International Journal of Science and Research (IJSR)
ISSN: 2319-7064
Impact Factor 2024: 7.101

Technical Limitations of Microservices Scaling and Ways to Overcome Them

Gleb Shkriabin

White Code Works LLC, CTO, USA, New Jersey

Abstract: ***The article explores the fundamental limits of scalability in microservice architectures, arguing that microservices do not scale as an architectural property but only remain scalable while three operational planes- the data plane, transaction plane, and control/measurement plane- stay within stable regimes. Once any plane crosses a critical threshold, the system undergoes a phase transition from stable operation to instability, manifesting as unbounded queue growth, silent correctness violations, or self-interference caused by orchestration and observability overhead. The article leans on benchmark data, a few experiments, and earlier surveys. Queue growth slope - or consumer lag- is a main signal; it shows when the data path starts slipping. Other failures come from concurrency-related consistency bugs, or from monitoring tools pulling too many resources and getting in the way. What stands out is that systems often begin to fall apart well before maxing out hardware; coordination costs and shared state play a much bigger role. The text outlines a few mitigations: deferred commit with rollback in memory, SLOs based on queue trends, and evolving telemetry setups alongside system changes. Overall, the goal is to map where scale breaks- not in theory, but in actual deployments. The article will be useful to system architects, SREs, and researchers designing or evaluating large-scale microservice systems, providing concrete criteria for identifying real scalability limits and reframing "scaling" as a stability problem rather than a resource-provisioning problem.***

Keywords: microservices scalability, distributed transactions, saga pattern, Service Level Objectives (SLOs), distributed systems performance

1. Introduction

In cloud-based software, microservices are often chosen when modular structure and scaling are needed. Instead of one large application, the system is split into parts—smaller services that can be changed or released on their own. This makes it easier to adjust specific components under load, compared to how monolithic systems behave. Leading practitioners (e.g. Netflix, Amazon) report benefits such as modularity, independent development, and fault isolation. However, achieving true scalable performance in microservices is non-trivial due to fundamental distributed systems challenges. This article argues that microservices do not inherently "scale" as an architecture; rather, they scale only as long as three planes of operation remain stable:

- Data Plane: the actual data flow and state management (throughput and state correctness).
- Transaction Plane: the consistency of data updates under distributed concurrency.
- Control/ Measurement Plane: the overhead of orchestration, coordination, and observability (monitoring, metrics, etc.).

Systems do not always degrade in a smooth line. In different layers, there is usually a threshold where performance drops off, errors emerge, and the shift is more sudden than slow. Past this point, services may falter, results can become unreliable, and overall stability is no longer guaranteed. For example, if the data plane cannot keep up with incoming load, message queues start growing without bound (a backlog explosion rather than mere latency increase). In the transaction plane, a surge in concurrent updates can trigger consistency anomalies (e.g. stale reads or lost updates)- a "silent" correctness collapse where throughput stays high but results become wrong. In the control and measurement layer, the monitoring or orchestration components themselves can sometimes interfere with the main workload. For instance, if sidecars or load generators start drawing too much CPU or memory, they may unintentionally affect the behavior of the system being tested, introducing feedback effects.

To pinpoint when scalability starts to degrade, one of the key metrics tracked is how quickly queues begin to grow- how fast backlog accumulates under load. Henning and Hasselbring (2024) used this metric, known as the consumer lag trend, to set a Service Level Objective: as long as queue growth stays below a small fraction of the input rate, the system is considered scalable. Yet once that slope turns positive and stays there, it usually means the system can no longer keep up in real time — and is slipping into instability. This notion of stability is extended to the other operational planes. Accordingly, the analysis focuses on three characteristic failure modes: queue-stability breakdown in the data plane, silent correctness collapse due to transactional inconsistency in the transaction plane, and measurement-plane interference caused by escalating control-plane overhead. For each mode, mechanism-based approaches for shifting these phase-transition boundaries outward are discussed.

