530270.

[9] "istio", *Istio: Open platform for secure, reliable, and observability of microservices*, Accessed: 2025-04-06, 2025. [Online]. Available: https://istio.io/.

[10] Cloud Native Computing Foundation, *Cloud native principles*, Placeholder for CNCF principles, Accessed 2024. [Online]. Available: https://www.cncf.io/.

[11] Linkerd Authors, *Linkerd service mesh*, Placeholder for Linkerd documentation, Accessed 2024. [Online]. Available: https://linkerd.io/.

[12] A. Wiggins, *The twelve-factor app*, Placeholder for 12factor.net methodology, Accessed 2024. [Online]. Available: https://12factor.net/.