
| RESEARCH ARTICLE

Demystifying Microservices Architecture: Breaking the Monolith

Mounika Lakka

UnitedHealth Group, USA

Corresponding Author: Mounika Lakka, **E-mail:** mounikalakka53@gmail.com

| ABSTRACT

The transformation from monolithic to microservices architecture represents a fundamental evolution in enterprise software development, addressing the growing complexity of modern applications through distributed system design principles. Monolithic applications, resembling traditional department stores where all functions operate within unified structures, face significant limitations, including deployment bottlenecks, scaling inefficiencies, and technology constraints that hinder organizational agility and innovation speed. Microservices architecture decomposes these large applications into collections of independent services that communicate through well-defined interfaces, enabling organizations to achieve unprecedented levels of operational flexibility and resource optimization. Each microservice functions as an independent business unit with dedicated technologies, autonomous deployments, and specialized operational procedures tailored to specific business requirements. The architectural paradigm facilitates loose coupling through formalized communication protocols, API-first design approaches, and database-per-service patterns that eliminate shared dependencies and enable independent service evolution. Granular scalability capabilities allow organizations to allocate computing resources precisely where needed, optimizing infrastructure costs while maintaining performance standards during varying demand patterns. Service discovery mechanisms provide dynamic location and communication capabilities that eliminate hard-coded dependencies, while comprehensive resilience patterns, including circuit breakers, bulkheads, and graceful degradation strategies, ensure system stability during partial failures. The distributed architecture enables development teams to work autonomously while maintaining system-wide consistency through contract-based integration and automated deployment pipelines that accelerate feature delivery and reduce coordination overhead.

| KEYWORDS

Microservices architecture, distributed systems, service discovery, scalability optimization, fault tolerance, loose coupling.

| ARTICLE INFORMATION

ACCEPTED: 12 June 2025

PUBLISHED: 21 July 2025

DOI: 10.32996/jcsts.2025.7.7.87

1. Introduction

The evolution from monolithic to microservices architecture represents one of the most significant paradigm shifts in modern software development, fundamentally transforming how organizations approach application design and deployment strategies. Microservices architecture involves breaking down large, complex applications into smaller, independent services that communicate through well-defined APIs and protocols [1]. This architectural approach enables development teams to work autonomously on different services while maintaining overall system consistency and functionality.

Traditional monolithic applications operate as single deployable units where all components remain tightly coupled within a unified codebase. This resembles massive commercial headquarters where all departments share identical infrastructure and operational procedures. Modern enterprise environments demonstrate significant challenges with monolithic systems, particularly when applications scale beyond certain complexity thresholds. The monolithic approach creates deployment bottlenecks where any change requires complete system redeployment, leading to extended downtime periods and increased coordination overhead among development teams [1].

Microservices architecture fundamentally transforms the monolithic model by decomposing applications into collections of loosely coupled services, each responsible for specific business capabilities and operating independently across a distributed infrastructure. Each microservice functions as an independent unit with dedicated databases, processing logic, and communication interfaces, enabling technical optimization for particular business functions. The architectural pattern facilitates independent scaling of individual services based on actual demand patterns rather than scaling entire applications uniformly [1].

Inter-service communication forms the backbone of microservices architecture, enabling distributed components to coordinate activities and share data through formalized protocols and messaging patterns. Communication mechanisms include synchronous approaches, such as HTTP REST APIs, and asynchronous methods involving message queues and event-driven architectures [2]. Synchronous communication provides immediate response capabilities for real-time operations, while asynchronous patterns enhance system resilience by decoupling service dependencies and enabling graceful handling of temporary service unavailability.

The technical implementation of microservices communication relies on multiple protocols and data formats designed to optimize performance and reliability across distributed networks. REST APIs typically use JSON data formats for lightweight data exchange, while message queuing systems employ protocols such as Advanced Message Queuing Protocol (AMQP) for reliable asynchronous communication [2]. Service discovery mechanisms enable dynamic location and connection establishment between services, eliminating hard-coded dependencies and supporting elastic scaling scenarios.

Fault tolerance and resilience patterns are critical considerations in microservices environments where network partitions and service failures represent normal operational conditions rather than exceptional circumstances. Circuit breaker patterns prevent cascading failures by temporarily isolating problematic services, while retry mechanisms with exponential backoff strategies handle transient network issues [2]. Load balancing distributes requests across multiple service instances to prevent individual services from becoming overwhelmed during peak demand periods.

The transformation to microservices architecture aligns with contemporary development practices, including continuous integration, containerization technologies, and cloud-native deployment strategies. Container orchestration platforms provide automated scaling, health monitoring, and service discovery capabilities essential for managing distributed microservices environments effectively. The architectural approach enables organizations to adopt polyglot programming strategies where different services use optimal technology stacks for specific requirements rather than conforming to uniform technological constraints.