The analysis pulls from recent benchmarks where stream-processing systems were pushed until they started missing SLOs. Each failure mode is shown in context, not just in theory. On the mitigation side, the paper sketches out options like commit deferral (with rollback limited to in-memory state) and keeping observability systems aligned with OpenTelemetry as they evolve. The three-plane model is used to frame these breakdowns- each one marks a shift into a less stable regime. Finally, survey results on monitoring tools are brought in to expose some of the blind spots that still persist in current observability tooling.

2. Methods and Materials

The analysis is grounded in findings reported by several recent empirical studies and surveys of microservice-based systems. Henning and Hasselbring (2024) conducted over 740 hours of experiments benchmarking five stream processing frameworks (Flink, Kafka Streams, Samza, Hazelcast Jet, and Apache Beam) deployed as microservices. Henning and Hasselbring (2024) examined scalability by pushing message throughput to 1 million msgs/sec and gradually increasing Kubernetes-based microservice deployments to 110 instances. The primary signal of overload was consumer lag growth, tracked as a slope-based SLO. Use cases UC1–UC4 illustrated how backlogs emerge even before full resource exhaustion.

Separately, Daraghmi et al. (2022) explored a modified Saga approach, adding a memory-resident quota cache and deferring database commits. Evaluation on a basic e-commerce service showed that, compared to the standard pattern, their version maintained read isolation and responded faster under failure- primarily because incomplete writes never reached the main store. Giamattei et al. (2024) performed a systematic grey literature review of 71 monitoring tools used in DevOps for microservice systems. They catalogued each tool's features (e.g. metrics, tracing, logging capabilities), instrumentation requirements, and integration aspects. The study includes quantitative summaries of what's commonly implemented for observability, but also draws attention to gaps- for instance, few tools support testing integration, and energy monitoring is largely absent.

In a separate review of 85 primary sources, Söylemez et al. (2022) observed that scalability issues rarely emerge alone. More often, they coincide with orchestration burdens and monitoring friction, suggesting that performance degradation in MSA is entangled with control-plane complexity. Their synthesis grouped the problems into nine thematic clusters, ranging from performance degradation to coordination complexity, underscoring the intertwined nature of these challenges. Elkhatib and Poyato (2023) executed controlled experiments to compare two popular service mesh frameworks- Istio and Linkerd- in an edge computing Kubernetes cluster. They measured end-to-end communication latency, CPU and memory overhead introduced by the mesh sidecars, and throughput impacts, by running a sample microservice application at different scales (varying number of services) with and without a mesh. Čilić et al. (2023) tested several container orchestrators- K3s, KubeEdge, and ioFog- using real-world edge setups instead of simulations. They were mainly interested in how these systems managed scheduling as conditions shifted, particularly across cloud and edge boundaries. Latency and adaptability under IoT workloads served as practical indicators of whether the platforms could handle production demands.

Erdei and Toka (2023), meanwhile, looked at how resource allocation affects the throughput of a specific microservice- Cortex, a distributed metrics store. They ran the system with different CPU and memory caps and used regression analysis to see how throughput responded at each level. The results pointed toward a resource-saving strategy that kept performance mostly stable. For general architectural context, Velepucha and Flores (2023) summarized several design patterns seen across recent microservice setups. Meanwhile, Senjab et al. (2023) focused on Kubernetes scheduling- especially newer techniques aimed at better load distribution and handling QoS requirements. These provide context but the core analysis centers on the empirical studies above.

3. Results and Discussion

A fundamental requirement for scalable microservices is that the data plane – the flow of requests or messages through the services – remains stable under increasing load. In a well-scaled system, each service instance can handle its share of the load such that queues do not continuously grow. The queue growth slope (consumer lag trend) is therefore a key indicator: a near-zero or negative slope (queue length staying constant or draining) means the system keeps up with input; a positive slope means backlog is accumulating, a clear sign of unstable behavior.

