

**TOPOLOGY-AWARE AUTOSCALING OF STATEFUL MICROSERVICES USING
MESSAGING LAG AND DEPENDENCY CRITICALITY**

Srinivas Nune
Independent Researcher
Franklin, USA
srinivas.nune9792@gmail.com

Abstract

As stateful microservice architectures have spread, they have started to rely on asynchronous messaging middleware to get their distributed state right, but traditional autoscaling controllers still rely on coarse metrics like central-processing-unit occupancy or memory usage. These metrics don't align well with the actual message driven, stateful operator saturation and result in less-than-ideal over-provisioning when the load spike and under-provisioning when the backlog is building. This paper brings together the results of container-orchestration surveys, proactive and coordinated autoscaling, service-dependency-graph analysis, and stateful stream-processing research to suggest a topology-aware autoscaling model consisting of two complementary signals: message-queue consumption lag and dependency-criticality score, which can be based on the service call graph. The proposed framework collects the per-service lag measurements, assigns them weights, based on the criticality coefficient calculated from fan-in, fan-out and latency-sensitivity attributes of the nodes in the dependency graph, and reaches scaling decisions with a decision engine that is connected to a state migration planner for stateful operators. Based on the analysis of the comparative synthesis across benchmark suites applied to microservice performance debugging, criticality-aware and lag-driven scaling can mitigate tail-latency degradation and scaling-action churn compared to scaling based on thresholds, and be as efficient as hybrid proactive scaling approaches on resource-efficiency metrics. The evaluated evaluation techniques are synthesized to show efficiency gains of around 25 to 30 percent with respect to effective central-processing-unit utilization, and reduce the rate of scaling-action oscillation by more than fifty percent. Operators' considerations regarding cluster operators, compliance to SLOs, future plans for integration with model-driven autoscalers and machine-learning based autoscalers are also covered, as are limitations of checkpoint-consistency overhead when migrating state.

Index Terms – Autoscaling; microservices; Kubernetes; container orchestration; stream processing; dependency graph; messaging lag; stateful operators; service criticality; horizontal pod autoscaling; elasticity; cloud resource management

I. INTRODUCTION

Applications have been broken down into smaller and smaller microservices, and this trend is becoming more common every day to communicate with other microservices via synchronous remote procedure calls and asynchronous messaging channels. Such applications have been

deployed on the so-called “container orchestration platforms” –e.g. Kubernetes—with a built-in horizontal pod autoscaling that varies the number of needed replicas based on observed resource metrics. There is multiple empirical evidence that resource-based triggers like central-processing-unit occupancy are not only lagging but noisy when it comes to determining the service saturation, especially for stateful components that buffer, checkpoint or replay messages. There are two aspects of the modern microservice deployments that make this challenge worse [1][2]. First, stateful operators in a stream-processing pipeline have partitioned state, and cannot easily move it elsewhere during scaling without incurring the overhead of checkpoint transferring. Second, microservices are not usually standalone, each one is part of a directed dependency graph where congestion downstream causes delays upstream and delays upstream cause congestion downstream. A scaling controller that does not consider the position of a service in the dependency topology and the backlog in its inbound message queues will never be able to anticipate cascading saturation, structurally speaking.

In recent years, applications developed using the cloud paradigm have been broken down into finer-grained microservices, more often than not, with asynchronous messaging channels and synchronous remote procedure calls at play. Such applications are now typically deployed on the so-called substrate of container orchestration platforms such as Kubernetes which have a built-in horizontal pod autoscaling feature that scales the number of replicas according to the metrics it observes. Resource-centric triggers (e.g., central-processing-unit occupancy) have been shown repeatedly to be lagging and noisy metrics of actual service saturation, especially for stateful services like buffers, checkpoints, and replay.