2. The Monolithic Foundation and Its Limitations

Monolithic architecture fundamentally operates as a single deployable unit where all application components remain tightly integrated within a unified codebase, resembling a traditional department store where every function—from inventory management to customer service—operates within a single, unified structure. The architectural pattern consolidates all business logic, data access layers, and user interface components into one cohesive application, sharing common infrastructure including databases, runtime environments, and deployment pipelines. Service-Oriented Architecture (SOA) and monolithic systems share some integration elements, yet monolithic applications maintain tighter coupling between functional modules, creating interdependencies that extend throughout the entire system architecture [3].

Traditional monolithic systems served organizations effectively during early software development phases when applications maintained relatively simple functional requirements and development teams remained small within single organizational units. The architectural approach becomes problematic as applications grow in complexity and scale, particularly when multiple development teams attempt to work simultaneously on shared codebases. SOA implements attempts to address some monolithic limitations through service interfaces, but monolithic deployments still require comprehensive coordination across all system components regardless of change scope [3].

Deployment challenges in monolithic systems illustrate the complexities of renovating occupied commercial buildings where ongoing business operations must continue during infrastructure modifications. Every code change, regardless of functional scope, requires comprehensive coordination across all development teams, extensive quality assurance testing cycles, and scheduled downtime windows to prevent disruptions to unrelated business functions. The coordination overhead creates substantial friction within development processes and significantly reduces innovation delivery speed to end users.

Microservices design patterns emerge as solutions to monolithic architecture limitations by promoting service autonomy and independent deployment capabilities. Circuit breaker patterns prevent system-wide failures by isolating problematic components, while bulkhead patterns compartmentalize resources to prevent resource exhaustion in one service from affecting

others [4]. Database-per-service patterns eliminate shared data dependencies that characterize monolithic infrastructures, enabling individual services to optimize data storage strategies for specific functional requirements.

Performance bottlenecks in monolithic systems occur when individual components experience varying load patterns, yet the entire application must scale uniformly to accommodate peak demand in any single functional area. Database contention frequently emerges when multiple application modules compete for shared data resources, leading to performance degradation across unrelated system components. The shared infrastructure model prevents targeted optimization strategies that could address specific performance requirements for individual business functions.

Maintenance complexity increases exponentially as monolithic applications expand in functionality, with development teams reporting that change impact analysis becomes increasingly difficult when codebases exceed manageable complexity thresholds. The interconnected nature of monolithic components creates ripple effects where isolated changes introduce unexpected behaviors in distant application architecture sections. Saga patterns in microservices environments provide distributed transaction management capabilities that monolithic systems struggle to implement efficiently [4].

Technology flexibility remains severely constrained in monolithic infrastructures where initial technology choices establish permanent limitations for entire application lifecycles. Microservices design patterns enable polyglot persistence and programming language diversity, allowing individual services to adopt optimal technologies for specific requirements [4]. Database migration efforts in monolithic systems require coordinated updates across all application components simultaneously, creating significant risk exposure and extended project timelines compared to service-by-service migration strategies available in distributed architectures.

