The American Journal of Engineering and Technology
ISSN 2689-0984 | Open Access

TYPE Original Research
PAGE NO. 90-98
DOI 10.37547/tajet/Volume07Issue10-12

Check for updates

Architectural Approaches to Scaling Distributed Microservice Systems in The Cloud

Kumar Avinash
Software Development Engineer, Google Seattle, USA

OPEN ACCESS

SUBMITTED 22 August 2025
ACCEPTED 30 September 2025
PUBLISHED 23 October 2025
VOLUME Vol.07 Issue 10 2025

CITATION
Kumar Avinash. (2025). Architectural Approaches to Scaling Distributed Microservice Systems in The Cloud. The American Journal of Engineering and Technology, 7(10), 90–98.
https://doi.org/10.37547/tajet/Volume07Issue10-12

Abstract: This article explores how distributed microservice systems in the cloud are scaled. The goal is to piece together, from scattered research, working solutions and what might still be experimental. The focus is on design and deployment strategies that aim for reliability, elasticity, and cost-effective growth. Open-access, peer-reviewed papers published since 2021 were reviewed with a special emphasis placed on those with empirical tests, diagrams, or case studies to see how ideas play out in practice. Across the papers certain themes keep reappearing. Infrastructure work revolves around horizontal and vertical scaling, while orchestration and autoscaling — Kubernetes HPA/VPA, serverless computing, various service meshes — are interpreted as part of the broader field. The unique contribution is the attempt to frame old and new together — to see classical ideas such as service modularity alongside hybrid autoscalers, energy-aware verification, probabilistic checks. In some cases, these tactics reinforce one another; in others they collide, or create unexpected trade-offs. Observing them in the same frame highlights that scaling microservices is as an ongoing experiment, where established patterns and cutting-edge proposals coexist. Taken together, the available studies do not present a single blueprint but instead sketch what looks like a layered approach. In many cases services are kept as stateless and decoupled as possible; in others that ideal is only partly met. Horizontal and vertical scaling appear side by side — sometimes combined in the same system — with their usefulness varying by workload. Orchestration policies are often automated, yet the literature also warns that energy use and security concerns grow quietly in the

background as systems expand and, if left unmanaged, can erode the gains of scaling. This article will provide value to cloud architects, DevOps engineers, developers, and researchers interested in gaining awareness of current practices as well as possible venues of where the field might be heading next.

Keywords: Microservices, Cloud computing, Horizontal scaling, Vertical scaling, Kubernetes, Autoscaling, Serverless computing, Reinforcement learning, Architectural design, Scalability.

Introduction

Microservices architecture is often described as breaking an application into a set of smaller, loosely connected services—each running on its own, talking through APIs. Advocates of microservice architectures usually mention higher availability, clearer fault isolation and, perhaps most often, the promise of horizontal scalability [3]. Breaking an application into distinct components can allow one area to be scaled separately from the rest. But the effect is uneven. In some reported cases the gains are striking; in others they are marginal or even negative [9]. A likely reason is that modularity opens up boundaries where scaling can occur independently, but the benefit depends on how much cross-service communication, caching and shared state exist. In other words, the same architectural feature that enables independent scaling can also create extra latency or coordination costs if the services are tightly coupled in practice.

Cloud-native firms—Netflix is the primary example—report major gains after moving from monolithic architectures to microservices. They can add servers or containers just for the overworked service rather than for the entire application. This may improve overall capacity and performance under heavy load. Unfortunately, such outcomes are not guaranteed. Smaller or more stable systems, for instance, might see little advantage. Some empirical work even suggests that on a single server a monolith can outperform an equivalent microservice arrangement [3]. So the apparent superiority of microservices could be context-dependent—true at massive scale but not at the small or medium range.

