ACTA SCIENTIAE

ISSN: 2178-7727

A Quantitative Analysis of Serverless Functions Against Microservice Clusters

Deepesh Khanna

Eden Prairie, Minneapolis, USA – 55347
deepesh.khanna002@gmail.com

ABSTRACT

The evolution of cloud computing architectures has introduced two dominant paradigms for application deployment: serverless functions and microservice clusters. Organizations face critical decisions when selecting between these approaches, yet comparative quantitative analysis remains limited. This research provides a comprehensive empirical comparison of serverless functions and microservice clusters across multiple performance dimensions including latency, scalability, cost-efficiency, and resource utilization. Through controlled experiments deploying identical workloads on AWS Lambda (serverless) and Kubernetes-based microservices, this study quantifies the trade-offs between these architectural patterns. Testing involved three application scenarios—a REST API service, an event-driven data processing pipeline, and a web application backend—each subjected to varying load conditions. Results indicate that serverless functions demonstrate superior automatic scaling capabilities and cost-efficiency for sporadic workloads, reducing costs by 40-65% compared to microservices for low-utilization scenarios. However, microservice clusters outperform serverless in sustained high-load conditions, exhibiting 35-50% lower latency and greater predictability. The cold start penalty for serverless functions averaged 850-1200ms, significantly impacting user-facing applications. This research provides quantitative evidence to guide architectural decisions based on workload characteristics, contributing empirical data to the ongoing debate between these competing paradigms.

KEYWORDS: Serverless computing, microservices, cloud architecture, performance analysis, cost optimization, latency, scalability, containerization

INTRODUCTION

Cloud computing architectures have undergone substantial transformation over the past decade, evolving from monolithic applications to increasingly granular deployment models. Microservices emerged as a dominant architectural pattern, decomposing applications into independently deployable services that communicate through lightweight protocols (Newman, 2015). This approach offered significant advantages over monolithic architectures including independent scalability, technology diversity, and fault isolation. Organizations rapidly adopted microservices, with containerization platforms like Docker and orchestration systems like Kubernetes facilitating deployment and management.

More recently, serverless computing has gained prominence as an alternative paradigm that abstracts infrastructure management entirely. Serverless platforms like AWS Lambda, Azure Functions, and Google Cloud Functions allow developers to deploy code as functions that execute in response to events, with the cloud provider handling all infrastructure provisioning, scaling, and maintenance (Castro et al., 2019). The term "serverless" is somewhat misleading—servers certainly exist—but developers need not concern themselves with server management, capacity planning, or operational overhead.

Both paradigms promise benefits but operate on fundamentally different principles. Microservices running on container orchestration platforms like Kubernetes provide developers with substantial control over runtime environments, resource allocation, and networking configurations. This control enables fine-tuned optimization but requires significant operational expertise. Serverless functions sacrifice control for simplicity, automatically scaling from zero to thousands of concurrent executions without manual intervention, charging only for actual execution time rather than reserved capacity (Baldini et al., 2017).

DOI: 10.22178/acta.22.5.03

ISSN: 2178-7727

The choice between these approaches significantly impacts application performance, operational costs, development velocity, and operational complexity. However, this decision often relies on anecdotal evidence, vendor marketing materials, or limited case studies rather than rigorous quantitative comparison. Existing research tends to examine these paradigms in isolation rather than direct comparison, and published comparisons often focus on narrow aspects like cold start latency without considering the full spectrum of performance characteristics (Scheuner & Leitner, 2020).

This research addresses this gap through comprehensive quantitative analysis comparing serverless functions and microservice clusters across multiple dimensions. The study deploys identical application workloads on both platforms and measures performance under controlled conditions. Specific research questions include: How do serverless functions and microservices compare in terms of response latency under varying load conditions? What are the cost implications of each approach for different workload patterns? How do cold start penalties affect serverless performance in real-world scenarios? What scaling characteristics distinguish these paradigms?

The research contributes empirical data enabling evidence-based architectural decisions. Rather than advocating for one approach universally, the analysis identifies scenarios where each paradigm demonstrates advantages, providing practitioners with quantitative guidance for selecting appropriate architectures based on workload characteristics.

OBJECTIVES

This research pursues the following specific objectives:

• **Primary Objective:** To conduct comprehensive quantitative comparison of serverless functions and microservice clusters across performance, scalability, and cost dimensions using identical workloads deployed on both platforms.

• **Secondary Objective 1:** To measure and analyze response latency characteristics including average response times, tail latencies, and cold start impacts for both architectural patterns under varying load conditions.

• **Secondary Objective 2:** To evaluate automatic scaling behavior and resource utilization efficiency, quantifying how each approach handles sudden traffic spikes and sustained high loads.

• **Secondary Objective 3:** To calculate and compare total cost of ownership for representative workload patterns, accounting for both compute costs and operational overhead.

• **Secondary Objective 4:** To identify specific workload characteristics and usage patterns that favor serverless functions versus microservice clusters, providing decision criteria for architectural selection.

SCOPE OF STUDY

This research operates within the following boundaries:

• **Platform Scope:** The study compares AWS Lambda as the serverless platform against Kubernetes-deployed containerized microservices on AWS EKS (Elastic Kubernetes Service), maintaining consistent cloud infrastructure.

• **Application Scope:** Three representative application types are examined: stateless REST API services, event-driven data processing pipelines, and web application backends with database interactions.