Hasselbring and Henning's (2024) streaming microservices benchmarks vividly demonstrate this threshold effect. For a fixed high input rate (e.g. 50,000 msgs/s in a Kafka Streams pipeline), scaling from 6 to 8 service instances turned the system from unstable to stable. With 6 instances, the number of queued messages kept increasing steadily over time, indicating the pipeline was falling behind. At 7 instances, the backlog initially grew but then started decreasing after a warm-up, and at 8 instances the queue length trend became almost flat (zero slope). In other words, somewhere between 6 and 8 instances lies a phase transition: below that point, each incremental load causes compounding queue buildup; above that point, the system can absorb the load without infinite queues. This transition defines the practical scalability limit for that specific workload and framework. Figure 1 illustrates how microservice scalability in the data plane exhibits a non-linear stability threshold rather than a gradual improvement with added replicas.

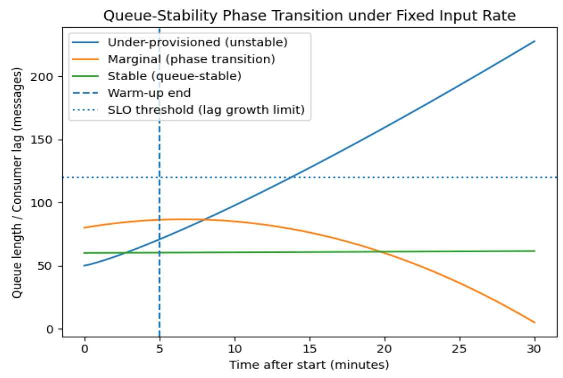

Figure 1: Queue-stability phase transition in stream-processing microservices under fixed load

In Figure 1, the under-provisioned setup shows queues growing steadily, even though the input rate stays constant. This suggests the system is not keeping up with incoming messages. The configuration above that threshold behaves differently: backlog builds up at first, but then starts to drop after a warm-up phase. Between the under-provisioned and well-provisioned setups, there's a narrow region where queue behavior changes. In the borderline case, the backlog initially rises but eventually starts to fall. After enough instances are added, the queue length levels off. In that configuration, the slope remains low enough to stay within the SLO. Figure 1 illustrates this transition using two lines: one marking the end of warm-up, and another showing the lag threshold. The change is not gradual- it shows up suddenly once the system crosses that replica count.

Henning and Hasselbring (2024) formalized this with an SLO: the lag trend (slope) must not exceed 1% of the input rate to consider the service scalable at that load. Exceeding that threshold is effectively SLO breach – the queue-stability failure mode. Notably, different frameworks exhibited this failure at different resource levels: e.g., for one use case (UC3, a complex streaming join), Hazelcast Jet managed to process all loads with a single instance (no backlog growth), whereas Apache Flink required ~10 instances and Kafka Streams 18 instances to avoid lag growth. This shows how architecture choice can change where the phase transition occurs. But in all cases, if load kept increasing without adding resources, eventually backlog growth would reappear.

In a real production scenario, a growing message queue is a red flag for SLO violation. For instance, Henning and Hasselbring (2024) set a concrete benchmark SLO that the backlog growth be <1% of input volume. If the consumer lag trend exceeds that (even if throughput is high), the system is not truly scaling. This underscores that throughput alone is insufficient – one must watch queue length metrics to detect looming instability.

Another insidious data-plane failure mode is silent correctness collapse. This occurs when throughput remains seemingly adequate (no obvious queue buildup) but the system starts silently dropping or corrupting data under pressure. Henning & Hasselbring observed this in certain stream processing configurations: under high load, some frameworks did not show rising lag, yet they were discarding records due to latency constraints (e.g. late arrivals beyond a window allowed lateness). In their benchmarks UC2 and UC4, if the input rate was too high, frameworks like Apache Beam would start skipping events that arrived outside the allowed window, thus maintaining throughput but producing incomplete results. The backlog stayed low (since dropped records don't queue), giving a false sense of stability, while output accuracy degraded. The authors had to introduce a second SLO: no more than 1% of messages should be dropped. Failing this SLO is a phase change in correctness: the system transitions into an undetected error state where it keeps up throughput by sacrificing correctness. Figure 2 illustrates a silent correctness collapse in the data plane, where conventional performance metrics mask semantic failure.