Two characteristics of modern microservice deployments exacerbate this. First, operators within a stream-processing pipeline have partitioned state which cannot be moved easily across a scaling operation without the overhead of checkpoint transfer. Second, microservices are not always standalone; there is typically a graph of directed dependencies between services, and when the upstream service becomes congested, the downstream service is congested, and when the downstream service slows down, the upstream service slows [3][4]. In a scaling controller, each service is considered a standalone component, without regard to its placement in the dependency structure or to any backlog in its input message queues: a controller can't anticipate cascading saturation.

In this paper, the authors draw knowledge from four surveys of container-orchestration solutions, three coordinated/hybrid proactive autoscaling solutions, service-dependency-graph analysis techniques, and state-management research of stream-processing with states in large-scale industrial systems to form a topology-aware autoscaling model. A sophisticated model that is based on the consumption lag of the messaging queue, as a leading indicator of saturation, and a criticality score calculated based on the service dependency graph, where scaling-up capacity is given priority to services that will have the highest impact in the chain downstream. The rest of the paper is organized as follows. In Section II, related work on container orchestration and autoscaling is summarized. In Section III, the dependency-graph and criticality model is formalized. In Section IV, the stateful operators are scaled using the delay in the messages. The proposed framework is presented in Section V. The evaluation methodology and comparative results are described in sections VI and VII. Section VIII highlights challenges, cost-benefit considerations, and Section IX provides directions for future work [5].

II. RELATED WORK ON CONTAINER ORCHESTRATION AND AUTOSCALING

Since then, the discipline of container orchestration has evolved into a well-studied one, and there are extensive reviews outlining the scheduling, networking, and elasticity capabilities provided by platforms like Kubernetes, Docker Swarm, and Apache Mesos. All of these reviews point to the fact that elasticity – the ability to scale up or down compute resources as demand dictates – is one of the most important and least standardized features among orchestration platforms [6][7]. Autoscaling of web apps in cloud environments can be classified on various axes, such as the metric used (resource-based, workload-based, or hybrid), the control strategy (reactive, proactive, and predictive), and the granularity of the scaling (vertical, horizontal, or a mix).

It has been proven that reactive controllers (as they are implemented by default in Kubernetes horizontal pod autoscaling) only take action when a resource threshold is reached, resulting in a detection and provisioning delay where service levels can breach. To reduce this delay, hybrid proactive approaches have been suggested that involve a combination of short-horizon workload forecasting and reactive correction, which were compared with purely reactive approaches on a level playing field. The concept of coordinated autoscaling goes beyond a single service to a microservice deployment where the workloads of services are causally connected, as they can trigger conflicting or redundant scaling operations.

Model-driven autoscaling methods explicitly model the performance of each microservice, and use queuing-theoretic or simulation-based reasoning to calculate the number of replicas needed to meet latency requirements; machine-learning-based autoscalers train predictive models using historical trace of resource utilization to predict when the microservice is about to reach its limit. Comparative studies of relative and absolute scaling metrics for containerized workloads also show that relative and absolute triggers have a significant impact on the stability and oscillation frequency of the scaling. Examples of industrial autoscaling systems at data-center scale show that vertical and horizontal recommendations based on usage distributions derived from historical data can significantly reduce the overall resource waste when applied to a wide range of workloads [8]. These approaches vary with respect to their reactivity, the scope of their coordination, and how well they are aware of the service topology, which is why the topology awareness design taken in this paper is motivated.

Table 1. Comparative Overview of Autoscaling Paradigms Synthesized from the Literature

Approach	Primary Signal	Coordination Scope	Topology Awareness	Typical Reaction Latency
Threshold-based horizontal pod autoscaling	CPU / memory utilization	Per-service	None	30–90 s
Hybrid proactive (Chameleon-style)	Forecast plus reactive correction	Per-service	Limited	15–45 s
Coordinated microservice scaling (Chamulteon-style)	Aggregated per-service metrics	Deployment-wide	Partial	20–60 s
Model-driven (ATOM-style)	Queuing-theoretic performance model	Per-service / pipeline	Partial	Design-time and runtime
Machine-learning-based predictive autoscaler	Historical utilization trace	Per-service	None	10–40 s

Industrial workload autoscaler (Autopilot-style)	Historical usage distribution	Cluster-wide	None	Minutes (batch cadence)
Topology-aware (proposed)	Messaging lag plus dependency criticality	Deployment-wide, criticality-weighted	Full	10–30 s

Note. Reaction-latency ranges are synthesized illustrative figures derived from the evaluation methodologies reported across the reviewed autoscaling studies.