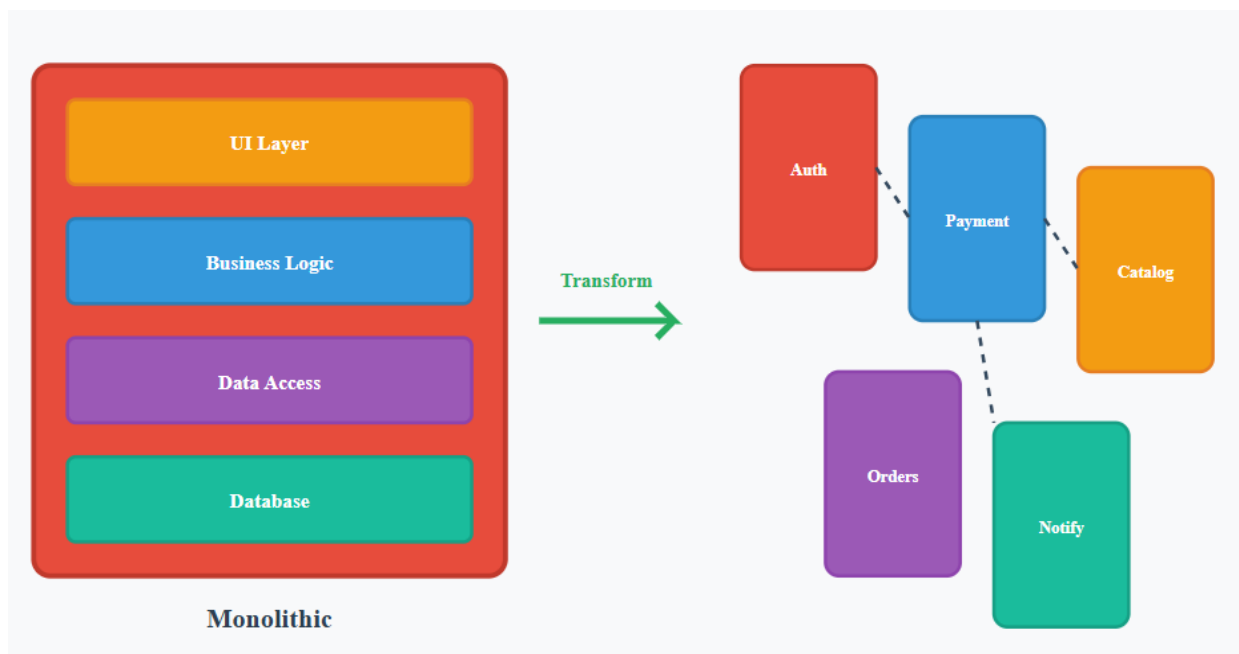


Fig 1. Monolithic vs Microservices Architecture Comparison [3, 4].

3. Understanding Microservices Through Business Analogies

Microservices architecture fundamentally transforms the monolithic department store model into a modern business district where each service operates as an independent storefront with complete operational autonomy. The architectural paradigm enables organizations to decompose large applications into collections of small, loosely coupled services that communicate through well-defined interfaces. Decomposition patterns break down complex systems into manageable components, allowing customer service divisions to operate from dedicated locations with specialized staff, equipment configurations, and independent decision-making capabilities. Inventory management services operate from entirely separate facilities optimized for storage operations and logistics coordination [5].

Each microservice functions as a specialized business unit, maintaining individual leadership structures, technology stack selections, and operational procedures tailored to specific business requirements. Database-per-microservice patterns ensure complete data ownership and eliminate shared database dependencies that create bottlenecks in traditional architectures [5].

Payment processing services may use entirely different programming languages, database systems, and infrastructure configurations compared to user authentication services, corresponding to how law firms and medical practices within the same business district operate with completely different operational requirements and internal system infrastructures.

Operational independence extends comprehensively to staffing models and development lifecycle management through DevOps culture principles that enable teams responsible for notification services to implement updates, scale infrastructure resources, and deploy new features without coordinating with product catalog teams. DevOps culture emphasizes continuous integration and continuous deployment practices that reduce manual intervention and accelerate delivery cycles [6]. The independent operational model enables rapid innovation cycles and significantly reduces coordination overhead that historically plagued monolithic systems during development and deployment phases.

Saga patterns manage distributed transactions across multiple microservices by coordinating sequences of local transactions, ensuring data consistency without requiring traditional two-phase commit protocols. The pattern enables complex business processes to span multiple services while maintaining transactional integrity through compensating actions when failures occur [5]. Load balancing mechanisms distribute incoming requests across multiple service instances to prevent individual services from becoming overwhelmed during peak demand periods, analogous to how shopping centers manage traffic flow during busy retail seasons.

Service discovery patterns enable microservices to locate and communicate with each other dynamically without hard-coded network addresses or configuration dependencies. Registry-based discovery mechanisms maintain centralized service location information, while service mesh architectures provide infrastructure-level capabilities for secure communication between distributed services [5]. Circuit breaker patterns prevent cascading failures by temporarily isolating problematic services, ensuring that issues in payment processing systems don't disrupt completely unrelated services such as product recommendation engines.

DevOps culture principles emphasize infrastructure as code practices that enable consistent environment provisioning and configuration management across development, testing, and production environments. Containerization technologies facilitate service packaging and deployment automation, while orchestration platforms provide automated scaling, health monitoring, and rolling update capabilities [6]. The automated approach ensures that microservices can maintain operational consistency while enabling independent evolution and scaling based on actual demand patterns.

Event sourcing patterns capture all changes to application state as sequences of events, enabling microservices to maintain complete audit trails and support complex business logic reconstruction. Command Query Responsibility Segregation (CQRS) separates read and write operations to optimize performance and scalability for different access patterns [5]. These patterns enable services to maintain eventual consistency while supporting high-throughput operations and complex business rule execution.

Monitoring and observability automation provide comprehensive visibility into distributed microservices behaviors through automated log aggregation, metrics collection, and distributed tracing capabilities. DevOps culture extends to incident response and remediation through automated alerting, self-healing systems, and automated rollback procedures when deployments encounter issues [6]. The comprehensive automation approach ensures that microservices environments can maintain operational reliability while supporting rapid feature delivery and independent service evolution cycles.