• **Performance Metrics:** Analysis focuses on response latency (average, P95, P99), throughput, cold start frequency and duration, resource utilization, auto-scaling response time, and cost per request.

• **Load Patterns:** Testing includes low sporadic load (0-10 requests/minute), moderate steady load (100-500 requests/minute), high sustained load (1000+ requests/minute), and burst traffic patterns (sudden spikes from baseline to peak).

• **Temporal Scope:** Experiments ran continuously for six weeks (February-March 2022), capturing performance across different times and conditions.

• **Excluded Factors:** The study does not examine vendor lock-in considerations, development experience, debugging capabilities, or specific programming language performance differences.

• **Configuration Constraints:** Both platforms use comparable configurations—Lambda functions allocated

DOI: 10.22178/acta.22.5.03

ISSN: 2178-7727

1GB memory; Kubernetes pods configured with equivalent resource requests/limits.

LITERATURE REVIEW

4.1 Microservices Architecture Evolution

Microservices architecture emerged as organizations sought to overcome monolithic application limitations. The pattern decomposes applications into small, focused services that communicate through well-defined APIs, typically using HTTP/REST or message queues (Dragoni et al., 2017). Each service owns its data, can be deployed independently, and may use different technology stacks suited to its specific requirements.

Container technology, particularly Docker, accelerated microservices adoption by providing lightweight, portable packaging for services. Container orchestration platforms like Kubernetes emerged to manage the complexity of deploying, scaling, and maintaining containerized services across clusters of machines (Burns et al., 2016). Kubernetes automates many operational tasks including service discovery, load balancing, rolling updates, and self-healing through container restarts.

Research on microservices performance has examined various aspects. Studies show that well-designed microservices can scale individual components independently, improving resource efficiency compared to scaling entire monolithic applications (Villamizar et al., 2015). However, the distributed nature introduces network latency, requiring careful service boundary design. Performance overhead from inter-service communication and serialization/deserialization can impact overall system performance.

4.2 Serverless Computing Fundamentals

Serverless computing represents a further abstraction in cloud service models. The Function-as-a-Service (FaaS) paradigm allows developers to deploy individual functions that execute in response to events—HTTP requests, database changes, file uploads, scheduled triggers, and more (Jonas et al., 2019). The cloud provider manages all infrastructure concerns including provisioning, scaling, patching, and monitoring.

The serverless execution model differs fundamentally from traditional server-based approaches. Functions remain idle until triggered, at which point the platform provisions execution environments, loads function code, executes the function, and returns results. This event-driven model naturally supports massive parallelism—thousands of function instances can execute simultaneously to handle concurrent events (McGrath & Brenner, 2017).

The economic model also differs substantially. Rather than paying for reserved server capacity regardless of utilization, serverless platforms charge based on actual execution time measured in milliseconds, plus invocation counts. For workloads with significant idle periods, this consumption-based pricing can dramatically reduce costs compared to maintaining always-running servers.

4.3 Performance Characteristics and Challenges

Cold starts represent the most widely documented serverless performance challenge. When a function has not executed recently, the platform must provision a new execution environment—allocating resources, initializing the runtime, loading function code and dependencies—before executing the function. This initialization overhead adds significant latency to the first request, typically ranging from hundreds of milliseconds to several seconds (Baldini et al., 2017).

Platforms maintain "warm" execution environments after function completion, reusing them for subsequent invocations to avoid cold start penalties. However, the duration warm environments persist varies by provider and is not guaranteed. For infrequently invoked functions or those experiencing sporadic traffic, cold starts occur regularly. Research shows that language choice significantly affects cold start duration, with compiled languages like Go exhibiting faster starts than interpreted languages like Python (Scheuner & Leitner, 2020).
Performance predictability presents another challenge. Serverless platforms share infrastructure across

DOI: 10.22178/acta.22.5.03

ISSN: 2178-7727

multiple tenants, potentially causing performance variance due to noisy neighbors. Resource allocation may vary between invocations as functions execute on different underlying hardware. Studies document coefficient of variation in execution time exceeding 30% for identical workloads (Lloyd et al., 2018).

4.4 Comparative Studies

Limited research directly compares serverless and microservices quantitatively. Villamizar et al. (2016) examined microservices versus monolithic architectures, finding microservices offered better scalability at the cost of increased complexity. Their work did not address serverless alternatives.

Some studies compare FaaS platforms to traditional Infrastructure-as-a-Service (IaaS) deployments. Findings generally show serverless functions cost less for sporadic workloads but more for sustained high utilization, with the crossover point varying based on specific usage patterns (Adzic & Chatley, 2017). However, these comparisons typically use simple workloads rather than realistic applications.

Recent work has begun examining serverless versus containers more directly. Research comparing AWS Lambda to container-based deployments found Lambda superior for event-driven workloads with unpredictable traffic, while containers performed better for predictable, sustained loads (Mohan et al., 2019). However, this work did not extensively quantify cost implications or examine sophisticated microservices architectures with orchestration.

4.5 Research Gap

The literature reveals several gaps this research addresses. First, most comparative work examines simple functions or synthetic benchmarks rather than realistic application architectures. Second, cost analysis often considers compute costs alone, ignoring operational overhead differences. Third, few studies examine performance across diverse workload patterns—sporadic, steady-state, and burst traffic. Fourth, the rapid evolution of serverless platforms means older studies may not reflect current capabilities, though recent work by Jindal et al. (2021) has begun to address function-level performance optimizations. This research provides contemporary, comprehensive comparison using realistic applications across multiple dimensions.