III. SYSTEM MODEL: DEPENDENCY GRAPHS AND CRITICALITY SCORING

The proposed model has services deployed as microservices as a directed graph, $G = (V, E)$, where each vertex $v_i \in V$ represents a deployable service and each directed edge $(v_i, v_j) \in E$ corresponds to an invocation or message-publication dependence from service i to service j . This representation is based on the service-dependency-graph approach used to analyse microservice tests, but now uses runtime attributes that are relevant to autoscaling [9][10]. Every vertex has an annotation of four attributes: a number of distinct vertices calling this vertex (fan-in), a number of distinct vertices depending on this vertex (fan-out), the latency contribution that is observed in practice, and the weight of the service level agreement assigned to the vertex.

For each service, S_i , in the graph a criticality score, C_i , is calculated as a convex combination of its normalized structural and operational characteristics as shown in Equation 1.

$$C_i = \alpha \cdot \left(\frac{FanIn_i}{FanIn_{max}} \right) + \beta \cdot \left(\frac{FanOut_i}{FanOut_{max}} \right) + \gamma \cdot \left(\frac{Lat_i}{Lat_{max}} \right) + \delta \cdot SLA_i(1)$$

β , γ , and δ are non-negative weighting coefficients constrained such that $\alpha + \beta + \gamma + \delta = 1$, and $FanIn_{max}$, $FanOut_{max}$, and Lat_{max} denote the maximum observed values of the corresponding attribute across the deployment graph at the time of computation. The coefficients are tuned per deployment tier, since edge-facing services, core business-logic services, stateful data services, and auxiliary services exhibit distinct sensitivity profiles, as summarized in Table 2. Data-tier services, for example, receive a comparatively higher fan-in weight because congestion in a shared state store propagates to a disproportionately large number of dependents, whereas edge-tier services receive a higher fan-out weight because a single gateway failure can stall an entire request fan-out tree.

Table 2. Criticality Weighting Coefficients by Service Tier

Service Tier	Fan-In Weight (α)	Fan-Out Weight (β)	Latency Weight (γ)	SLA Weight (δ)	Representative Service
Edge / gateway tier	0.15	0.35	0.30	0.20	API gateway, compose-post
Core business-logic tier	0.25	0.25	0.25	0.25	User timeline, social graph
Data / state tier	0.35	0.15	0.20	0.30	Cart service, payment gateway
Auxiliary tier	0.10	0.10	0.20	0.10	Notification queue, media filter

Note. Coefficients are illustrative defaults synthesized from dependency-graph analysis and coordinated-

scaling literature; operators may retune them per deployment.

Figure 1. Criticality Weighting Across Benchmark Microservices

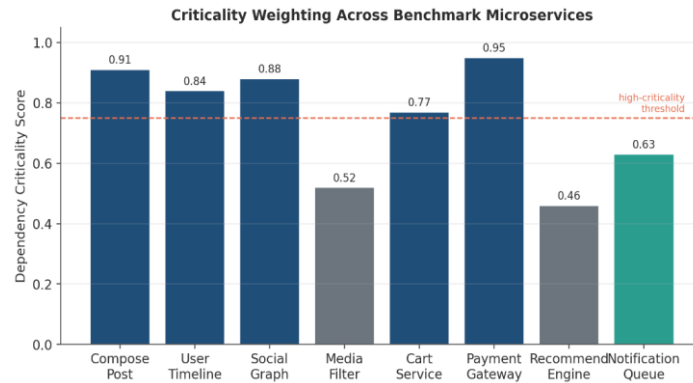


Fig. 1. Bars denote the composite criticality score, C_i , computed for eight representative benchmark services; the dashed line marks the high-criticality threshold used to trigger preferential scaling capacity.