4. Achieving Loose Coupling in Distributed Systems

Loose coupling in microservices architecture functions like the relationship between independent businesses in a commercial district, where each service defines clear interfaces for interaction, analogous to how businesses establish standard protocols for customer service, payment processing, and communication channels. Microservices represent small, independent services that communicate over well-defined APIs, enabling organizations to build applications as suites of services rather than monolithic blocks [7]. Inventory services operate without requiring detailed knowledge of the recommendation engine's internal workings; instead, they respond to standardized requests for product availability through established interface contracts that maintain consistent behavior regardless of underlying implementation changes.

Independent deployment capabilities distinguish microservices from traditional service-oriented architectures by enabling individual services to undergo updates, scaling, and technology changes without affecting other system components. Each microservice can be developed, deployed, and scaled independently, allowing organizations to respond quickly to changing business requirements [7]. The deployment independence parallels how commercial businesses can renovate, expand, or modify operations without disrupting neighboring establishments in the same business district.

Database isolation patterns ensure that each microservice maintains complete ownership over its data, eliminating shared database dependencies that create tight coupling in traditional architectures. Services communicate through APIs rather than shared databases, preventing data integrity issues and enabling independent schema evolution [8]. The data ownership model allows user profile services to modernize database schemas while other services continue operating normally, provided external interface contracts remain stable and consistent.

Communication protocols between microservices emphasize lightweight, technology-agnostic approaches that enable language and platform diversity across distributed systems. HTTP-based REST APIs provide standardized communication mechanisms that work across different programming languages and runtime environments. Event-driven architectures enable services to communicate through published events rather than direct synchronous calls, further reducing coupling by eliminating the need for services to maintain detailed knowledge about downstream consumers.

Service boundaries define a clear separation of concerns by aligning microservice responsibilities with specific business capabilities rather than technical layers. Each microservice should represent a single business function with clear inputs, outputs, and responsibilities that minimize dependencies on other services [8]. The business capability alignment ensures that services can evolve independently while maintaining system-wide consistency and avoiding the distributed monolith anti-pattern, where services remain tightly coupled despite physical separation.

Technology diversity enables teams to select optimal programming languages, databases, and frameworks for specific service requirements without constraining the entire system to uniform technology stacks. The polyglot approach allows payment processing services to use different technologies compared to user authentication services, analogous to how law firms and medical practices in the same business district operate with entirely different operational requirements [7]. Technology flexibility enables organizations to adopt emerging technologies incrementally without requiring comprehensive system rewrites.

Fault isolation prevents failures in individual services from propagating throughout the entire system architecture, ensuring that problems in payment processing components don't disrupt unrelated services such as product inventory or recommendation engines. Circuit breaker patterns and timeout mechanisms provide additional protection against cascading failures while maintaining system responsiveness during partial outages [8]. The isolation approach enables services to fail independently while maintaining overall system integrity and user experience.

Contract-based integration establishes explicit agreements between services regarding interface specifications, data formats, and behavioral expectations, enabling independent evolution while maintaining interoperability. API contracts define service capabilities without exposing internal implementation details, allowing services to undergo internal restructuring without affecting dependent components [7]. The contract approach reduces technical debt accumulation and enables organizations to modernize individual services incrementally without requiring coordinated system-wide overhauls that create significant risk exposure and extended project timelines.