Microservices might be inherently well suited to cloud environments, where on-demand resources and autoscaling tools abound. Yet even in this case, scaling is not automatic. Running dozens—or hundreds—of independent services introduces new challenges: service discovery, inter-service latency, and data consistency among them. This could erode the very gains that microservices promise. In practice, architectural patterns and operational discipline seem to matter as much as the architecture itself. Researchers and practitioners therefore discuss stateless service design, careful API boundaries, orchestration layers, and increasingly, algorithmic resource management [4; 6]. These measures may, but do not always, ensure that microservice systems grow smoothly while maintaining performance.

This mixed record suggests a field still lacks consensus. The literature from 2021 onward shows a lively search for scalable designs and management strategies—some quite technical, others organizational. It is probably appropriate to suggest there is no single solution; rather, scaling microservices involves a collection of overlapping practices, tested under different conditions, each with its own trade-offs.

Methods and Materials

This article uses a literature-based approach to explore how architects and researchers are trying to scale microservices in the cloud. The focus is on recent work, mostly since 2021, but the boundary is porous; some earlier ideas still echo through the newer papers. The studies sampled range widely: basic design principles, performance comparisons between microservice and monolithic systems, autoscaling algorithms, and assorted cloud-native technologies. The first set of themes includes service statelessness, loose coupling, clean API boundaries. Yet these supposedly “fundamental” patterns don’t always work as advertised; some studies hint at unexpected bottlenecks or coordination costs. After that come more infrastructural questions—horizontal versus vertical scaling, container orchestration, and the sometimes messy business of distributing loads across replicas. Kubernetes appears everywhere in the literature, but opinions diverge on whether its autoscaling features are enough or need to be augmented by hybrid or machine-learning–driven methods. Serverless computing crops up as a suitable alternative in some cases, though critics note it may only shift complexity elsewhere.

Alharthi et al. sketch the landscape of autoscaling in cloud computing, teasing apart reactive and proactive strategies and hinting at unresolved research problems [1]. Berardi et al. dig into microservice security—spoofing, denial-of-service—and in doing so indirectly

show how scaling and security intertwine [2]. Blinowski et al. run empirical tests comparing monoliths and microservices under various workloads, and the results are uneven: sometimes microservices win, sometimes not [3]. Chavan explores how scaling can drive costs up and looks to FinOps to rein things back in [4]. Domakonda sets out design principles for secure and scalable systems but also notes how complicated they can be to achieve in practice [5]. Filippone et al. experiment with choreography-based microservice systems to improve coordination—another route to scalability [6]. Jawaddi et al. use probabilistic model checking to handle energy-efficient autoscaling on Kubernetes, suggesting yet another trade-off space [7]. Pandiya outlines core scalability patterns—horizontal, vertical, serverless [8]. Sharma investigates system-level scalability impacts, presenting evidence that flexibility and performance may improve but not universally [9]. Xu et al. introduce CoScal, a reinforcement-learning framework mixing horizontal and vertical scaling with feature brownout—showing how technical creativity continues at the edges [10].

Results and Discussion

Achieving scalability begins with how microservices are designed. A core principle is to make services stateless wherever possible, meaning they do not maintain session-specific data internally. Stateless services can be replicated and load-balanced freely, since any instance can handle a request without relying on local state. Adopting stateless, loosely coupled services ensures that scaling out (adding instances) will not introduce consistency problems within the service [8]. In real systems this often drifts toward pushing state out — distributed databases, caches, anything that lets the service itself stay lighter.

Another idea that keeps surfacing is bounded contexts. Each microservice has its own area of responsibility by design. In practice boundaries blur, business rules change, and the clean lines can fray. Still, the modularization combined with tech-agnostic communication (REST, queues) does seem to give teams room to pick their own stacks and scale each piece as needed. As Domakonda notes, microservices improve resource utilization and flexibility compared to monoliths by isolating functionality and enabling independent deployment and scaling [5]. This capacity for independent scalability is often cited as a key advantage. Components of an application that experience surges in demand—such as a “checkout” service in an e-commerce system—can be replicated to absorb the load without having to over-provision the entire application [9]. Yet this benefit comes with an architectural trade-off: a microservice system adds layers of communication and coordination, which in turn can introduce latency and operational complexity. As a result, effective API management, service discovery, and robust monitoring infrastructures become essential to keep the system coherent and to prevent small issues from propagating across services [5]. Figure 1 summarizes the distribution of security threats reported in recent microservice deployments.

