Sarcouncil Journal of Engineering and Computer Sciences

ISSN(Online): 2945-3585

Volume- 04| Issue- 08| 2025

Research Article **Received:** 27-06-2025| **Accepted:** 18-07-2025 | **Published:** 07-08-2025

Microservices and Serverless Computing: Architectural Patterns for Modern Distributed Systems

Srinivasa Rao Gunda
Independent Researcher, USA

Abstract: Modern software architecture has shifted from monolithic architectures toward distributed architectures such as microservices and serverless computing. Microservices architecture can be characterized as an architecture that disaggregates the application into independently deployable parts (or services) which inherit important characteristics such as scalability, fault-isolation, and flexibility. Serverless computing can be thought of as a unique level of abstraction where the developer can solely concentrate on business logic with complete abstraction of the control and complexity of managing servers. Serverless offers automatic scaling of resources and only charges the organization on a pay-as-you-go basis based on usage, complete with an automatically scaling model that allows an organization not to charge for capacity that is not being utilized. With each architecture, there are trade-offs. Microservices can become complex, as microservices can incur overhead in the communication between services. Serverless can potentially incur cold-start latency, along with complexity in addressing latency and vendor lock-in. A hybrid architecture is emerging that allows organizations to leverage attributes from microservices and serverless architectures, so that organizations can gain the advantages from both architectures. Ultimately, factors contribute to making a decision either way based on organizational constraints, project features, and the competence of the team. It is important to have an understanding of the patterns of these distributed architectures and their constraints.

Keywords: Microservices architecture, serverless computing, Function-as-a-Service, distributed systems, cloud-native applications.

INTRODUCTION

Transition from Unified to Distributed Software Structures

Software engineering practices have undergone major transitions in architectural paradigms in the last several decades. Until fairly recently, one-code-base applications, built and deployed as an indivisible whole, were the normative practice for constructing enterprise systems for nearly the entire history of computing. The monolithic form permitted simple deployment strategies and straightforward debugging methods. However, the typical modern enterprise systems exhibit a clear preference for flexibility in scaling, quickness in the delivery of features, and are developed by teams of engineers working together. The pressures of contemporary business expose glaring deficiencies in tightly coupled application design patterns. Essentially, technical debt builds quickly when interacting with multiple connected parts, and horizontal scaling becomes nearly impossible for tightly-coupled apps. When one module exhibits performance issues, the entire business logic is negatively impacted. Furthermore, a significant update to the WAM stack (whatever application management system) requires a change to the entire application. These operational realities compel enterprises to consider decomposed architectural paradigms providing independent and granular capability over their various constituent components (Blinowski, G. *et al*., 2022).

Goals and Discussion of Event-Driven and Decomposed Architectures

This examination addresses two revolutionary approaches reshaping enterprise software design: service-oriented decomposition and function-based computing models. Service-oriented decomposition divides applications into discrete operational units, enabling autonomous lifecycle management for each component. Function-based computing abstractions eliminate infrastructure provisioning concerns entirely, permitting code execution without server management overhead (Nguyen, K. *et al*., 2023). These architectural philosophies transform traditional development workflows, deployment strategies, and operational practices. Current discourse analyzes fundamental concepts underlying each approach, practical implementation techniques, measurable advantages, and technical obstacles encountered during adoption. Furthermore, convergent strategies combining both methodologies receive attention, offering practitioners actionable insights for selecting appropriate architectures based on specific project requirements and organizational capabilities.

FOUNDATIONAL CONCEPTS AND DISTINCTIVE PROPERTIES

Concepts, Partitioning, and Autonomous Deployment of Service-Based Architecture

Contemporary software construction employs component-based strategies where applications

***Corresponding Author:** Srinivasa Rao Gunda

DOI- https://doi.org/10.5281/zenodo.16757614

comprise numerous compact, self-contained modules communicating via standardized protocols. Individual modules encapsulate particular operational logic while maintaining dedicated data repositories, thereby avoiding centralized database coupling. Software partitioning follows domain-driven boundaries, creating cohesive units around business capabilities rather than technical layers (Hilbrich, M., & Lehmann, F. 2022). Technical teams exercise full control over implementation choices within module boundaries, selecting programming languages, frameworks, and storage solutions independently. Release cycles operate autonomously across different modules, enabling continuous updates without system-wide coordination. Versioning strategies support backward compatibility through careful interface management, allowing gradual transitions between module iterations. Network-based communication replaces method calls, introducing latency considerations but providing location transparency and failure isolation benefits.