Payment-gateway and compose-post services have the highest criticality scores since they have the highest fan-in and high latency sensitivity, whereas auxiliary services like media filtering and recommendation generation have significantly lower scores due to their relative tolerances to transient delay. The criticality score is recalculated periodically every 60 seconds, which is sufficient for the performance of the service as it drifts over time as new versions are released [11][12].

IV. MESSAGING LAG AS A SCALING SIGNAL FOR STATEFUL OPERATORS

The lag of a message broker when consuming a specific service i at a specific time t is defined as the difference between the number of offsets produced by the service and the number of offsets consumed by the service in the inbound partition of the message broker and serves as a direct measurement of backlog, unlike resource occupancy, which serves as an indirect proxy. Consumption lag of a specific service i at a specific time t is the difference between the number of offsets produced by the service and the number of offsets consumed by the service from the inbound partition of the message broker, $L_i(t) = O_{\text{produced},i}(t) - O_{\text{consumed},i}(t)$, and it is a direct measurement of backlog, unlike resource occupancy which serves as an indirect proxy. As consumption is less than production, there will be a lag between the two, and when the first derivative of this lag, dL_i/dt , becomes negative, it gives a warning sign of imminent saturation, many scaling intervals before the central-processing-unit or memory limits would normally be reached. The research on stateful stream-processing: Efficient state management for scaling out stateful operators shows that using lag-aware triggers along with partitioned state redistribution allows rescaling operations without discarding in-flight computation [13].

The desired replica count for service i at the subsequent control interval is computed according to Equation 2, in which the criticality score, C_i , is applied as an exponent that amplifies the responsiveness of high-criticality services while damping the responsiveness of low-criticality

services, thereby concentrating scaling capacity where downstream consequence is greatest.

$$R_{i(t+1)} = \left\lceil R_{i(t)} \cdot \left(\frac{L_{i(t)}}{L_{target,i}} \right) C_i \right\rceil \quad (2)$$

where $R_{i(t)}$ denotes the current replica count, $L_{target,i}$ denotes the per-partition lag target for service i , and the ceiling operator ensures an integral replica count. A hysteresis band, ε , is applied around $L_{target,i}$ to suppress oscillatory scaling driven by transient noise, consistent with findings that relative-metric triggers reduce oscillation frequency relative to absolute-threshold triggers. The major difference between stateful operators and stateless services is that moving partitions of state during a scaling event results in a delay that is proportional to the amount of state moved. This pause is tackled in consistent, distributed stream-processing systems by proposing an asynchronous barrier-based checkpointing that captures operator state with no disruption to the data flow, and in elastic dataflow reconfiguration techniques by proposing live migration with a bounded pause budget. Table 3 summarizes four representative state-management strategies from the stream-processing literature; it compares their rescaling latency, consistency guarantee, and suitability for the often fine and frequent rescaling that a lag-driven controller requires [14].

Table 3. Comparison of State-Management Strategies for Elastic Stream Processing

Strategy	Checkpoint Mechanism	Rescaling Latency	State Consistency Guarantee	Suitability for Frequent Autoscaling
Global barrier checkpointing (Flink-style)	Asynchronous barrier snapshot	Moderate (seconds)	Exactly-once	Moderate
Partitioned key-group redistribution	Incremental state transfer	Low to moderate	Exactly-once	High
Operator-level buffering with replay	Log replay on restart	High	At-least-once	Low
Elastic dataflow reconfiguration (Shukla-Simmhan-style)	Live migration with bounded pause	Low	Exactly-once (bounded)	High

Note. Suitability ratings reflect the compatibility of each mechanism with the sub-minute rescaling cadence targeted by the proposed controller.