Figure 1. Prevalence of threat categories in microservice-oriented systems by Berardi et al. [2].

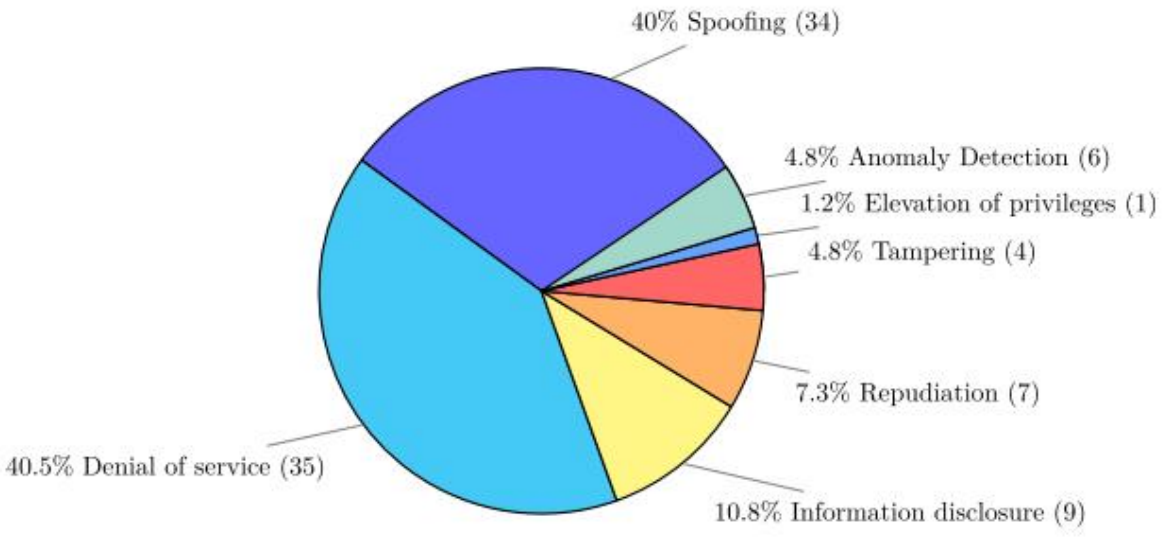


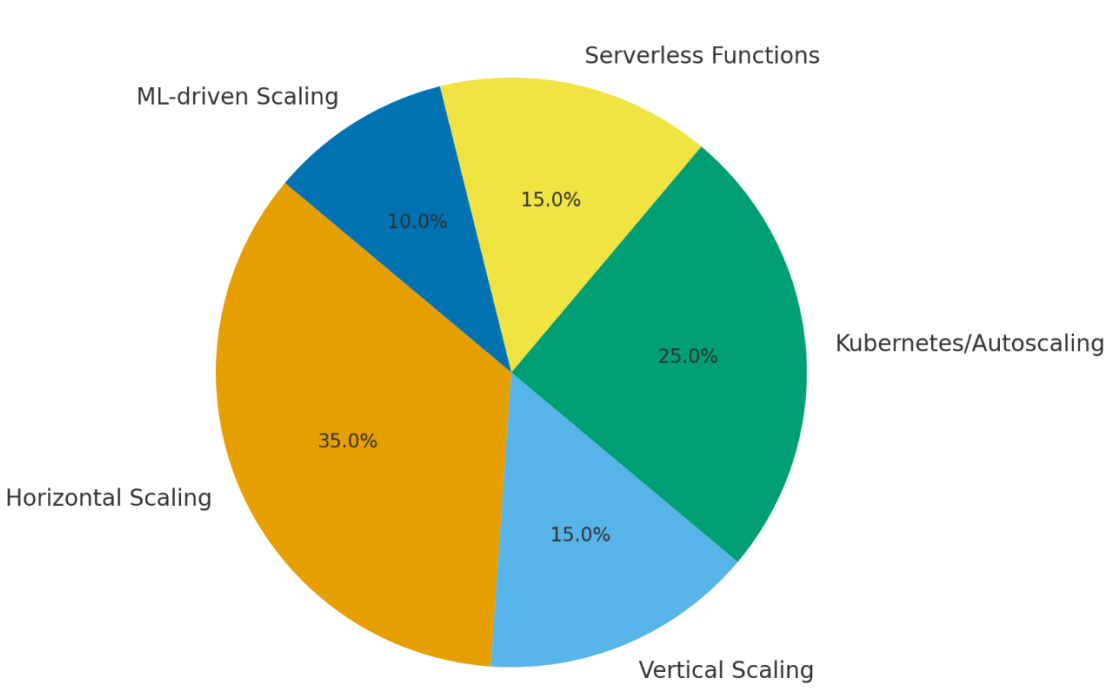
Figure 1 sketches the distribution of attack types in microservice-oriented systems as mapped onto the STRIDE threat model. The biggest portions — at least in this sample — appear to be denial of service (around 40.5 percent) and spoofing (roughly 40 percent). After that the numbers decrease: information disclosure about 10.8 percent, repudiation near 7.3 percent, with only slivers for tampering, anomaly detection, and elevation of privilege. It is not clear whether these proportions would hold in a different dataset; one possible explanation is that DoS and spoofing are simply easier to detect and report than subtler threats. This demonstrates that, as microservice systems become more distributed and independently scaled, their attack surface shifts heavily toward identity/communication exploits (spoofing) and availability threats (denial of service).

At the infrastructure level, there are two primary approaches to scale a service: horizontal scaling (also known as scale-out) and vertical scaling (scale-up). Horizontal scaling means increasing capacity by adding more instances of a service (e.g. launching additional container replicas or VMs), and distributing the workload among them via load balancing [8]. This is the de facto scaling method for microservices – since services are modular, one can run 5, 10, 100 copies of a microservice behind a load balancer to handle growing traffic. Horizontal scaling is often regarded as essentially unbounded — in practice it is limited by infrastructure and management overhead — and it tends to improve fault tolerance as well. If one instance goes down, others can usually keep serving [8]. Vertical scaling is the mirror image: instead of multiplying instances, more resources are piled into one.