Event-Triggered Computing: Execution Models and Infrastructure Abstraction

Code execution without server management characterizes modern event-driven platforms where computational units activate through external stimuli. Developers write standalone functions responding to predetermined triggers, bypassing traditional concerns about hardware provisioning or operating system configuration (Ekwe-Ekwe, N. 2024). Triggering mechanisms span diverse sources, including web requests, data modifications, timer schedules, file operations, and queue messages. Platform providers spin up execution instances according to incoming workload patterns, taking care of all scaling decisions automatically. Memory allocation occurs dynamically per invocation, releasing resources immediately after completion. Cost structures reflect actual usage rather than reserved capacity, calculating charges based on execution duration and memory consumption. Development workflows focus exclusively on business logic implementation, as platforms manage runtime environments, security patches, and availability zones transparently.

Fundamental Distinctions Between Architectural Approaches

Component-based systems and event-driven architectures utilize different resource management approaches and have different execution models. Through transitive execution, a component-based architecture allows session processes to manage ongoing processes to maintain component internal state over multiple requests, support demanding workflows, and maintain stateful functions. An event-driven system operates based on an event, as it limits the runtime context and the initialized context of the requests in regard to the container model execution context staggering. An architecture using components typically communicates with components with inter-component communications utilizing various protocols (e.g., REST APIs, message brokers, RPC conventions), while a triggered event-driven function on a platform operates because of an action and as such there is no structural linking between components (i.e., no guaranteed execution context) because events happen and there are no session links. The methods of resourcing greatly differ between component-based and event-driven architectures. Component-based systems must have the agent estimate the capacity in the architecture and implement scaling policies, while event-driven systems provide resources dynamically when making a request. Ultimately, both event-driven and component-based systems can provide flexibility for the overall development complexity because often you are trading off the operational complexity of managing an emerging infrastructure for the constraints of the programming platform and the dependencies of vendor-associated services.

Table 1: Architectural Characteristics Comparison (Blinowski, G. *et al*., 2022; Nguyen, K. *et al*., 2023)

Characteristic	**Microservices**	**Serverless**
Deployment Unit	Service/Container	Function
Execution Model	Long-running processes	Event-triggered execution
State Management	Stateful within service boundaries	Stateless by design
Scaling Granularity	Service-level	Function-level
Resource Allocation	Pre-provisioned/Reserved	Dynamic per invocation
Communication Pattern	Synchronous/Asynchronous APIs	Event-driven triggers
Development Focus	Full service lifecycle	Business logic only
Infrastructure Management	Developer/Operations responsibility	Platform-managed

Publisher: SARC Publisher

BENEFITS TO OPERATIONS AND STRATEGIC ADVANTAGES

Distributed Services: Team Productivity, Error Containment, and Resource Optimization

Modular application designs enable precise resource distribution across system components. Computing power is allocated proportionally to individual module demands, avoiding wasteful replication of underutilized functionalities. Module borders are where load balancing takes place, guiding traffic to instances that are suitably scaled according to real-time measurements. Error propagation halts at module interfaces, preventing single-point failures from compromising entire systems (Rasheedh, J. A., & Saradha, S. 2021). Recovery mechanisms operate independently per module, restoring functionality without system-wide restarts. Rollback procedures target specific modules, minimizing disruption during problematic deployments. Programming teams operate with minimal coordination overhead, as module boundaries define clear ownership and responsibility divisions. Technical decisions remain localized within modules, permitting framework migrations and language updates without cross-team negotiations. Quality assurance processes focus on module-specific behaviors and interface contracts, reducing test complexity and execution time.

Platforms Based on Functions: Usage-Based Pricing, Dynamic Provisioning, and Streamlined Operations

Computational platforms that execute code on demand transform traditional capacity planning paradigms. Processing power materializes instantaneously when requests arrive, accommodating unpredictable traffic patterns without manual intervention. Financial models align expenses directly with consumption, charging organizations only for milliseconds of actual execution (Ugwueze, V. U. 2024). Server administration tasks vanish from developer workflows, as managed platforms handle hardware provisioning, network configuration, and security patching autonomously. Feature delivery accelerates when infrastructure concerns disappear, allowing developers to concentrate on algorithmic improvements and business requirements. Global request routing happens transparently, with platforms selecting optimal execution locations based on latency and availability criteria. Memory and CPU quotas apply per execution, preventing runaway processes from affecting concurrent workloads. Observability features integrate seamlessly, providing metrics and traces without custom implementation efforts.