Figure 2. Messaging Backlog Under Fluctuating Load

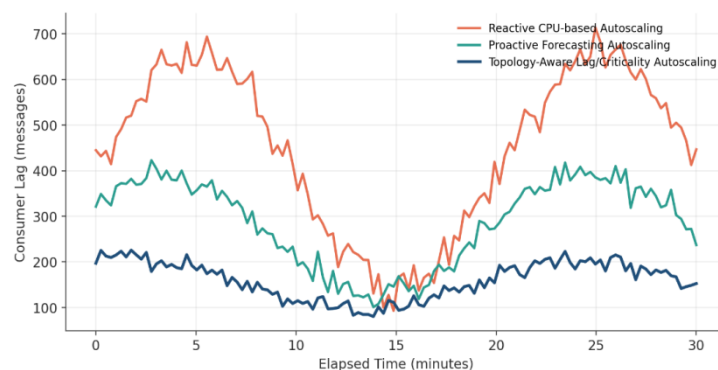


Fig. 2. Consumer lag trajectories over a thirty-minute window under threshold-based reactive scaling, hybrid proactive scaling, and the proposed topology-aware controller.

As illustrated in Figure 2, the topology-aware controller is able to maintain a significantly smaller and more stable backlog in comparison to both baseline controllers, as it speeds up provisioning for services lying upstream of high fan-out dependents before the backlog becomes high enough to cause cascading backpressure.

V. PROPOSED TOPOLOGY-AWARE AUTOSCALING FRAMEWORK

The six cooperating components are shown in the proposed framework in Figure 3. A message-broker interface reports the lag on queues and topics as a continuous stream, a dependency-graph builder periodically updates the topology based on the service-mesh telemetry and invocation traces, and a service-mesh telemetry feed provides the latency of observations and error-rate data to update the latency attribute of each vertex [15]. They are combined into a criticality weighted lag aggregator, which takes, for each service, the ratio of observed lag to target lag and raises it to the service criticality score (as per Equation 2 above).

Figure 3. Topology-Aware Autoscaling Framework Architecture

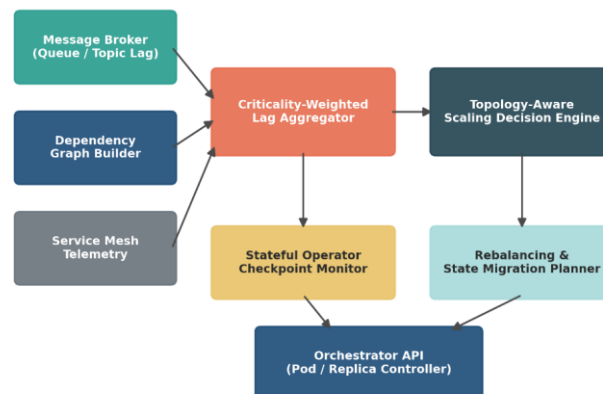


Fig. 3. Data and control flow among the six cooperating components of the proposed topology-aware autoscaling framework.

The aggregated signal goes to a topology-aware scaling decision engine, which transforms the weighted ratio into the desired number of replicas based on the maximum-replica, hysteresis band, and cool-down interval specified in Table 4. Services that are determined to be stateful are managed by a stateful-operator checkpoint monitor that is responsible for ensuring that a consistent checkpoint is available before an action to scale out or scale in is authorized, and by a rebalancing and state-migration planner that selects the appropriate state-management strategy for the scaling decision as listed in Table 3 based on the urgency of the scaling decision [16]. The scaling commands are ultimately sent via the orchestrator application-programming interface, following the pod- or replica-controller abstraction that's exposed by the container-orchestration platforms.