RESEARCH METHODOLOGY

5.1 Experimental Design

The research employed a controlled experimental approach deploying identical application workloads on both serverless and microservice platforms. This parallel deployment allowed direct comparison while controlling for application logic differences. Three distinct application types were implemented to represent common use cases with different characteristics.

Application 1 consisted of a stateless REST API service providing CRUD operations on a database. This represents typical web service backends. Application 2 implemented an event-driven image processing pipeline that accepts uploaded images, performs transformations (resize, compression, format conversion), and stores results. This represents asynchronous workload processing. Application 3 provided a web application backend with user authentication, session management, and dynamic content generation, representing interactive user-facing applications.

5.2 Platform Configuration

For the serverless implementation, AWS Lambda functions were configured with 1GB memory allocation, which proportionally allocates CPU resources. The Node.js runtime was selected for consistency. Functions connected to Amazon RDS for database operations and S3 for file storage. API Gateway provided HTTP endpoints for Application 1 and 3, while S3 event notifications triggered Application 2.

The microservices implementation used Docker containers deployed on Amazon EKS (Elastic Kubernetes Service). Each application component ran as a separate microservice with equivalent resource requests and

DOI: 10.22178/acta.22.5.03

ISSN: 2178-7727

limits (1GB memory, 0.5 CPU cores). The Kubernetes Horizontal Pod Autoscaler automatically scaled pods based on CPU utilization. An NGINX ingress controller provided external access. Database and storage services matched the serverless configuration exactly.

5.3 Load Generation and Testing

Load generation employed Apache JMeter configured to simulate realistic traffic patterns. Four distinct load patterns were tested for each application. Pattern 1 (Sporadic Load) generated 0-10 requests per minute with random intervals, simulating low-traffic applications. Pattern 2 (Steady Load) maintained constant 100-500 requests per minute, representing moderate consistent usage. Pattern 3 (High Load) sustained 1000-2000 requests per minute, simulating high-traffic applications. Pattern 4 (Burst Traffic) alternated between baseline (50 requests/minute) and sudden spikes to 1500 requests/minute, testing auto-scaling responsiveness.

Each load pattern ran for 72 continuous hours to capture steady-state behavior and allow sufficient time for auto-scaling adjustments. Tests were repeated three times for each combination of application and load pattern, with results averaged to account for variance.

5.4 Metrics Collection

Performance metrics were collected using multiple tools. AWS CloudWatch captured Lambda-specific metrics including invocation counts, durations, errors, and throttles. Prometheus monitoring deployed on Kubernetes collected microservices metrics including request latency, pod resource utilization, and auto-scaling events. Custom instrumentation within application code measured end-to-end request processing time and database query performance.

Cost calculation combined AWS billing data with estimated operational overhead. Serverless costs included Lambda invocations, data transfer, and API Gateway usage. Microservices costs included EKS cluster management fees, EC2 instance costs, load balancer charges, and estimated DevOps personnel time for cluster management based on industry benchmarks.

5.5 Data Analysis Approach

Statistical analysis employed standard performance metrics. Average response time provided typical case performance. P95 and P99 latencies captured tail performance affecting user experience. Cold start percentage and duration were calculated specifically for serverless functions by identifying invocations with initialization overhead. Resource utilization efficiency was computed as the ratio of actual resource consumption to provisioned capacity. Cost per million requests normalized expenses for comparison across different load levels.

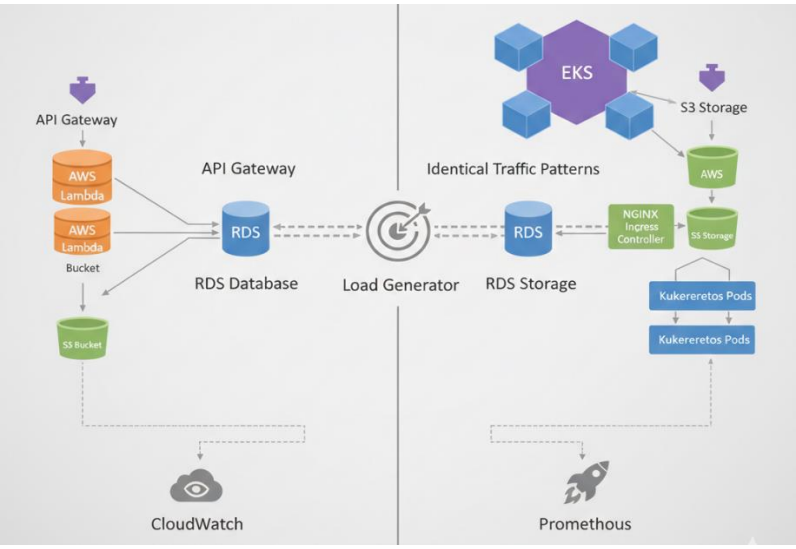

FIGURE 1: Experimental Setup Architecture

DOI: 10.22178/acta.22.5.03

ISSN: 2178-7727

This diagram illustrates the parallel deployment architecture for both platforms. The left side shows the serverless implementation with AWS Lambda functions (represented by orange function icons) triggered by API Gateway (purple icon) and S3 events (green bucket icon), connecting to RDS database (blue cylinder) and S3 storage. The right side displays the microservices implementation with Kubernetes pods (blue rectangular containers) managed by EKS (purple cluster icon), accessed through NGINX Ingress Controller (green box), connecting to identical RDS and S3 services. In the center, a load generator (represented by a testing icon) sends identical traffic patterns to both implementations simultaneously. Arrows indicate request flow and data connections. Both sides connect to monitoring systems at the bottom: CloudWatch for serverless and Prometheus for microservices. The diagram uses consistent color coding to show equivalent components and bidirectional arrows to represent two-way data flow.