It is important to note that horizontal and vertical scaling are not mutually exclusive; they can be combined for optimal results. Many cloud deployments use moderate vertical scaling (ensuring each instance has enough resources to be efficient) in tandem with horizontal replication. A 2022 study turned up an interesting pattern in a managed cloud environment (Azure). For a while, purely vertical scaling looked more cost-effective than horizontal scaling [5] — which seems counterintuitive given the usual enthusiasm for scaling out. Yet as more instances were added, performance started to sag, apparently from inter-service overhead. It is premature to state exactly where the break-even point lies; maybe it depends on workload shape or how services share data. Still, the finding suggests that there is a kind of “sweet spot” in scaling — too far in either direction and the gains begin to reverse. If there are too few instances and each node shoulders too much load; too many and coordination costs—network latency, load-balancer churn, cache-coherency headaches—start to erode throughput.

In real deployments, architects seem to confront this balance constantly. Some even place hard caps on autoscaling to prevent thrashing or runaway complexity. Blinowski et al., for example, found that scaling out past a moderate number of instances yielded diminishing returns and, in some cases, outright performance drops [2]. One possible explanation is that microservices introduce their own friction—API calls, inter-process communication, state synchronization—that scales non-linearly with distribution. Yet that friction may be offset in other workloads or architectures. To contextualize the architectural approaches discussed in the literature, Figure 2 illustrates the relative emphasis of scaling strategies found in recent microservice research and practice.