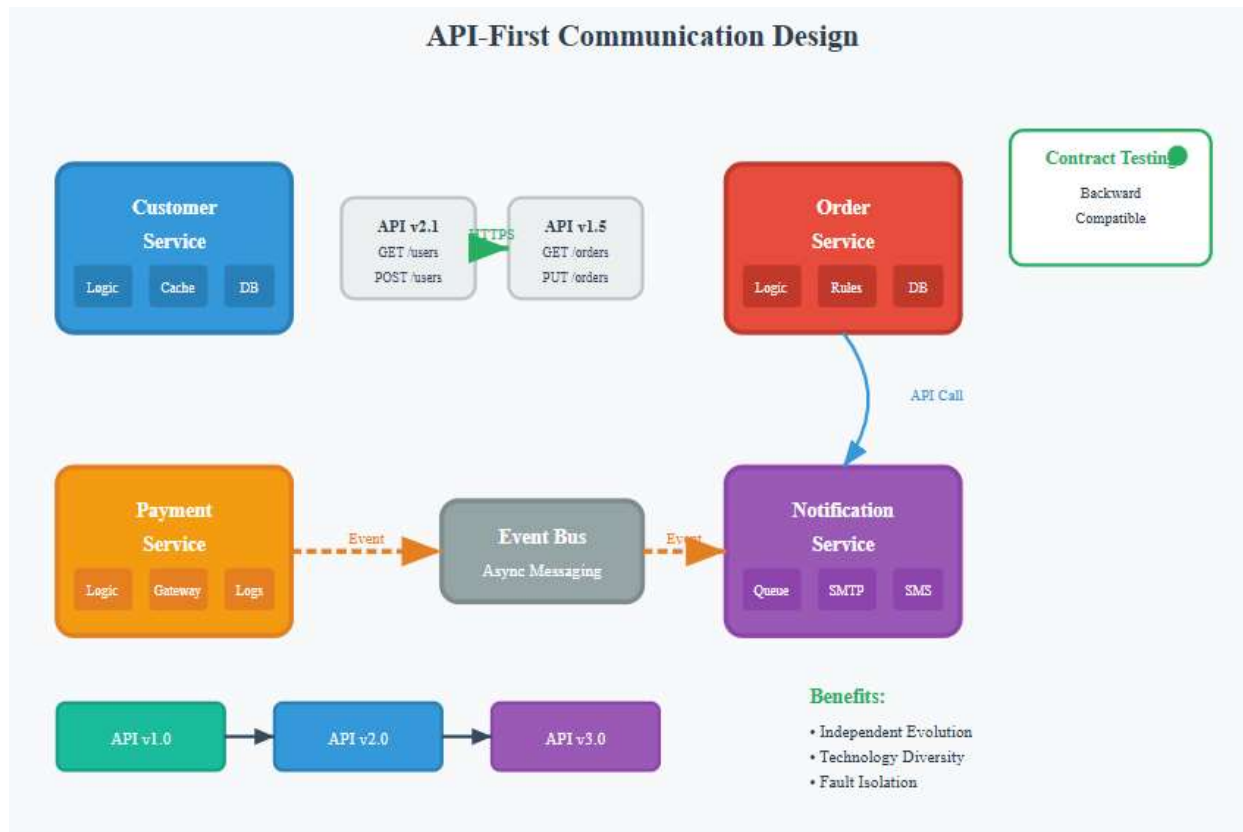


Fig 2. Loose Coupling Communication Patterns [5, 6].

5. Scalability and Performance Optimization

Microservices architecture enables granular scalability that mirrors how successful business districts grow organically, allowing individual services to scale independently based on specific demand patterns and resource requirements. Independent scaling represents a strategic approach where organizations can scale specific services experiencing high demand while maintaining standard operations for other components [9]. During peak shopping seasons, retail stores might hire additional staff and extend operating hours while neighboring businesses maintain normal operations, similarly enabling image processing services to scale up during high-traffic periods while user authentication services operate at standard capacity.

Targeted scaling approaches optimize resource utilization and significantly reduce operational costs by enabling organizations to allocate computing resources precisely where needed rather than over-provisioning entire application stacks. Database services might require high-performance storage configurations and memory optimization strategies, while notification services prioritize network throughput capabilities and message queuing performance. The granular approach eliminates waste associated with scaling entire monolithic applications when only specific components experience increased load [9].

Load distribution mechanisms ensure that incoming requests are spread across multiple service instances to prevent individual services from becoming overwhelmed during peak demand periods. Microservices infrastructures facilitate horizontal scaling by adding more instances of specific services rather than vertically scaling hardware resources [10]. The horizontal scaling approach provides better fault tolerance and cost effectiveness compared to traditional vertical scaling methods that require expensive hardware upgrades and create single points of failure.

Performance monitoring becomes significantly more sophisticated in microservices environments, enabling development teams to identify bottlenecks at individual service levels and implement targeted optimization strategies. Observability tools provide detailed insights into service behavior, response times, and resource consumption patterns across distributed systems [10]. The granular monitoring approach enables more effective capacity planning and resource allocation strategies compared to monolithic systems, where performance issues remain difficult to isolate and address.

Resource efficiency improvements result from the ability to right-size individual services based on actual performance requirements rather than provisioning uniform resources across entire application stacks. Each microservice can utilize optimal

resource allocations for specific workload characteristics, enabling CPU-intensive services to receive adequate processing power while memory-intensive services gain sufficient RAM allocation [9]. The tailored resource approach reduces infrastructure costs while improving overall system performance and responsiveness.

Fault isolation capabilities prevent performance degradation in one service from affecting other system components, ensuring that bottlenecks in image processing services don't impact user authentication or payment processing functionality. Circuit breaker patterns automatically isolate failing services while maintaining system integrity for unaffected components [10]. The isolation approach enables organizations to maintain partial system functionality during performance issues while addressing problems in affected services.

Auto-scaling mechanisms automatically adjust service instances based on real-time demand metrics, enabling systems to handle traffic spikes without manual intervention. Container orchestration platforms can dynamically increase or decrease service replicas based on CPU utilization, memory consumption, or custom business metrics [9]. The automated scaling approach ensures optimal resource utilization while maintaining service quality and performance standards during fluctuating business patterns.

Deployment flexibility enables teams to optimize individual services for specific performance characteristics without affecting other system components. Different services can use various deployment strategies, caching mechanisms, and optimization techniques based on unique requirements [10]. The flexible deployment approach allows organizations to experiment with performance optimizations incrementally while maintaining system stability and minimizing risk exposure during optimization initiatives.