Contextual Suitability Across Application Domains

Architectural decisions derive from workload characteristics, team capabilities, and business constraints. Deployments of dedicated services that maintain state throughout several contacts are preferred by continuous connectivity requirements. Similarly, batch processing workloads with sporadic schedules benefit from pay-per-execution models that don't incur costs when resources are idle. Tandem processing triggered by events or streamed by a constant influx of viewers requires the continuous processing capabilities of service architectures, especially since purely ephemeral functions don't provide those capabilities. Financial considerations shift around traffic patterns. With predictable workloads, there is potentially less cost than reserved service instances, with consumption-based pricing benefiting variable workloads. The size and skill makeup of the team are directly reflected in architectural decision-making, where larger-scale organizations tend to use services to maintain boundaries between teams, and smaller teams usually find services versus traditional apps to be advantageous in terms of operational overhead. Compliance can sometimes restrict architectural choices by stipulating a required model of deployment, such as certain regulated industries requiring data locality or audit trails. Latency sensitivity, together with cold start penalties, will frame platform choices to balance the operational management information alongside response times.

Table 2: Operational Benefits Matrix (Rasheedh, J. A., & Saradha, S. 2021; Ugwueze, V. U. 2024)

Benefit Category	Microservices	Serverless
Scalability	Independent service scaling	Automatic function scaling
Cost Model	Reserved capacity pricing	Pay-per-execution
Fault Tolerance	Service isolation boundaries	Function-level isolation
Development Speed	Team autonomy per service	Rapid function deployment
Operational Overhead	Requires infrastructure management	Minimal operational burden

Publisher: SARC Publisher

Technology Flexibility	Polyglot services	Platform-constrained languages
Performance Predictability	Consistent latency	Variable (cold starts)

TECHNICAL RESTRICTIONS AND REALISTIC CHALLENGES

Distributed Architecture: State Management, Network Dependencies, and Coordination Overhead

With modularity comes coordination complexity introduced by partitioned software systems not found in monolithic codebases. When a procedure call is remote, one can no longer perform function calls; instead, they are network calls with possible latencies and failures that must be managed with appropriate error handling. Dynamic service registration and lookup further complicates coordination, as services change location at each scale event or upon recovering from failure responses (Fernando, C. 2021). Message ordering when delivering events to distributed subscribers introduces protocol complexity; in business logic where processing order is relevant, care must be taken to select the appropriate protocol. Maintaining an equivalent and coherent data view across independent datastores also introduces coordination that does not fit well into traditional consistency models, and stateful persistence. In some cases, architects are forced to select between strong consistency and system availability. Compensating transaction patterns aimed at achieving idempotent operations replace what would normally be a series of atomic database operations, and rollback logic must be written whenever a multi-service operation fails. Tracking request correlation across component boundaries is also an impediment when troubleshooting user interaction due to the many services the request may cascade through. Configurations hover near zero when servicing requests via many separate deployments of services that rely on settings, and developers must address the burden of maintaining a singular configuration repository with a strategy for environment-specific overrides. Authorization and authentication flows multiply with service endpoints, and passing tokens between these boundaries for validation is critical.

Event-Driven Platforms: Startup Delays, Troubleshooting Barriers, and Platform Dependencies

Execution-on-demand models introduce performance unpredictability through container initialization overhead. Fresh container provisioning for dormant functions creates variable response latencies, particularly problematic for user-facing endpoints (olec, M. *et al*., 2024). Development workflows suffer from limited debugging capabilities, as breakpoints and step-through debugging become impractical in managed environments. Replicating cloud-specific behaviors locally proves challenging, resulting in integration issues discovered only during deployment. Platform-specific service integrations create migration barriers, as code depends on proprietary APIs for storage, messaging, and orchestration. Execution constraints imposed by providers limit processing duration, available memory, and simultaneous executions. Stateless execution mandates external storage for any information persisting beyond single invocations, complicating data access patterns. Verification procedures must accommodate platform-specific trigger formats, permission models, and execution contexts. Operational visibility requires platform-native tooling, as conventional monitoring agents cannot access managed runtime environments.