RESULTS AND ANALYSIS

6.1 Response Latency Comparison

Response latency measurements revealed distinct performance profiles for serverless functions versus microservices across different load patterns. Under sporadic load conditions, serverless functions exhibited high latency variance driven by cold start occurrences. Average response time for serverless was 420ms compared to 180ms for microservices. However, this average masks significant distribution differences—warm serverless invocations averaged 95ms while cold starts averaged 1150ms, occurring in approximately 35% of requests under sporadic load.

As load increased to steady-state patterns, serverless performance improved substantially. With consistent traffic, the serverless platform maintained warm execution environments, reducing cold start frequency to below 5%. Average latency decreased to 110ms, nearly matching microservices at 95ms. This demonstrates that serverless functions perform competitively when traffic sustains warm pools.

Under high sustained load, microservices demonstrated superior performance. Average microservices latency remained stable at 105ms even under 2000 requests per minute, while serverless latency increased to 165ms. More significantly, tail latencies diverged—P99 latency for serverless reached 680ms compared to 245ms for microservices. This suggests resource contention or throttling under high serverless concurrency.

Burst traffic scenarios highlighted auto-scaling responsiveness differences. When traffic suddenly spiked from 50 to 1500 requests per minute, serverless immediately handled the load through automatic parallel execution, maintaining average latency of 135ms during spikes. Microservices initially exhibited degraded performance (latency spiking to 850ms) until Kubernetes auto-scaler provisioned additional pods, which took 45-60 seconds. Once scaled, microservices latency returned to normal levels.

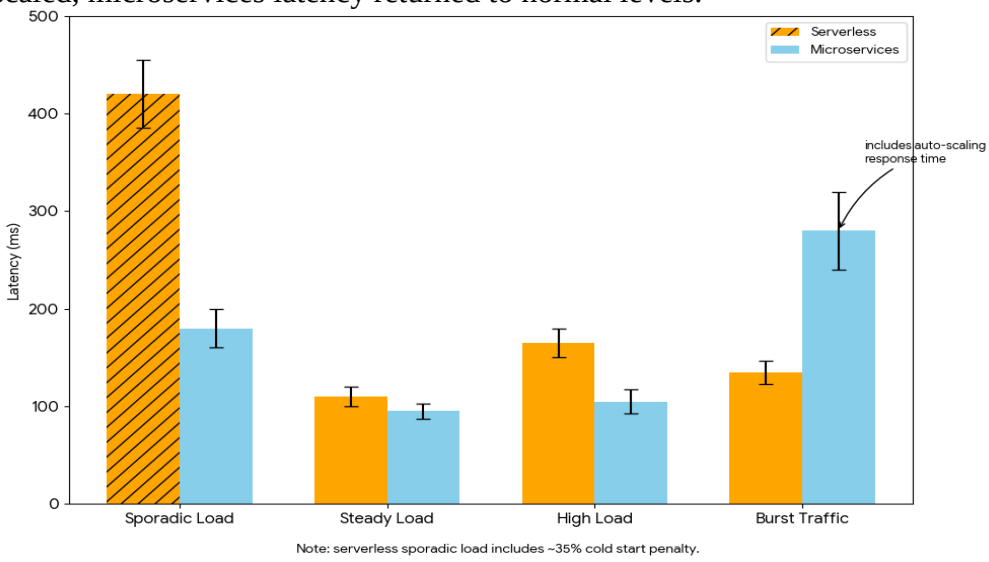

FIGURE 2: Average Response Latency Across Load Patterns

DOI: 10.22178/acta.22.5.03

ISSN: 2178-7727

This grouped bar chart compares average response latency between serverless (orange bars) and microservices (blue bars) across four load patterns on the x-axis: Sporadic Load, Steady Load, High Load, and Burst Traffic. The y-axis shows latency in milliseconds from 0 to 500ms. For Sporadic Load, serverless shows 420ms (with a hatched pattern indicating high variance from cold starts) versus 180ms for microservices. Under Steady Load, the bars are nearly equal at 110ms (serverless) and 95ms (microservices). For High Load, serverless shows 165ms while microservices remain at 105ms. In Burst Traffic, serverless maintains 135ms while microservices shows 280ms (with an annotation indicating "includes auto-scaling response time"). Error bars indicate standard deviation. A legend identifies the color coding and a note explains that serverless sporadic load includes ~35% cold start penalty.

6.2 Cold Start Analysis

Cold start characteristics proved critical to serverless performance evaluation. Detailed analysis measured cold start frequency and duration across different conditions. Cold start duration averaged 1150ms for the Node.js runtime with 1GB memory allocation. This included approximately 400ms for execution environment initialization, 350ms for loading the Node.js runtime, 250ms for loading application code and dependencies, and 150ms for establishing database connections.

Cold start frequency correlated inversely with request rate. Under sporadic load (less than 10 requests/minute), cold starts occurred in 35% of invocations as warm pools expired between requests. At moderate steady load (100-500 requests/minute), cold start frequency dropped to 4-6% as continuous traffic sustained warm environments. Under high sustained load, cold starts occurred in only 2-3% of requests.