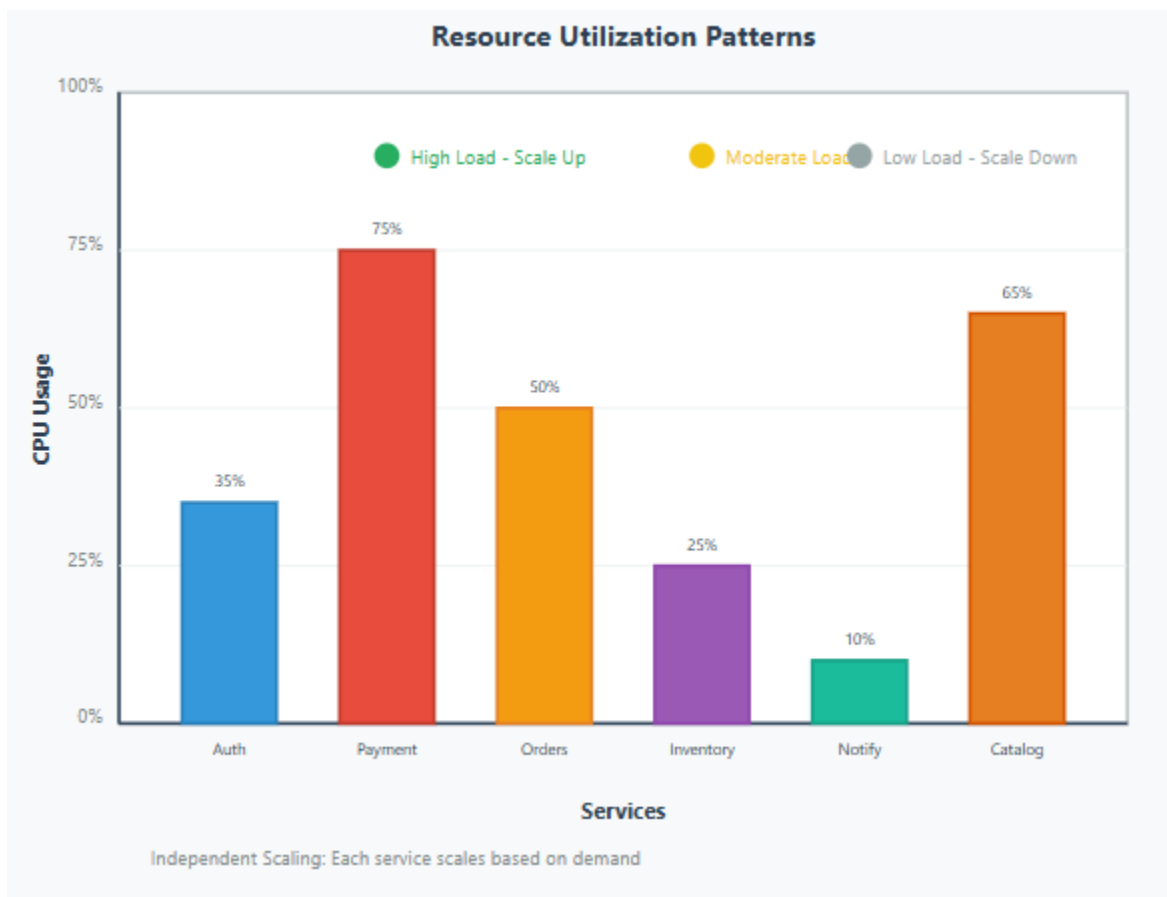


Fig 3. Scalability and Performance Metrics Chart [9, 10].

6. Service Discovery and System Resilience

Service discovery in microservices architecture functions like a business directory that automatically updates as companies relocate, change phone numbers, or modify service offerings, enabling dynamic location and communication between distributed services without hard-coded network addresses. Microservices communicate through networks, making service discovery essential for locating and connecting to dependent services across distributed infrastructure [11]. Services register

availability and capabilities with central registries, allowing other components to locate and communicate with them dynamically through automated discovery mechanisms that eliminate hard-coded dependencies and enable services to move between servers or scale across multiple instances seamlessly.

Client-side discovery patterns enable services to query service registries directly to obtain location information for dependent services, while server-side discovery utilizes load balancers or API gateways to handle service location and routing automatically. Service registries maintain real-time information about service instances, including network addresses, health status, and metadata that supports intelligent routing decisions [11]. The discovery mechanisms ensure that services can adapt to changing infrastructure conditions while maintaining reliable communication pathways across distributed environments.

Resilience in distributed systems requires implementing multiple layers of fault tolerance that protect against various failure modes, including network partitions, service crashes, and performance degradation. Circuit breaker patterns prevent cascading failures by temporarily isolating problematic services, analogous to how electrical systems protect against overloads through automatic circuit mechanisms. Bulkhead patterns isolate resources to prevent failures in one area from affecting others, while timeout patterns prevent services from waiting indefinitely for responses [12].