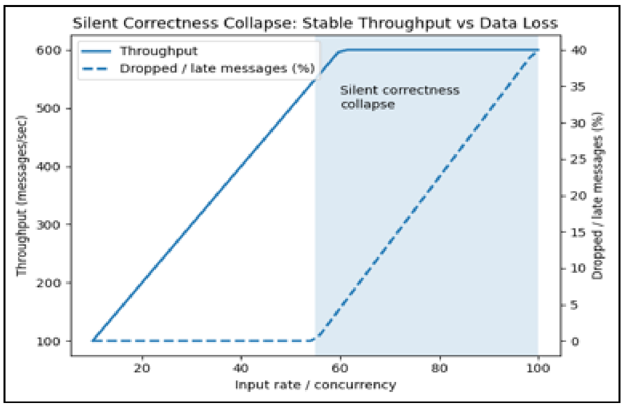

Figure 2: Silent correctness collapse under load: throughput remains stable while data loss increases

As shown in Figure 2, when input rate increases, system throughput initially improves but eventually reaches a plateau- this would typically indicate stability under load. Yet past a certain threshold, the rate of dropped or delayed messages begins to climb, even though throughput itself stays mostly unchanged. In this regime, throughput-based SLOs can remain within configured thresholds even as data integrity violations accumulate. The reason is largely accounting-related: events that are dropped, skipped, or processed beyond application-level deadlines do not contribute to queue length or backlog growth metrics. As a consequence, ingress–egress rates remain stable despite the fact that the effective result set is already incomplete or temporally invalid.

Systems that handle stateful streams or low-latency analytics tend to surface subtle issues as they scale. Henning and Hasselbring's benchmarks (e.g., UC2, UC4) showed that some frameworks dropped events arriving outside allowed time windows, even though throughput remained stable. These drops did not always show up in conventional metrics. In those cases, output completeness degraded without a corresponding change in message rate. To identify such issues, instrumentation must consider not just performance indicators but the validity of what is actually output. Failures typically arise in one of two forms: (i) performance disruption, where queues or latencies increase significantly; (ii) semantic divergence, in which the system's outputs deviate from correct or complete behavior even while overall flow seems unaffected. Figure 3 synthesizes the core argument of the paper by mapping scaling limits across three distinct operational planes of a microservices architecture.

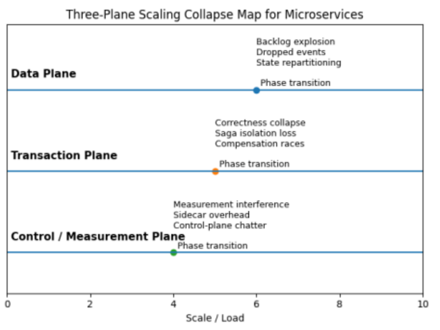

Figure 3: Phase-transition model of microservices scaling limits across data, transaction, and control planes

Each plane on Figure 3- data, transaction, and control/ measurement- exhibits a non-linear phase transition as scale or load increases, but the nature of the collapse differs by plane. In the data plane, scaling beyond the stability threshold leads to backlog explosion, state repartitioning overhead, and dropped events. As the number of services increases, so does the overall system activity. Some of that increase comes from the coordination layer—control messages, health checks, and various observability components. Control and monitoring components are usually treated as background utilities. However, during scaling experiments in Henning and Hasselbring's (2024) benchmark, these layers showed signs of interference with the main workload. The paper attributes this partly to resource contention between the service-under-test and ancillary processes. Notably, this was most evident beyond 30 microservice instances in one of the clusters they examined. These effects are not always linear. Several studies have observed abrupt shifts in system dynamics once specific load thresholds are crossed. The figure emphasizes that microservices do not "fail to scale" uniformly—each plane collapses differently, and often before resource exhaustion is reached.