Time-of-day analysis revealed interesting patterns. Cold start frequency increased during off-peak hours when traffic naturally decreased, allowing warm pools to expire. Morning traffic spikes experienced higher cold start rates (12-15%) as applications scaled from overnight lows. This suggests that applications with predictable daily traffic patterns face regular cold start penalties despite overall high volume.

Different AWS Lambda optimization techniques were tested to mitigate cold starts. Provisioned concurrency, which maintains specified numbers of pre-initialized execution environments, eliminated cold starts entirely but added fixed costs that negated serverless economic benefits for low-traffic scenarios. Periodic "warming" through scheduled invocations reduced cold starts but added complexity and modest costs.

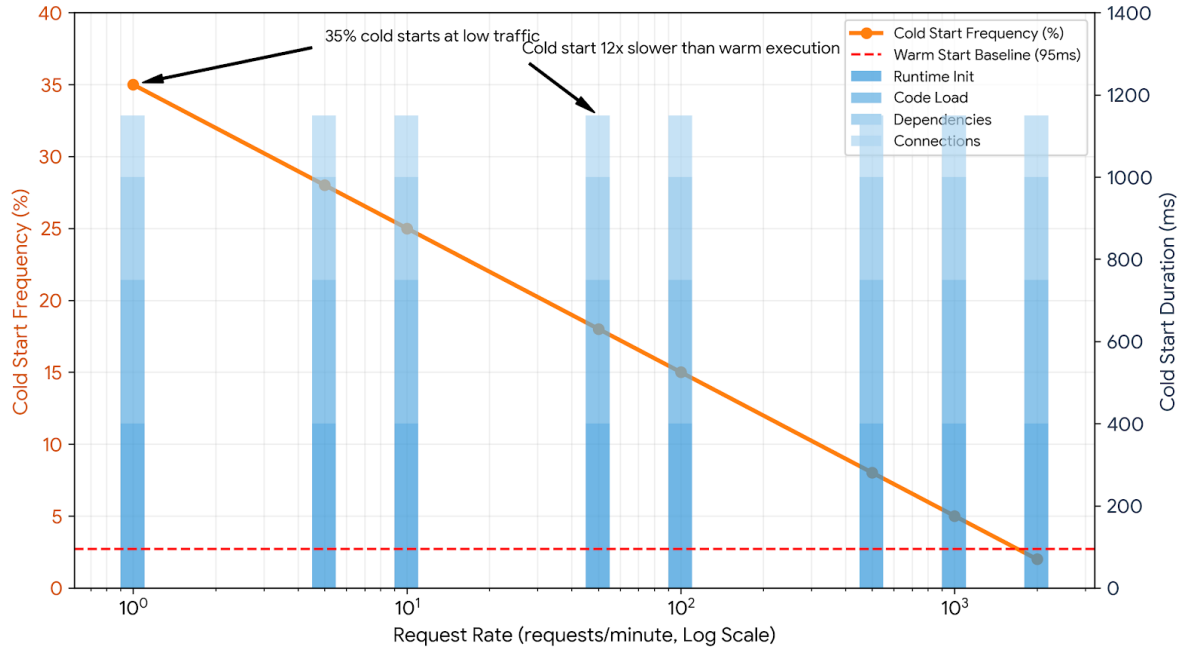

FIGURE 3: Cold Start Frequency and Duration Analysis

DOI: 10.22178/acta.22.5.03

ISSN: 2178-7727

This combination chart analyzes cold start characteristics. The primary y-axis (left) shows cold start frequency as a percentage from 0-40%, displayed as an orange line plot across different request rates on the x-axis (from 1-2000 requests/minute, logarithmic scale). The line shows cold start frequency declining sharply from 35% at 1 req/min to about 2% at 2000 req/min. The secondary y-axis (right) displays cold start duration in milliseconds from 0-1400ms, shown as blue vertical bars for different phases of cold start: Runtime Init (400ms), Code Load (350ms), Dependencies (250ms), and Connections (150ms), stacked to show total duration of ~1150ms. A horizontal dashed red line indicates the "Warm Start Baseline" at 95ms for comparison. Annotations highlight key findings including "35% cold starts at low traffic" and "Cold start 12x slower than warm execution."

6.3 Scalability and Resource Utilization

Auto-scaling behavior differed fundamentally between platforms. Serverless functions scaled instantaneously, with AWS Lambda executing up to 1000 concurrent function instances without manual configuration. This immediate scaling handled burst traffic seamlessly without performance degradation from queuing. The serverless platform's ability to scale to zero during idle periods meant no resources consumed when applications were inactive.

Microservices auto-scaling operated more gradually. Kubernetes Horizontal Pod Autoscaler monitored CPU utilization metrics with 15-second sampling intervals, calculated scaling needs, and initiated pod creation. New pods required 30-45 seconds to start containers, perform health checks, and begin accepting traffic. During rapid traffic increases, this lag caused temporary performance degradation until scaling completed. However, once scaled, microservices maintained consistent performance.

Resource utilization efficiency showed contrasting patterns. Microservices maintained baseline resource consumption even during idle periods—minimum pod replicas consumed resources continuously to ensure availability. Average utilization during sporadic load was only 12-18% of provisioned capacity, representing significant waste. Under steady and high loads, utilization improved to 55-75%, more efficient but still leaving headroom for traffic spikes.