Retry patterns with exponential backoff strategies help services handle transient failures gracefully by attempting failed operations multiple times with increasing delays between attempts. The pattern prevents overwhelming failing services while providing opportunities for temporary issues to resolve naturally [12]. When payment processing services experience issues, circuit breakers prevent repeated failed requests while alternative payment methods remain available, maintaining system functionality during partial outages.

Health check patterns continuously monitor service availability and performance characteristics, enabling automatic detection of service failures and triggering appropriate recovery actions. Load balancers can remove unhealthy service instances from rotation, while container orchestration platforms can replace failed services or provision replacement instances [11]. The automated health monitoring approach reduces recovery time by detecting and addressing service failures without manual intervention or incident reporting.

Redundancy strategies ensure that system functionality persists even when individual services fail or become unavailable during maintenance or unexpected outages. Recommendation engines might switch to cached results or simplified algorithms when real-time processing becomes unavailable, maintaining user experience while addressing underlying issues. Graceful degradation patterns enable systems to provide reduced functionality rather than complete failure, preserving core business operations during service disruptions [12].

Saga patterns manage distributed transactions across multiple microservices by coordinating sequences of local transactions, ensuring data consistency without requiring traditional two-phase commit protocols. The pattern handles complex business processes spanning multiple services while maintaining transactional integrity through compensating actions when failures occur [12]. Event sourcing patterns capture all changes to application state as sequences of events, enabling services to maintain complete audit trails and support complex business logic reconstruction.

Monitoring and observability capabilities provide comprehensive visibility into distributed system behavior through centralized logging, distributed tracing, and metrics collection that enable rapid problem identification and resolution. Service mesh architectures offer additional observability features, including automatic traffic collection, security policy enforcement, and detailed telemetry collection across service communications [11]. The observability approach enables proactive problem detection and faster incident response through automated alerting and detailed system analysis across distributed microservices environments.

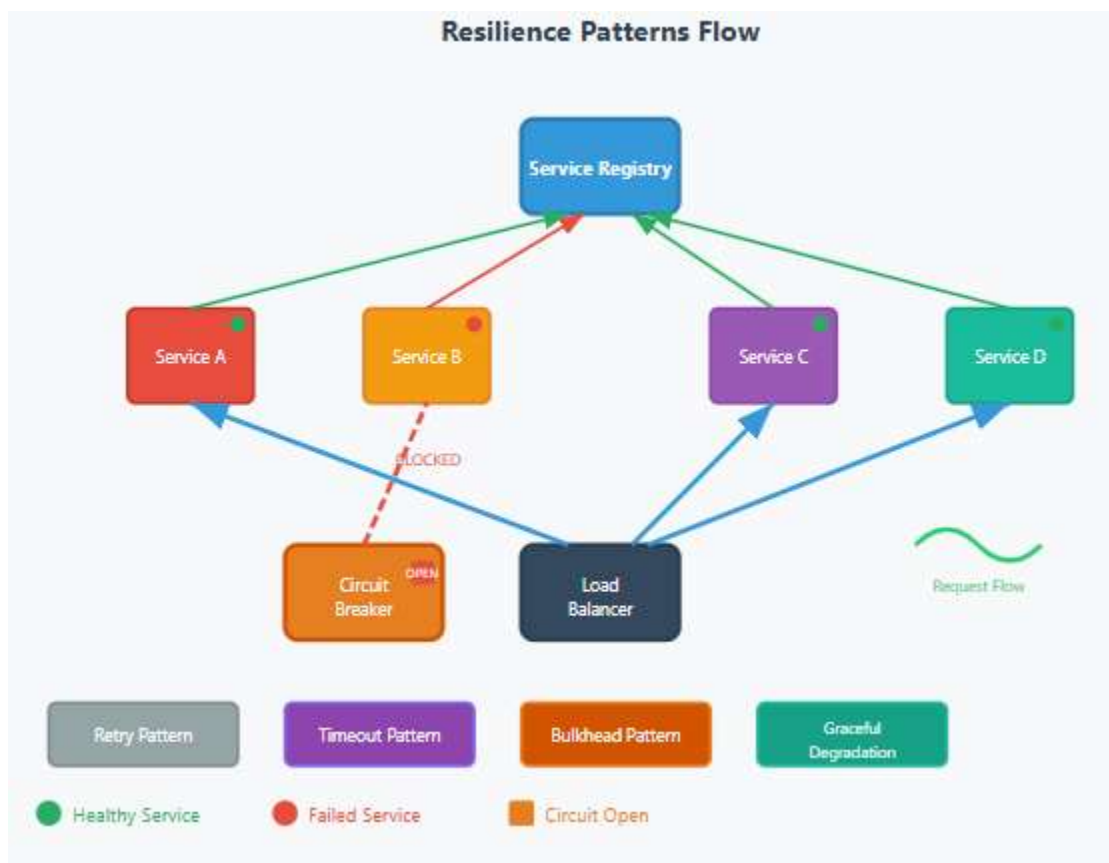


Fig 4. Service Discovery and Resilience Patterns [11, 12].