To overcome data plane scaling limits, one obvious approach is to add more instances (horizontal scaling). All frameworks in the streaming benchmark showed roughly linear scalability until hitting their limit, meaning provisioning additional microservice replicas did linearly increase throughput. However, horizontal scaling has diminishing returns if it introduces significant coordination overhead or if some component becomes a bottleneck (e.g. the message broker or a downstream datastore). An alternative or complementary approach is vertical scaling – giving each instance more CPU or memory. Interestingly, Henning & Hasselbring (2024) found that for certain frameworks like Flink and Hazelcast Jet, scaling up the CPU cores per instance could be more resource-efficient than adding instances. For example, in one scenario Flink on a single powerful node achieved what multiple weaker nodes would, implying the overhead of distributing work across instances was the limiting factor. They conclude that vertical scaling can complement horizontal scaling, and caution that one should identify if a workload scales better by multi-core usage than by multi-instance parallelism. This is likely because adding instances introduces network and partitioning overhead, whereas adding cores avoids inter-instance communication (but one must watch out for shared resource contention on a single node).

Empirical evidence of this comes from Erdei & Toka's (2023) study of a metrics service (Cortex). The study observed that throughput scaled proportionally with CPU allocation for a given microservice, up to a point. This made it possible to estimate performance and tune resource allocation more precisely. For workloads operating within predictable resource envelopes, vertical scaling- assigning more cores to a single instance- proved efficient. However, once those limits were hit (e.g., due to garbage collection stalls or I/O saturation), the benefits plateaued, and distributing the workload across multiple instances became necessary. In practice, a hybrid strategy (scale each instance vertically to an efficient size, then replicate) can avoid the extremes of many tiny instances (with high coordination cost) or one huge instance (with potential single point of failure).

The Saga pattern manages a distributed transaction as a sequence of local transactions with compensating actions for rollback on failure. However, the classic Saga has a known limitation: lack of isolation. Because each step commits locally, other services might see partial updates before the saga fully completes or compensates. In database terms, Saga provides Atomicity and Durability, but not true Isolation (it's ACD, not ACID). At scale, when many sagas execute concurrently, this can lead to anomalies like dirty reads, lost updates, or cascading rollbacks. In the context of the "phase transition" theme, there may be a threshold of concurrency beyond which the probability of such anomalies skyrockets, effectively breaking the transactional correctness plane.

Daraghmi et al. address this with a mechanism-based solution that can be seen as shifting the phase transition of consistency to a higher load. They propose an Enhanced Saga pattern that introduces a quota cache layer and a commit coordination service. In this design, when a distributed transaction is initiated, all the involved services perform their CRUD operations on an in-memory cache (allocated as a quota of the main database) instead of immediately on the primary database. These tentative updates in the cache remain invisible to other services (ensuring isolation) until the entire saga is confirmed successful. Only then does a commit-sync service propagate the changes from cache to the actual databases in one go. If any step fails, compensating transactions roll back the changes in the cache only, never having polluted the main database state. This effectively defers the final commit to the end of the saga – achieving commit deferral- and ensures that any partial work is ephemeral (cache-only) unless the whole transaction succeeds.

In terms of phase transition, the transaction plane instability often manifests as either thrashing (where many transactions repeatedly abort/compensate each other, wasting effort) or inconsistency outbreaks (where system state temporarily diverges and causes errors). For instance, consider an inventory microservice and an order microservice: under heavy concurrent orders, a naive saga might allow several orders to oversell an item because each reads the old stock before previous orders have decremented it. Only later, compensating actions might cancel some orders. Up to a certain order volume, stock updates may succeed, but beyond a threshold, the system might get swamped in cancellations- a sharp drop in effective throughput (completed orders) and customer confusion. The enhanced saga approach would serialize the stock decrement in the cache, preventing that oversell scenario at higher loads, thus increasing the stability region.

The control plane in microservices encompasses the infrastructure and management tasks – deploying services, routing requests, scaling instances, etc. Closely related is the measurement/observability plane, which involves monitoring and collecting metrics, logs, and traces from the running services. These planes are critical for reliability and automated scaling (e.g. autoscalers use metrics to decide when to add instances). However, they introduce their own overhead. At small scale, this overhead is usually negligible, but as the number of microservice instances grows, the control/measurement plane can itself become a bottleneck or source of instability. A classic example is a monitoring agent or sidecar container that runs alongside each microservice: having N services means also possessing N sidecars, which consume CPU/memory and generate metric traffic.