Figure 2. Relative emphasis of scaling approaches in microservice architectures by the author based on studies by Sharma [9] and Domakonda [5]



The pie chart on Figure 2 shows the distribution of attention and adoption among five key scaling approaches. Independent scalability is frequently presented as a major advantage. In practice, it allows only the parts of an application under pressure — for instance, a "checkout" service in an e-commerce system — to be replicated without over-provisioning the rest [9]. Yet this gain is not free. Splitting a system into many small services brings extra communication and coordination overhead, which can quietly erode performance. One consequence is that strong API management, service discovery, and careful monitoring move from "nice to have" to "required" [5]. These mechanisms help keep the pieces aligned but also introduce their own complexity, so the trade-off is not always straightforward.

Modern microservice systems almost always rely on container orchestration platforms (such as Kubernetes) to manage scaling in an automated way. A real-world example of large-scale orchestration came from the author's work on Prime Video's Startover Playback feature. This capability lets viewers restart a live event from the beginning while it is still being broadcast — a function that might sound simple but requires complex coordination behind the scenes. Multiple microservices handle ingest, encoding, and delivery, each scaling independently. When audience numbers spiked during big live events, raw server power alone wasn't enough. The system used distributed caching and adaptive load balancing to soak up sudden surges, keeping playback smooth even with millions of concurrent viewers. In day-to-day operation, the design mixed horizontal scaling with event-driven orchestration so the entire pipeline stayed responsive and fault-tolerant worldwide. This experience illustrates how thoughtful microservice design can turn backend scalability directly into visible reliability and higher viewer engagement.

Kubernetes is an open-source cluster manager that automates deployment, scaling, and operation of containerized services. It provides built-in autoscalers at both the horizontal and vertical level – the Horizontal Pod Autoscaler (HPA) can automatically add or remove container replicas based on metrics like CPU utilization, and the Vertical Pod Autoscaler (VPA) can adjust

resource limits of containers on the fly [7]. Choosing to rely on orchestration platforms is itself an architectural move. Rather than manually spinning up or shutting down instances, the system is built so that Kubernetes—or something similar—monitors service metrics and adjusts capacity based on the feedback. In principle this should deliver strong elasticity for cloud-based microservices [7], yet in practice the effect can be uneven. Monitoring intervals, scaling thresholds, and workload bursts can all shape how well the mechanism actually responds. Saradgishvili notes that Kubernetes' native autoscaling features have played a major role in improving efficiency under variable workloads [7]. One possible explanation is that the platform's built-in logic fits common use cases but may need tuning or augmentation for more complex systems. For example, it is possible to set a policy that if the CPU usage of a microservice's pods exceeds 80% for a sustained period, the HPA will create a new pod (container instance) of that microservice, and the ingress load balancer will start routing traffic to the new pod as well. Conversely, when load drops, the HPA can scale down the number of pods to reduce resource usage. Such dynamic provisioning ensures the application always has enough capacity to maintain performance without wasting resources during lulls [8].

An alternative architectural approach for scaling microservices is to use serverless computing, typically via Function-as-a-Service (FaaS) platforms (e.g. AWS Lambda, Azure Functions). In a serverless model, the unit of deployment is a small function (which may correspond to a microservice endpoint) that the cloud platform can instantiate on demand in response to events. Serverless architectures inherently provide automatic scaling – when an event (such as an HTTP request or message trigger) occurs, the platform runs the function, scaling out to as many parallel function instances as needed to handle incoming events, and scaling down to zero when idle. This paradigm can be viewed as an extreme form of microservices, focusing on single-function services that start up on demand. The key advantage is elasticity: resources are allocated exactly in proportion to the workload, with no need to manage servers or container pools for each microservice [8]. For example, an image-processing service might be deployed as a serverless function; if a spike of 1000 images to process comes in, the cloud will concurrently execute many function instances, then wind them down once finished, with billing only for actual execution time. As Pandiya (2021) explains, under a serverless architecture the application automatically scales up with inbound requests and scales down when demand falls, offering essentially unparalleled scalability without human intervention [8]. This on-demand scaling is often granular to the request level – e.g., AWS Lambda can start additional function instances in milliseconds when new events arrive, effectively matching even highly unpredictable loads.