7. Conclusion

The architectural evolution from monolithic to microservices represents more than a technical transformation; it constitutes a fundamental reimagining of how organizations can design, deploy, and maintain software systems in increasingly complex digital environments. Microservices architecture enables enterprises to overcome the inherent limitations of monolithic systems by providing granular control over individual service components while maintaining overall system consistency through well-defined interfaces and communication protocols. The business district analogy effectively illustrates how independent services can operate autonomously while contributing to larger organizational objectives, analogous to how specialized businesses within commercial districts serve distinct functions while supporting the broader economic ecosystem. Implementation of microservices patterns, including circuit breakers, service discovery, database-per-service, and automated scaling mechanisms, creates robust distributed systems capable of handling modern enterprise requirements for agility, reliability, and efficiency. The architectural approach addresses critical challenges, including deployment coordination overhead, technology stack restrictions, and uniform scaling limitations that characterize traditional monolithic applications. Organizations adopting microservices architecture gain significant competitive advantages through accelerated development cycles, improved fault isolation, and the ability to scale individual components based on actual demand patterns rather than uniform resource allocation strategies.

The distributed nature of microservices enables teams to select optimal technologies for specific service requirements while maintaining system-wide consistency through formalized communication protocols and contract-based integration strategies. Success with microservices requires embracing the complexity of distributed systems while implementing comprehensive automation, monitoring, and resilience practices that ensure operational stability and performance optimization across service boundaries. As digital transformation initiatives continue accelerating across industries, microservices architecture provides the foundational capabilities necessary for organizations to build scalable, resilient, and efficiently managed applications that can adapt to evolving business requirements while optimizing infrastructure investments and operational effectiveness.

Funding: This research received no external funding

Conflicts of Interest: The author declare no conflict of interest.

Publisher's Note: All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers

References

- [1] Angela D (2023) An In-Depth Guide to Microservices Design Patterns, OpenLegacy, 2023. [Online]. Available: <https://www.openlegacy.com/blog/microservices-architecture-patterns/>
- [2] Aykut B, (2023) A Guide to Microservices Design Patterns for Java, diffblue, 2023. [Online]. Available: <https://www.diffblue.com/resources/a-guide-to-microservices-design-patterns-for-java/>
- [3] Daniel L, (2017) Building Microservices: Designing Fine-Grained Systems, by Sam Newman, 017. [Online]. Available: <https://danielleaver.com/2017/09/24/building-microservices-designing-fine-grained-systems-by-sam-newman/>
- [4] Geeksforgeeks, (2024) Inter-Service Communication in Microservices, 2024. [Online]. Available: <https://www.geeksforgeeks.org/system-design/inter-service-communication-in-microservices/>
- [5] Geeksforgeeks, (2025) What are Microservices?, 2025. [Online]. Available: <https://www.geeksforgeeks.org/system-design/microservices/>
- [6] Google Cloud, (2024) Introduction to microservices, 2024. [Online]. Available: <https://cloud.google.com/architecture/microservices-architecture-introduction>
- [7] Gui A, (2022) SOA vs Microservices: What's the Difference? CrowdStrike, 2022. [Online]. Available: <https://www.crowdstrike.com/en-us/cybersecurity-101/cloud-security/soa-vs-microservices/>
- [8] Hossein A (2022) What Are Microservices? Definition, Examples, Architecture, and Best Practices for 2022, SpiceWorks, 2022. [Online]. Available: <https://www.spiceworks.com/tech/devops/articles/what-are-microservices/>
- [9] Jay L, (2023) What are microservices, and what are their benefits? Ynatac 2023. [Online]. Available: <https://www.dynatrace.com/news/blog/what-are-microservices/>
- [10] Jetinder S, (2018) The What, Why, and How of a Microservices Architecture, Medium, 2018. [Online]. Available: <https://medium.com/hashmapinc/the-what-why-and-how-of-a-microservices-architecture-4179579423a9>
- [11] Rajesh B (2024) Microservices Design Patterns: Essential Architecture and Design Guide, DZone, 2024. [Online]. Available: <https://dzone.com/articles/design-patterns-for-microservices>
- [12] Satyendra J, (2024) Principles of Microservices — Culture Of Automation, Medium, 2024. [Online]. Available: <https://medium.com/@satyendrajaiswal/principles-of-microservices-culture-of-automation-0e55b446ad24>