Henning & Hasselbring's (2024) experiments provided a concrete manifestation of this: when they tried to scale beyond 30 microservice instances in one cluster, they encountered interference between the system under test (SUT) and the infrastructure components. The load generator services, the Kafka message brokers, and the Kubernetes control plane all started contending for resources, causing unstable, irreproducible results beyond that point. Essentially, the measurement harness became part of the problem- the cluster was saturating not because the stream processing logic failed, but because the ancillary components (intended to support or measure the system) were overstressed. The authors mitigated this by isolating roles on separate node pools (dedicating some VMs to load generation and Kafka, and others purely to SUT instances). This is instructive: it shows that the act of measuring and managing a system can perturb its scalability. For faithful scaling tests (and, by extension, production deployments), one must budget resources for control-plane components and possibly decouple them from the data plane as much as possible.

Another aspect of control plane overhead is the use of service mesh proxies. Service mesh (SMT) adds a proxy (sidecar) next to each service instance to handle networking, tracing, security policies, etc.. While this yields powerful control (e.g. dynamic routing rules, mTLS encryption, uniform telemetry), it comes at a cost. Elkhatib & Poyato's (2023) evaluation of Istio vs Linkerd quantifies this cost: Istio's proxy induced about ~10% higher request latency compared to baseline (no mesh), and used 1.2–1.4× more memory and ~1.2× more CPU per node than Linkerd. Linkerd was lighter but still an overhead relative to no-mesh. For edge environments with constrained resources, they concluded that the heavier mesh (Istio) might not be suitable. Importantly, this overhead accumulates with scale – every additional microservice carries the tax of its sidecar. If each sidecar uses, say, 50 MB memory and 5% CPU, then 100 microservices would collectively consume 5 GB memory and significant CPU just for the mesh layer. At some point, the cluster could run out of resources not because of the business logic, but because the control plane components (proxies, agents, etc.) eat too much. This is a phase transition in overhead: the system goes from a state where overhead is marginal (and largely constant as a fraction of load) to a state where overhead dominates incremental resources (e.g. adding more instances yields diminishing returns because so much extra overhead comes along).

A recent survey of 71 microservice monitoring tools found that nearly all (68 out of 71) require installing some form of instrumentation or agent into the system. In fact, 59 of 71 tools rely on a vendor-specific platform or collector service running alongside the microservices. Only 3 tools (4.2%) claimed to need no instrumentation at all. This means almost every observability solution adds some code or sidecar that consumes resources. At large scale, those agents can introduce non-trivial overhead and even points of failure (if a monitoring agent crashes, it can affect the service it monitors).

The survey by Giamattei et al. (2024) shows that while tools support various "monitoring patterns" (metrics, logging, tracing, etc.), using all of them incurs cost. The analysis encompassed 71 monitoring tools, of which only 11 provided comprehensive support for six fundamental observability functions. The remainder omitted one or more, typically to avoid the resource cost associated with full-spectrum instrumentation. Support for energy-related metrics was especially rare: only three tools included any such capability. This lack is not unexpected, given the difficulty of collecting reliable power data in distributed environments. Nonetheless, fluctuations in energy usage have been observed to coincide with performance issues such as CPU throttling. Some newer systems have tried minimizing the burden of monitoring by moving parts of it into the kernel (e.g., via eBPF) or reducing trace volume through aggressive sampling.

The fragmented landscape of observability tools means many deployments have uneven coverage. For example, distributed tracing is supported by only 30/71 tools (~42%), with the majority 41 tools not offering tracing. Meanwhile, an overwhelming 60/71 tools (85%) have no integration with testing frameworks – meaning they cannot easily correlate runtime metrics with test cases or CI/CD. This "observability skew" implies that teams often bolt on monitoring that can tell them when something is wrong, but lack the integration to proactively test or trace why at scale. Consequently, when systems scale up and fail, diagnosing the root cause can be as challenging as fixing it.

One insight from Giamattei et al. is that monitoring must evolve with the system. They advocate for supporting open standards like OpenTelemetry and ensuring the instrumentation code evolves in tandem with microservice code (what they term co-evolution). OpenTelemetry provides a vendor-neutral API for tracing/metrics, so developers are not locked into heavy proprietary agents. Co-evolution means when a microservice changes (say, new operations or data flows), the telemetry is updated accordingly, preventing blind spots or excessive logging of obsolete data. This approach can keep the measurement plane effective and minimal – you collect just what is needed for the current architecture.