Pricing adds another complication. Not every workload fits the serverless billing model. Long-running or very steady tasks can end up cheaper on reserved instances; bursty, unpredictable workloads are where serverless usually shines. In a separate context focused on optimizing performance within constrained hardware environments, a subtitle processing system was engineered as part of Prime Video's just-after-broadcast (JAB) pipeline. The goal was to keep subtitles accurate and in sync even on older smart TVs with very limited memory. To achieve this, the team used a small segmentation algorithm that broke subtitle data into short, time-based chunks and eliminated repeated lines — a method later reflected in U.S. Patent 10,893,331 B1. This cut memory use and bandwidth at the same time, keeping playback stable and allowing the system to scale upward without changing the hardware.

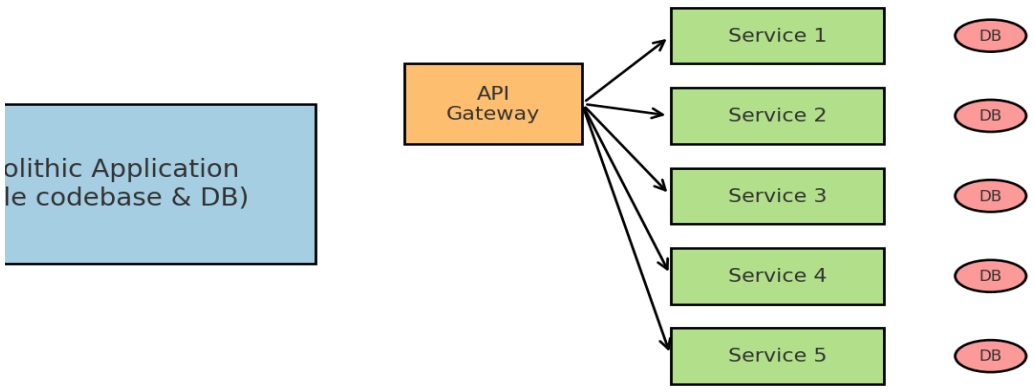
This case also shows that scalability doesn't always come from adding servers or infrastructure. Careful tuning of algorithms and data handling can achieve similar gains. At the same time, it highlights that serverless computing is not a one-size-fits-all solution. Its value depends on the workload pattern, the limits of the chosen cloud provider, and overall cost sensitivity. In practice, many teams end up with a hybrid design — core services running in long-lived containers, while event-driven or bursty components run as serverless functions. Architecturally, this means designing the system to emit and respond to events. For instance, an e-commerce site might use microservice containers for its main web API and use serverless functions for background tasks like sending notifications or generating reports on demand. Such a serverless event-driven architecture is depicted by Pandiya, where AWS Lambda functions are integrated as event handlers in the microservice ecosystem [8]. The takeaway is that serverless computing can be viewed as another tool for scaling: it shifts the responsibility of scaling to the cloud provider's platform, which will transparently allocate containers and threads to meet the event rate.

One strand getting attention is the use of machine learning — especially reinforcement learning — to guide autoscaling. The thinking goes that the old rule-of-thumb triggers ("if CPU > 70%, add one instance") can be inefficient, sometimes overprovisioning or reacting too late. By contrast, trained models might anticipate demand and adjust capacity ahead of time. The implication is that machine-learning autoscaling is promising but still experimental. Xu et al. introduce CoScal, a multi-faceted scaling approach that combines horizontal scaling, vertical scaling, and even brownout techniques under an RL agent [10]. Brownout in this context means dynamically disabling non-essential features when the system is under heavy load, thereby reducing resource usage. CoScal's RL agent uses deep learning to forecast workload and then decides the best combination of actions – e.g., scale out certain services, scale up resources on another, or temporarily disable a feature – to minimize response time and cost. Studies using these techniques generally report better resource utilization and service quality than static policies [10]. In effect, the architecture takes on an extra layer — an intelligent controller that keeps adjusting its own scaling rules for a complex microservice setup. One example is Alqassem et al., who built a proactive autoscaler using Random Forest predictors to anticipate bursts in microservice workloads [10].