Architectural Compromises and Remediation Approaches

Technical decisions require evaluating trade-offs while implementing compensating controls for identified risks. Infrastructure overlays simplify service interactions through unified networking policies, traffic management, and security enforcement without modifying application code. Choreographed workflows replace orchestrated transactions, using event-driven compensation for maintaining consistency without distributed locks. Strategic caching reduces cross-service calls while implementing sophisticated invalidation logic to maintain data freshness. Scheduled invocations keep function environments warm, trading minimal costs for improved response predictability. Mixed deployment models position persistent workloads on dedicated computers while routing variable tasks to managed functions. Emerging development platforms improve local testing fidelity through enhanced emulation and integrated debugging capabilities. Continuous cost analysis prevents budget overruns from inefficient implementations or unexpected usage patterns.

Publisher: SARC Publisher

Table 3: Implementation Challenges Overview (Fernando, C. 2021; olec, M. *et al*., 2024)

Challenge Type	Microservices	Serverless	Mitigation Strategy
Complexity	Distributed system management	Platform limitations	Service mesh, abstraction layers
Communication	Network latency, failures	Event format dependencies	Async patterns, retries
Data Consistency	Distributed transactions	External state management	Saga pattern, eventual consistency
Debugging	Distributed tracing needs	Limited runtime access	Correlation IDs, platform tools
Vendor Dependencies	Container orchestration	Cloud provider lock-in	Standardized interfaces
Performance	Inter-service overhead	Cold start latency	Caching, function warming

ARCHITECTURAL COMPARISON AND INTEGRATION STRATEGIES

Contrasting Development and Deployment Methodologies

Service-oriented and function-driven systems embody contrasting implementation philosophies throughout their lifecycles. Containerized services operate continuously after deployment, processing multiple requests while preserving operational context between invocations. Function implementations activate momentarily for individual requests, discarding runtime environments after completion (Gilbert, J. 2021). Building procedures vary substantially, with service development encompassing dependency management, container image creation, and infrastructure provisioning. Function development focuses solely on business logic, delegating environment concerns to managed platforms. Capacity planning distinguishes these approaches fundamentally, as service deployments require upfront resource estimation while function platforms adjust dynamically. Network interactions follow different paradigms, with services establishing direct connections between components versus functions responding to platform-mediated events. Information persistence mechanisms diverge, as services utilize local memory and disk storage while functions rely exclusively on external data stores. Verification processes adapt accordingly, necessitating comprehensive environment replication for services compared to isolated function testing.

Blended Architectures: Integrating Persistent Services with Event Functions

Contemporary systems frequently merge component-based services with event-triggered functions, creating architectures that capitalize on both paradigms. Central processing logic typically executes within dedicated service instances, maintaining transaction integrity and complex state management. Peripheral operations migrate to managed functions, handling file processing, notification dispatch, and periodic maintenance tasks (Gilbert, J. 2021). Request routing layers abstract implementation details, presenting cohesive APIs while directing traffic to appropriate backends. Data pipelines incorporate both models, using services for stream processing while functions perform discrete transformations. Scheduled operations benefit from serverless execution, eliminating idle resource consumption between batch runs. Geographic distribution strategies deploy functions at network edges for latency-sensitive operations while centralizing services for consistency. Resource utilization improves through strategic placement, reserving service capacity for baseline loads while functions absorb traffic spikes.

Selection Criteria for Architectural Patterns

Systematic evaluation frameworks guide architectural decisions based on multifaceted project requirements. Traffic profiles strongly influence selection, where uniform request rates justify dedicated services while sporadic activity favors on-demand functions (Megargel, A. *et al*., 2021). Solution complexity affects architectural fit, as interconnected business processes benefit from service encapsulation while isolated calculations suit function deployment. Personnel skills constrain choices, requiring infrastructure expertise for service management versus platform knowledge for serverless adoption. Response time constraints impact viability, especially when initialization delays exceed acceptable thresholds. Economic analysis encompasses total expenditure, including development effort, operational overhead, and runtime costs across expected usage scenarios. Governance requirements sometimes dictate deployment options, particularly in

Publisher: SARC Publisher

regulated sectors mandating specific security controls. Evolutionary paths from current systems shape adoption strategies, often necessitating transitional architectures during modernization initiatives.