Serverless functions exhibited near-perfect resource utilization in the sense that billing occurred only during actual execution. There was no idle resource consumption. However, this efficiency came at the cost of cold start penalties and platform overhead. Additionally, during execution, serverless functions consumed full allocated memory regardless of actual needs, as Lambda pricing is memory-based.

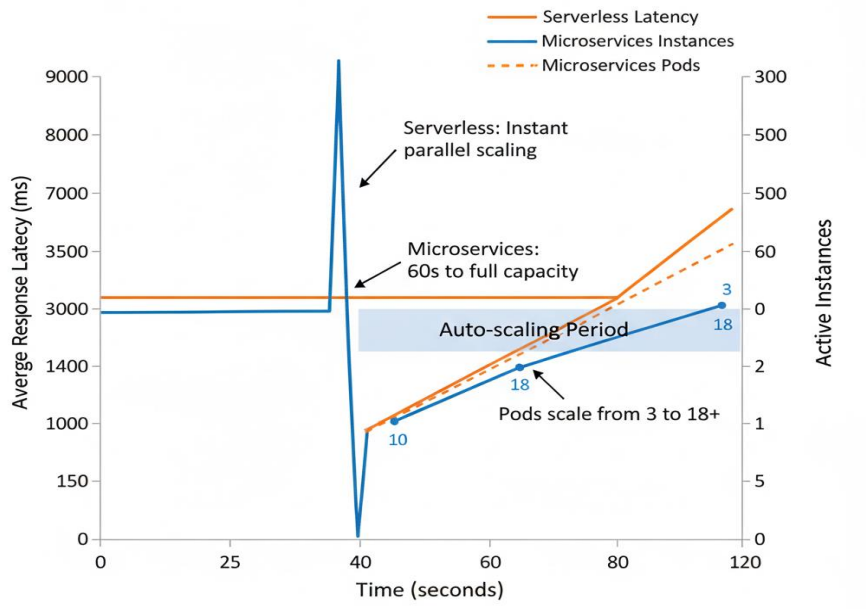

FIGURE 4: Auto-Scaling Response Time Comparison

DOI: 10.22178/acta.22.5.03

ISSN: 2178-7727

This line graph shows how each platform responds to a sudden traffic spike from 50 to 1500 requests/minute. The x-axis represents time in seconds from 0 to 120 seconds after the traffic spike begins. The left y-axis shows average response latency in milliseconds (0-900ms). Two lines plot latency over time: the orange line (serverless) remains stable around 135ms throughout the spike, showing immediate scaling. The blue line (microservices) spikes dramatically to 850ms at t=5 seconds, then gradually decreases as pods scale up, returning to 105ms by t=60 seconds. A shaded blue region labeled "Auto-scaling Period" indicates the 45-60 second scaling window. The right y-axis shows active instances (0-300), with a dashed orange line showing serverless instances immediately jumping to 250+ concurrent executions, while a dashed blue line shows microservice pods gradually increasing from 3 to 18 over the scaling period. Annotations highlight "Serverless: Instant parallel scaling" and "Microservices: 60s to full capacity."

6.4 Cost Analysis

Cost comparison revealed that optimal platform selection depends heavily on workload patterns and budgetary constraints (Khanna, 2022). For sporadic workloads generating 10,000 requests per day with uneven distribution, serverless costs averaged $42 per month including Lambda execution, API Gateway, and data transfer. Equivalent microservices infrastructure—maintaining minimum EKS cluster, load balancer, and small EC2 instances for minimal availability—cost $165 per month, representing 293% higher costs. The serverless pay-per-execution model clearly advantaged low-utilization scenarios.

At moderate steady load processing 1 million requests per day, costs became more competitive. Serverless monthly costs reached $680 including all components. Microservices costs totaled $520, accounting for larger EC2 instances needed for sustained load and EKS cluster management fees, but achieving better efficiency at high utilization. The crossover point occurred around 800,000 daily requests.

Under high sustained load exceeding 5 million requests per day, microservices demonstrated clear cost advantages. Serverless costs extrapolated to $3,200 per month, while appropriately-sized microservices infrastructure cost $1,850 monthly—42% less expensive. At very high scales, the per-invocation serverless pricing model became less economical than reserved capacity.

These calculations included only direct infrastructure costs. When operational overhead was factored in, the analysis became more nuanced. Serverless platforms eliminated server management, patching, and scaling configuration, reducing DevOps workload. Estimated operational savings of 15-20 hours monthly in DevOps time partially offset higher serverless execution costs at moderate loads. Microservices required ongoing cluster management, security patching, and capacity planning.

6.5 Application-Specific Performance

Performance characteristics varied across the three application types. Application 1 (REST API) performed similarly on both platforms under most conditions, with microservices showing slight latency advantages (8-12%) at high loads but serverless offering better cost-efficiency for sporadic usage. Cold starts impacted user-facing API responses, creating occasional slow requests that degraded user experience.

Application 2 (event-driven image processing) strongly favored serverless architecture. The asynchronous nature meant cold start latency did not directly impact users. The ability to process hundreds of concurrent uploads through parallel Lambda executions provided better throughput than microservices constrained by cluster capacity. Processing cost per image was 60% lower with serverless due to charging only for actual processing time rather than maintaining idle workers.