While these approaches sound promising, they also pull in added complexity and remain very much in the research stage. Architecturally, adopting a learning-based autoscaler means collecting far more telemetry — response times, resource usage, request rates, maybe even custom application metrics — and feeding it back into a model that may need retraining or at least regular updates. This extra loop can pay off, but it also risks introducing new points of failure or drift over time. In practice the "intelligent" autoscaler can end up as another system to monitor, rather than a self-managing black box. Although such techniques are still maturing, they represent an architectural layer on top of autoscaling – essentially a verification or optimization layer – that could be incorporated into future microservice platforms to automatically tune scaling behavior for multiple objectives (performance, cost, energy).

It is important to discuss the cost and complexity trade-offs that come with scaling microservices. The ability to scale each service independently is powerful, but it often means an organization ends up managing dozens or hundreds of service instances, each incurring resource and management overhead. Chavan points out that as an organization grows its microservices deployment, the operational costs can "skyrocket" because each microservice demands its own infrastructure, development pipeline, monitoring, and security measures [3]. The contrast between a monolithic deployment and a microservice-based deployment is illustrated in Figure 3.

Figure 3. From monolith to microservices: Increased components and operational overhead - the author's illustration of implications made by Blinowski et al. [3]



The diagram on Figure 3 highlights how scaling microservices adds multiple independent services, databases, and a gateway layer, illustrating the cost and complexity trade-offs of distributed scaling. On the left, the diagram shows a single monolithic application — one codebase, one database — almost spartan in its simplicity. Scaling here is straightforward, if limited: adding more power to the same machine or running a few replicas behind a load balancer. On the right, by contrast, are multiple microservices, each with its own container and a small database or cache, fronted by an API gateway.

In short, microservices swap simplicity for agility. They gain the ability to evolve, deploy, and scale independently but at the cost of more moving parts. As a result, the architecture must include compensating mechanisms: an API gateway to centralize some cross-cutting concerns (authentication, routing), a service mesh to make inter-service communication more predictable, and robust monitoring and alerting so that engineers can actually see what's happening. Autoscaling to zero (shutting down idle services) and rightsizing resources are two common tactics to avoid waste, and many teams now adopt FinOps practices alongside their technical scaling strategies [3]. Despite these challenges, when it works, microservices let organizations line up resources closely with demand. A well-designed microservice system can deliver predictable performance under load while optimizing cost — but only if it is designed with caution. That might mean picking the right scaling strategy for each service, setting safe limits to prevent runaway replicas, and constantly tuning the metrics that drive autoscalers.

Taken together, the findings suggest scaling distributed microservices in the cloud is less a single technique and more a layered approach. First, services need to be as stateless, decoupled, and independently deployable as possible — otherwise scaling flexibility disappears. Second, horizontal scaling is probably still the main lever for increasing capacity — yet vertical scaling hasn't disappeared; it keeps a role, at least until the point where returns flatten out [2].

None of these elements really solves the scalability challenge on its own. Each addresses a fragment of it, and sometimes they even work at cross-purposes. When combined carefully they can make it possible for a microservice system to withstand heavy demand on cloud infrastructure without losing reliability or efficiency. In the real world the picture is usually patchier, with monitoring gaps, cost surprises or autoscalers behaving unpredictably. The mix of patterns, compromises and ongoing tuning is perhaps the real hallmark of scaling microservices today — less a single formula than a set of overlapping practices.

Conclusion

Scaling distributed microservice systems in the cloud seems to demand both thoughtful architecture and cloud-native tooling. At the design level, modularity and statelessness look like the bedrock principles — services should, at least in theory, be able to scale out simply by adding instances without fighting state-synchronization problems. Still, the evidence suggests that when microservices are kept as stateless and decoupled as feasible, scaling out becomes far less taxing. Embracing loose coupling and independent deployment ensures that each service can be scaled (or updated) on its own timeline, which is fundamental for responding to uneven load patterns across an application. A careful choice between horizontal and vertical scaling (and often a blend of both) is part of the architectural strategy: horizontal scaling provides practically unbounded capacity by replicating services, whereas vertical scaling can yield performance gains up to hardware limits – the optimal mix depends on the application's characteristics and cost considerations.