As seen in the streaming benchmark, isolating control plane components on separate resources can prevent interference. In practice, this means running monitoring infrastructure (log collectors, metric databases, APM tool backends) on dedicated nodes or clusters, and similarly isolating load generation or client simulation tools. Kubernetes and other orchestrators increasingly support this via taints/tolerations or separate clusters for ops tooling. It ensures the core services aren't vying with the monitoring for CPU. Additionally, one should budget the overhead: e.g., if each service instance needs 200 millicpu for the sidecar, factor that into scaling decisions (essentially treat it as part of the service's requirements). Some mesh frameworks allow deploying proxies per host instead of per service ("shared sidecar") to cut down duplication – this could be a worthwhile trade-off if per-service granularity isn't needed.

The control plane itself should be scalable and adaptive. For example, Kubernetes has autoscaling for its etcd and controller managers in large clusters, but often these defaults aren't tuned for very large microservice counts. Senjab et al. (2023) surveyed various Kubernetes schedulers and pointed to a rise in specialized ones—for example, those that take into account network location or energy use. In some cases, the scheduler could pause scaling if the control plane is nearing its limit. Monitoring tools can also cut back on metric collection when the system gets too large in order to stay under resource caps.

4. Conclusion

Microservice systems often appear scalable, but real-world tests show this depends heavily on how they are set up and run. As systems grow, limitations associated with data processing behavior, transactional coordination, and overall system management tend to recur, rather than appearing as isolated issues. When any one of these lags or breaks, the system undergoes a phase transition to an unstable regime.

The data plane must handle growing throughput without unbounded queues or data loss. Queue backlog slope serves as an early warning: a rising slope heralds loss of real-time processing ability. Techniques like dynamic autoscaling, backpressure, and careful load balancing should be employed to keep this slope near zero. One possible direction for future work is smarter admission control- say, slowing or dropping requests when lag gets worse, to keep queues from exploding.

On the transaction side, consistency under concurrency remains tricky. Some teams (see [2], and also noted in [7]) have attempted to extract patterns of access from historical logs, aiming to predict or mitigate such conflicts ahead of time. It seems no universally accepted standard or formal procedure has yet emerged in this area. Lungu and Nyirenda (2024) describe a large set of solutions, coordination layers, fallback systems—but ultimately, consistency across services remains brittle. Even when sagas compensate, failures during the rollback itself are hard to trace. That's made worse by limited tracing support: only 30 out of 71 tools (survey data) include distributed tracing. That leaves a gap—more than half of systems effectively blind to what breaks first when sagas fail under pressure.

The control and measurement plane must not become a bottleneck itself. For example, only 11 of 71 tools implemented all key monitoring patterns – future tools should aim for completeness without excess overhead, perhaps by using unified collection agents (avoid duplicating instrumentation for metrics vs tracing vs logs). OpenTelemetry adoption is a step in the right direction; it can prevent redundant instrumentation by standardizing it. Researchers and practitioners should also collaborate on sidecar co-evolution: as microservice frameworks (service meshes, etc.) evolve, ensure the sidecars or agents are optimized (using techniques like zero-copy telemetry, kernel offload, or aggregator proxies that reduce N sidecars to 1 per host). Advances in cloud management (like Kubernetes evolving to manage 1000s of nodes) will also expand the control plane capacity, raising the bar for when that plane fails.

Finally, a key recommendation for both practitioners and researchers is to treat scalability as a multi-faceted quality. Rather than framing scalability as a binary property of a microservice architecture, scalability should be analyzed as a set of bounded stability regimes across distinct operational planes. The notion of a data plane is used here in a broad sense to refer to how much throughput or user load a microservice system can sustain while still processing data correctly and within acceptable time limits. This aspect is typically the most visible, since performance degradation often appears first as increased latency or backlog. Transaction-related behavior introduces a different set of constraints. When concurrency increases or transactions span multiple services, maintaining consistency and isolation becomes more difficult and does not always degrade in a predictable way. In addition to these concerns, overall system size also matters. As deployments grow in terms of service count and telemetry volume, orchestration and observability mechanisms consume additional resources, and at some point this control-plane overhead can begin to affect normal application behavior.