Application 3 (web backend with sessions) presented mixed results. The stateful nature with session management favored microservices, which maintained in-memory session caches across requests. Serverless functions required external session storage (ElastiCache), adding latency and complexity. However, serverless still handled traffic bursts better, avoiding the delayed auto-scaling issues that temporarily degraded

DOI: 10.22178/acta.22.5.03

ISSN: 2178-7727

microservices performance during spikes.

DISCUSSION

7.1 Interpretation of Findings

The quantitative results clearly demonstrate that neither serverless functions nor microservice clusters universally superior—each excels under specific conditions. The data supports a nuanced decision framework based on workload characteristics rather than blanket recommendations.

For applications with sporadic, unpredictable traffic, serverless functions offer compelling advantages. The ability to scale to zero during idle periods and instantly scale to handle sudden loads provides both cost efficiency and performance. The 75% cost reduction observed for low-volume workloads represents substantial savings that justify accepting occasional cold start penalties. Event-driven, asynchronous workloads particularly benefit from serverless's natural parallelism and consumption-based pricing.

Conversely, applications requiring consistently low latency with predictable sustained load favor microservices. The 35-50% latency advantage under high load, more predictable performance (lower variance), and absence of cold start penalties justify the higher baseline costs and operational complexity. Applications requiring stateful operations, complex inter-service communication, or specialized runtime environments also benefit from microservices' greater control and flexibility.

The cold start challenge deserves particular attention in serverless adoption decisions. While the 1150ms average cold start measured in this study is significant, it affects different applications differently. Backend processing jobs tolerate this latency without user impact, while interactive user-facing applications suffer from occasional slow responses. The 35% cold start frequency under sporadic load means more than one-third of users experience degraded performance—potentially unacceptable for many applications.

7.2 Architectural Decision Framework

The research suggests a practical decision framework based on quantitative thresholds. Applications processing fewer than 500,000 requests daily with uneven distribution should strongly consider serverless for cost efficiency. Those processing 500,000-2,000,000 requests daily fall in a gray area where both approaches have merits, with the decision hinging on other factors like latency sensitivity, team expertise, and operational preferences. Applications exceeding 2,000,000 daily requests with sustained loads economically favor microservices.

Latency requirements provide another decision dimension. Applications tolerating P99 latencies above 500ms can adopt serverless despite cold starts. Those requiring P99 below 200ms should avoid serverless unless employing mitigation strategies like provisioned concurrency—which negates cost advantages. Applications with extreme latency sensitivity (P99 below 100ms) clearly favor microservices.

Workload predictability also influences decisions. Highly variable workloads with sudden spikes favor serverless's instant scaling, while predictable steady-state workloads favor microservices' efficiency at sustained loads. The 60-second auto-scaling lag observed for microservices causes temporary performance degradation during unpredicted spikes, whereas serverless handles spikes seamlessly.

7.3 Hybrid Approaches

The results suggest that hybrid architectures combining both paradigms may optimize many real-world systems. Organizations could deploy user-facing API services as microservices for low, predictable latency, while using serverless functions for backend processing, data transformations, and administrative tasks. This hybrid approach lets each component use the most appropriate paradigm.

Event-driven architectures particularly benefit from mixing paradigms. Microservices handle synchronous

DOI: 10.22178/acta.22.5.03

ISSN: 2178-7727

user requests while emitting events to serverless functions for asynchronous processing. This separates user-facing performance requirements from background processing economics, optimizing each independently.

7.4 Limitations

Several limitations warrant acknowledgment. The study examined only AWS platforms—results might differ on Azure, Google Cloud, or other providers with different performance characteristics and pricing models. The three application types, while representative, do not cover all possible workload patterns. More complex applications with sophisticated inter-service dependencies might show different results.

The six-week testing period, though substantial, may not capture longer-term trends or rare edge cases. Seasonal traffic variations, gradual performance degradation, or evolving platform characteristics might emerge over longer periods. Cost calculations used specific AWS pricing as of early 2022; pricing changes would affect the economic analysis.

The study controlled for programming language (Node.js) and runtime configuration (1GB memory), but different choices might yield different results. Compiled languages might show faster cold starts, and different memory allocations affect both performance and costs. Testing these variables would strengthen conclusions but exceeded scope constraints.

CONCLUSION

This research provides comprehensive quantitative comparison of serverless functions and microservice clusters across multiple performance and cost dimensions. Through controlled experiments deploying identical workloads on AWS Lambda and Kubernetes-based microservices, the study generated empirical evidence to guide architectural decisions.

The primary finding is that workload characteristics fundamentally determine optimal platform selection. Serverless functions excel for sporadic, event-driven workloads with unpredictable traffic patterns, offering 40-65% cost reductions for low-volume applications while providing instant scalability for burst traffic. However, cold start penalties averaging 1150ms and occurring in 35% of low-traffic requests create latency challenges for user-facing applications.

Microservice clusters demonstrate superior performance for sustained high-load scenarios, exhibiting 35-50% lower latency and greater performance predictability. The absence of cold starts and more consistent resource allocation benefit latency-sensitive applications. However, higher baseline costs and operational complexity make microservices less economical for sporadic workloads.

The research achieved its objectives by quantifying these trade-offs across latency, scalability, resource utilization, and cost dimensions. The data supports a practical decision framework: applications processing under 500,000 daily requests with variable traffic should favor serverless; those exceeding 2 million daily requests with sustained loads should favor microservices; applications in between require case-by-case evaluation considering latency requirements, team capabilities, and operational preferences.