Modern cloud-based microservice architectures tend to lean on horizontal scaling through orchestration platforms — though exactly how well this works can vary widely. Kubernetes, for instance, has become almost a default choice. It bundles autoscaling, load balancing, and self-healing into one ecosystem, which seems essential during unpredictable traffic spikes. Yet it is not always obvious whether the built-in mechanisms alone are enough; in some workloads they are smooth, in others they need heavy tuning.

Scaling a microservice system is an ongoing architectural concern. The goal is always a moving target: enough capacity for peaks without wasting resources during lulls. Patterns such as stateless service design and database sharding can make scaling simpler, while advanced autoscaling frameworks give architects more control — but only if applied thoughtfully. Sometimes these strategies do hold up — the system stays surprisingly responsive under heavy load, which seems

to validate the microservice idea. Other times the same setup turns costly or fragile, and the gains aren't obvious at all. It's hard to predict which way it will go without running it at scale. Looking ahead, future research can focus on energy-aware scaling, cross-layer optimizations, and "autonomic" orchestration — architectures that are supposed to learn and adapt over time.

References

1. Alharthi, S., Alshamsi, A., Alseiari, A., & Alwarafy, A. (2024). Auto-Scaling Techniques in Cloud Computing: Issues and Research Directions. *Sensors*, *24*(17), 5551. https://doi.org/10.3390/s24175551
2. Berardi, D., Giallorenzo, S., Mauro, J., Melis, A., Montesi, F., & Prandini, M. (2022). Microservice security: a systematic literature review. *PeerJ Computer Science*, *8*, e779. https://doi.org/10.7717/peerj-cs.779
3. Blinowski, G. J., Ojdowska, A., & Przybyłek, A. (2022). Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE Access, 10*, 20357–20374. https://doi.org/10.1109/ACCESS.2022.3152803
4. Chavan, A. (2023). Managing Scalability and Cost in Microservices Architecture – Balancing Infinite Scalability with Financial Constraints. *Journal of Artificial Intelligence & Cloud Computing*, *2*(4), 1–14. https://doi.org/10.47363/JMHC/2023(5)E102
5. Domakonda, D. (2025). Secure and Scalable Microservices Architecture: Principles, Benefits, and Challenges. *International Journal of Scientific Research in CSEIT*, *11*(2), 1897–1902. https://doi.org/10.32628/CSEIT23112569
6. Filippone, G., Pompilio, C., Autili, M., & Tivoli, M. (2022). An architectural style for scalable choreography-based microservice-oriented distributed systems. *Computing*, *105*(9), 1933–1956. https://doi.org/10.32628/CSWEIT23112569
7. Agos Jawaddi, S.N., Ismail, A., Sulaiman, M.S. et al. Analyzing Energy-Efficient and Kubernetes-Based Autoscaling of Microservices Using Probabilistic Model Checking. *J Grid Computing*, *23*, 3 (2025). https://doi.org/10.1007/s10723-024-09789-9
8. Dileep Kumar Pandiya. (2021). Scalability Patterns for Microservices Architecture. *Educational Administration: Theory and Practice*, *27*(3), 1178–1183. https://doi.org/10.53555/kuey.v27i3.6897
9. Saurav Sharma. (2025). The Impact of Microservices Architecture on System Scalability. *American Scientific Research Journal for Engineering, Technology, and Sciences*, *102*(1), 140-148. https://asrjetsjournal.org/American_Scientific_Journal/article/view/11677
10. Xu, M., Song, C., Ilager, S., Gill, S. S., Zhao, J., Ye, K., & Xu, C. (2022). CoScal: Multi-faceted scaling of microservices with reinforcement learning. *IEEE Transactions on Network and Service Management*, *19*(4), 3995–4009.https://doi.org/10.1109/TNSM.2022.3210211