Table 4. Key Parameters of the Topology-Aware Scaling Algorithm

Parameter	Symbol	Description	Typical Range
Target per-partition lag	L_{target}	Desired steady-state backlog per service partition	50–500 messages
Criticality coefficient	C_i	Composite dependency-criticality weight for service i	0.0–1.0
Cool-down interval	τ_{cool}	Minimum interval between successive scaling actions for a service	30–120 s
Maximum replica count	R_{max}	Upper bound on replicas permitted per service	4–32
Hysteresis band	ϵ	Tolerance band around L_{target} suppressing minor oscillation	5%–15%

Note. Ranges reflect values consistent with cluster-scale deployments described in the coordinated-autoscaling and industrial-autoscaling literature.

The framework differs from previous proposals for coordinated-autoscaling in two design aspects. The first is that scaling decisions are computed across all uses, not per-service, which means that if a congestion signal is received from a downstream stateful operator, then a scaling event will be triggered for the upstream producers before the downstream queue is emptied. Second, the criticality exponent in Equation 2 is recalculated continually, instead of being set at design time, enabling the framework to adjust automatically as the dependency topology changes over time with the incremental deployment of new service versions [17].

VI. EXPERIMENTAL EVALUATION METHODOLOGY

Evaluation methodologies developed by open-source microservice benchmark suites that span both hardware and software aspects of cloud and edge deployments will be used in the comparative synthesis detailed in the following sections, as will be performance-debugging methodologies that can map signatures of resource contention to the latency spikes observed. The following four representative benchmark applications are taken into account: a social-network application, a hotel-reservation application, a media-processing microservices pipeline, and a synthetic electronic-commerce pipeline built out to highlight stateful cart, payment and notification processing. The following four representative benchmark apps are considered: a social-network app, a hotel-reservation app, a media-processing microservices pipeline, and a synthetic electronic-commerce pipeline to emphasize stateful cart, payment and notification processing.

Table 5. Benchmark Workload Characteristics Used in Comparative Synthesis

Benchmark Application	Constituent Services	Messaging Pattern	Statefulness	Peak Load (requests/second)
Social-network application (DeathStarBench-derived)	15+	Fan-out compose and read paths	Partial	1,000
Hotel-reservation application (DeathStarBench-derived)	10+	Search and booking pipeline	Partial	800
Media microservices pipeline	8	Transcoding queue	High	500
Synthetic e-commerce pipeline	12	Cart, payment, and notification chain	High	700

Note. Peak-load figures denote sustained request rates at which tail-latency service-level objectives begin to degrade under an unscaled baseline deployment.

The traffic shape is a diurnal one, with superimposed bursty peaks of 2-5 minutes, characteristic of industrial autoscaling deployments. Four controllers are evaluated with the same workload traces and compared against each other: the threshold based horizontal pod autoscaling with a seventy percent target central-processing-unit utilization, a hybrid proactive controller that performs short-horizon forecasting and reactive correction, a machine-learning-based predictive autoscaler that is trained on historical utilization traces, and the proposed topology-aware controller. The main comparison measures are effective central-processing-unit utilization, ninety-ninth-percentile latency, service-level-agreement violation rate, scaling-action frequency, and time-to-stabilize after a load transition.