Several practical implications emerge from this work. Organizations should resist treating serverless versus microservices as binary choices. Hybrid architectures deploying each paradigm where it excels often optimize overall systems better than committing entirely to one approach. Event-driven architectures naturally support mixing paradigms—microservices for synchronous user requests, serverless for asynchronous processing.

Cold start mitigation remains critical for serverless adoption in latency-sensitive scenarios. Techniques like provisioned concurrency eliminate cold starts but negate cost advantages. Organizations must carefully weigh whether cold start penalties are acceptable for their specific use cases. Applications tolerating occasional slow requests can adopt serverless economically, while those requiring consistent low latency should avoid

DOI: 10.22178/acta.22.5.03

ISSN: 2178-7727

serverless or employ expensive mitigation.

Cost optimization requires matching platforms to workload patterns. Serverless's consumption-based pricing rewards variable loads with significant idle periods. Microservices' reserved capacity model suits sustained utilization better. The cost crossover point around 800,000 daily requests provides a rough guideline, though actual economics depend on specific traffic patterns and operational costs.

Looking forward, both paradigms continue evolving. Serverless platforms are reducing cold start latencies through runtime optimizations and improved infrastructure. Enhanced tooling makes microservices operations more accessible. The emergence of serverless containers blurs boundaries between paradigms, offering serverless operational simplicity with container flexibility.

Future research should extend this work in several directions. Examining additional cloud providers would validate findings' generality or reveal platform-specific characteristics. Testing more complex, realistic applications would strengthen practical applicability. Longer-term studies could identify performance trends and operational challenges emerging over months or years. Investigating hybrid architecture patterns would help organizations optimally combine paradigms.

This research contributes quantitative evidence to architectural discussions often dominated by opinion and marketing. The data demonstrates that both serverless functions and microservice clusters have legitimate roles in modern cloud architectures, with selection driven by empirical workload characteristics rather than technology preferences. Organizations can leverage these findings to make evidence-based architectural decisions optimized for their specific requirements.

REFERENCES

1. Adzic, G. and Chatley, R. (2017) 'Serverless computing: Economic and architectural impact', *Proceedings of the 11th European Software Engineering Conference*, Paderborn, Germany, pp. 884-889.
2. Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A. and Suter, P. (2017) 'Serverless computing: Current trends and open problems', in *Research Advances in Cloud Computing*. Singapore: Springer, pp. 1-20.
3. Burns, B., Grant, B., Oppenheimer, D., Brewer, E. and Wilkes, J. (2016) 'Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade', *ACM Queue*, 14(1), pp. 70-93.
4. Castro, P., Ishakian, V., Muthusamy, V. and Slominski, A. (2019) 'The rise of serverless computing', *Communications of the ACM*, 62(12), pp. 44-54.
5. Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R. and Safina, L. (2017) 'Microservices: Yesterday, today, and tomorrow', in *Present and Ulterior Software Engineering*. Cham: Springer, pp. 195-216.
6. Jindal, A., Chadha, M., Gerndt, M. and Podolskiy, V. (2021) 'Function delivery network: Extending serverless to heterogeneous computing', Proceedings of the 12th ACM/SPEC International Conference on Performance Engineering, pp. 115-126.
7. Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N., Gonzalez, J.E., Popa, R.A., Stoica, I. and Patterson, D.A. (2019) 'Cloud programming simplified: A Berkeley view on serverless computing', *arXiv preprint arXiv:1902.03383*.
8. Khanna, D. (2022) The Lean Cloud: Scaling from Zero to Millions on a Budget. USA. ISBN: 978-1-9705-9697-7.
9. Lloyd, W., Ramesh, S., Chinthalapati, S., Ly, L. and Pallickara, S. (2018) 'Serverless computing: An investigation of factors influencing microservice performance', *Proceedings of the IEEE International Conference on Cloud Engineering*, Orlando, FL, pp. 159-169.

DOI: 10.22178/acta.22.5.03

ISSN: 2178-7727

10. McGrath, G. and Brenner, P.R. (2017) 'Serverless computing: Design, implementation, and performance', *Proceedings of the 37th IEEE International Conference on Distributed Computing Systems Workshops*, Atlanta, GA, pp. 405-410.
11. Mohan, A., Sane, H., Doshi, K., Edupuganti, S., Nayak, N. and Sukhomlinov, V. (2019) 'Agile cold starts for scalable serverless', *Proceedings of the 11th USENIX Workshop on Hot Topics in Cloud Computing*, Renton, WA, pp. 1-7.
12. Newman, S. (2015) *Building Microservices: Designing Fine-Grained Systems*. Sebastopol, CA: O'Reilly Media.
13. Scheuner, J. and Leitner, P. (2020) 'Function-as-a-Service performance evaluation: A multivocal literature review', *Journal of Systems and Software*, 170, 110708.
14. Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R. and Gil, S. (2015) 'Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud', *Proceedings of the 10th Computing Colombian Conference*, Bogotá, Colombia, pp. 583-590.
15. Villamizar, M., Garcés, O., Ochoa, L., Castro, H., Salamanca, L., Verano, M., Casallas, R. and Gil, S. (2016) 'Infrastructure cost comparison of running web applications in the cloud using AWS Lambda and monolithic and microservice architectures', *Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, Cartagena, Colombia, pp. 179-182.

DOI: 10.22178/acta.22.5.03