Table 4: Architectural Decision Framework (Gilbert, J. 2021; Megargel, A. *et al*., 2021)

Decision Factor	Favor Microservices	Favor Serverless	Consider Hybrid
Traffic Pattern	Consistent, high-volume	Sporadic, variable	Mixed workloads
Application Complexity	Complex business logic	Simple transformations	Core services + utilities
State Requirements	Stateful operations	Stateless processing	Both requirements
Team Expertise	Container/K8s experience	Cloud platform knowledge	Diverse skill sets
Cost Sensitivity	Predictable budgets	Usage-based optimization	Baseline + variable
Latency Requirements	Low, consistent latency	Tolerates cold starts	Mixed SLA requirements
Compliance Needs	Data locality control	Platform compliance sufficient	Selective control

CONCLUSION

In many ways, modern distributed architectures are changing the paradigm of software delivery by leveraging service-based decomposition and function-based execution. Each architecture solves specific problems but comes with its own operational assumptions that must be examined carefully by the organization. Service-based architectures are well-suited for the following situations: when the system requires persistent state or manual orchestration of many components, when the number of functions and service calls that can degrade performance are known, and when stable performance characteristics can be maintained within the system. Executable platforms take advantage of running workloads using event-driven models that reduce the operational cost of managing infrastructure; instead, the resource usage is "on-demand" and automatically scaled for the number of events per second. Many organizations are finding that a hybrid of both distributed service and function-based architecture is the ideal solution, meaning using persistent services for their business-critical logic, while using serverless functions for other ancillary functions. Which architecture pattern to use depends heavily on factors such as event characteristics and clusters, operational capabilities, budgets, and compliance guidelines. Developments in distributed computing will likely cause more overlap between architectural patterns as tools and services evolve to provide greater flexibility and abstraction. Understanding the architectural options is critical to enabling organizations to make informed building decisions and aligning organizational technical implementation with their target business outcomes. Modern software development is about accepting architectural diversity and making choices based on use case, rather than relying on one story or approach.

REFERENCES

1. Blinowski, G., Ojdowska, A., & Przybyłek, A. "Monolithic vs. microservice architecture: A performance and scalability evaluation." *IEEE access* 10 (2022): 20357-20374.
2. Nguyen, K., Loh, F., Nguyen, T., Doan, D., Thanh, N. H., & Hoßfeld, T. "Serverless computing lifecycle model for edge cloud deployments." *2023 IEEE International Conference on Communications Workshops (ICC Workshops)*. IEEE, 2023.
3. Hilbrich, M., & Lehmann, F. "Discussing microservices: Definitions, pitfalls, and their relations." *2022 IEEE International Conference on Services Computing (SCC)*. IEEE, (2022)
4. Ekwe-Ekwe, N., & Amos, L. "The State of FaaS: An analysis of public Functions-as-a-Service providers." *2024 IEEE 17th International Conference on Cloud Computing (CLOUD)*. IEEE, (2024)
5. Rasheedh, J. A., & Saradha, S. "Reactive microservices architecture using a framework of fault tolerance mechanisms." *2021 second international conference on electronics and sustainable communication systems (ICESC)*. IEEE, (2021)
6. Ugwueze, V. U. "Serverless Computing: Redefining Scalability And Cost Optimization In Cloud Services." (2024)
7. Fernando, C. "Designing Microservices Platforms with NATS." Packt Publishing, (2021)

Publisher: SARC Publisher

8. olec, M., Gill, S. S., Wu, H., Can, T. C., Golec, M., Cetinkaya, O., ... & Uhlig, S. "Master: Machine learning-based cold start latency prediction framework in serverless edge computing environments for industry 4.0." *IEEE Journal of Selected Areas in Sensors* 1 (2024): 36-48.
9. Gilbert, J. "Software Architecture Patterns for Serverless Systems." Packt Publishing, (2021)
10. Megargel, A., Poskitt, C. M., & Shankararaman, V. "Microservices orchestration vs. choreography: A decision framework." *2021 IEEE 25th International Enterprise Distributed Object Computing Conference (EDOC)*. IEEE, (2021)

Source of support: Nil; **Conflict of interest:** Nil.

Cite this article as:
Gunda, S. R. "Microservices and Serverless Computing: Architectural Patterns for Modern Distributed Systems" *Sarcouncil Journal of Engineering and Computer Sciences* 4.8 (2025): pp 199-205.

Publisher: SARC Publisher