By identifying these thresholds (through benchmarks like Henning & Hasselbring's UC1–UC4 tests or through chaos engineering experiments that push the system), practitioners can plan capacity and improvements proactively. Future research could develop automated phase transition detectors that monitor for signs of impending instability in each plane- e.g., algorithms that detect when queue growth accelerates or when monitoring lag increases, and then alert or trigger adaptive responses.

In conclusion, microservices can be made to scale – but it requires holistic management of throughput, consistency, and observability. The architecture itself is no silver bullet; it is the engineering around these three planes that determines how far a given microservice-based system can go without hitting a wall. As systems grow ever more complex (edge computing, IoT, and 5G scenarios will push microservices to new extremes of distribution), acknowledging and addressing these technical limitations will be crucial. The community should strive for solutions that extend the stable operating region of all three planes, ensuring that microservices-based systems remain robust and efficient even at massive scale.

References

[1] Čilić, I., Krivić, P., Podnar Žarko, I., & Kušek, M. (2023). Performance evaluation of container orchestration tools in edge computing environments. *Sensors*, *23*(8), 1-23. https://doi.org/10.3390/s23084008

[2] Daraghmi, E., Zhang, C.-P., & Yuan, S.-M. (2022). Enhancing saga pattern for distributed transactions within a microservices architecture. *Applied Sciences*, *12*(6242), 1-24. https://doi.org/10.3390/app12126242

[3] Elkhatib, Y., & Poyato, J. P. (2023). An evaluation of service mesh frameworks for edge systems. In Proceedings of the 6th International Workshop on Edge Systems, Analytics and Networking (EdgeSys '23) (pp. 19–24). ACM. https://doi.org/10.1145/3578354.3592867

[4] Erdei, R., & Toka, L. (2023). Minimizing resource allocation for cloud-native microservices. *Journal of Network and Systems Management*, *31*(2), 1-18. https://doi.org/10.1007/s10922-023-09726-3

[5] Giamattei, L., Guerriero, A., Pietrantuono, R., Russo, S., Malavolta, I., Islam, T., Dînga, M., Koziolek, A., Singh, S., Armbruster, M., Gutierrez-Martinez, J. M., Caro-Alvaro, S., Rodriguez, D., Weber, S., Henss, J., Vogelin, E. F., & Panojo, F. S. (2024). Monitoring tools for DevOps and microservices: A systematic grey literature review. *Journal of Systems and Software*, *208*, 1-24. https://doi.org/10.1016/j.jss.2023.111906

[6] Henning, S., & Hasselbring, W. (2024). Benchmarking scalability of stream processing frameworks deployed as microservices in the cloud. *Journal of Systems and Software*, *208*(111879), 1-17. https://doi.org/10.1016/j.jss.2023.111879

[7] Lungu, S. and Nyirenda, M. (2024) Current Trends in the Management of Distributed Transactions in Micro-Services Architectures: A Systematic Literature Review. *Open Journal of Applied Sciences*, *14*, 2519-2543. https://doi.org/10.4236/ojapps.2024.149167

[8] Senjab, K., Abbas, S., Ahmed, N., & Khan, A. U. R. (2023). A survey of Kubernetes scheduling algorithms. *Journal of Cloud Computing*, *12*(87). 1-26. https://doi.org/10.1186/s13677-023-00471-1

[9] Söylemez, M., Tekinerdogan, B., & Tarhan, A. K. (2022). Challenges and solution directions of microservice architectures: A systematic literature review. *Applied Sciences*, *12*(5507), 1-40. https://doi.org/10.3390/app12115507

[10] Velepucha, V., & Flores, P. (2023). A survey on microservices architecture: Principles, patterns and migration challenges. *IEEE Access*, *11*, 88339–88358. https://doi.org/10.1109/ACCESS.2023.3305687