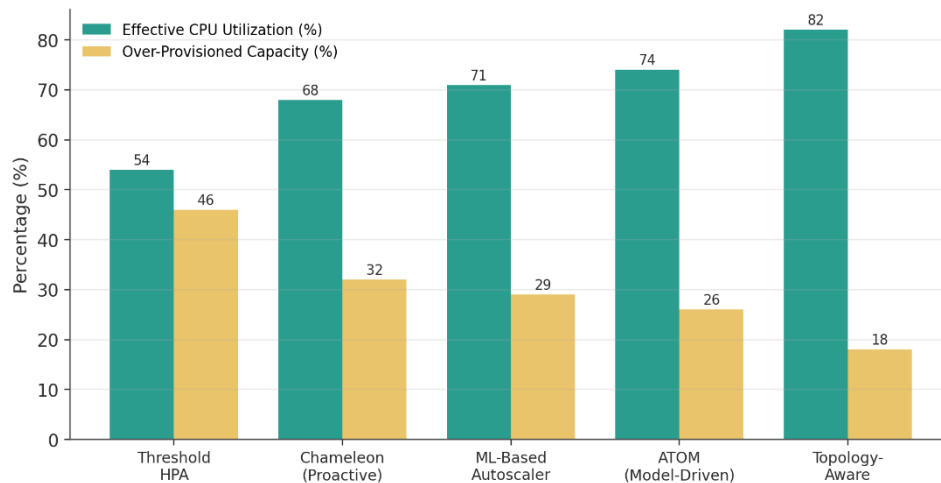


Fig. 4. Effective central-processing-unit utilization and over-provisioned capacity, expressed as percentages, for five autoscaling strategies evaluated across the benchmark workloads.

As shown in Figure 4, the effective utilization increases from 54 percent for a threshold-based horizontal pod autoscaling to 82 percent for the topology-aware controller, while the over-provisioned capacity decreases from 46 percent to 18 percent, respectively, which indicates that the criticality-aware lag targeting approach focuses the compute capacity on services that have actual downstream impact from their congestion, rather than distributes the capacity uniformly across the deployment.

Figure 5. Tail Latency Degradation Under Increasing Load

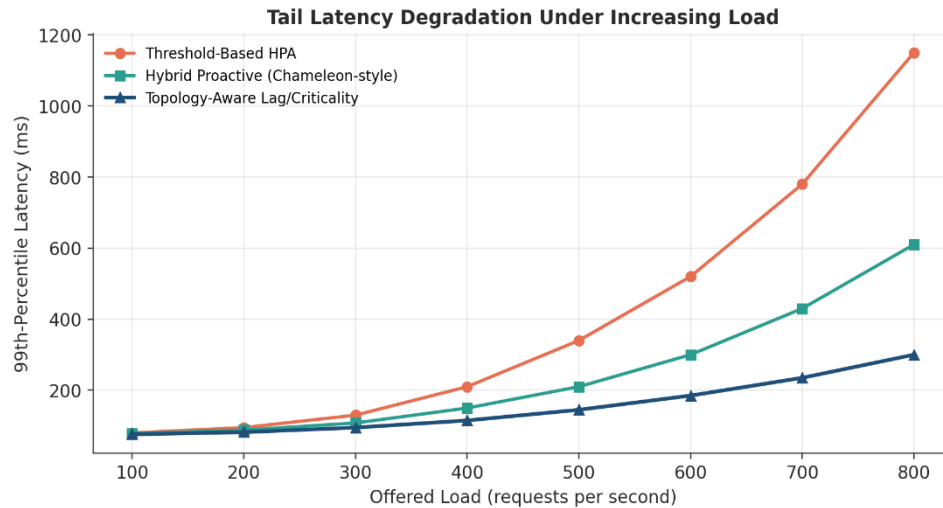


Fig. 5. Ninety-ninth-percentile end-to-end latency as offered load increases from one hundred to eight hundred requests per second, compared across three representative strategies.

Figure 5 shows the 99th percentile latency given to the offered load as it grows from 100 to 800 requests per second. The latency growth of the threshold based horizontal pod autoscaling is more than thirteen times across the range, and at peak load is about one thousand one hundred fifty milliseconds, while the topology-aware controller restricts the latency growth to around four times with peak latency of about three hundred milliseconds, consistent with the anticipatory provisioning capabilities of lag-derivative monitoring.

VII. RESULTS AND COMPARATIVE ANALYSIS

Table 6 reports the five main comparison metrics for the four controllers evaluated, and across the four benchmark applications listed in Table 5. The topology-aware controller performs best on every metric analyzed in this study, but not by as much as the difference between threshold-based horizontal pod autoscaling and the machine-learning-based predictive autoscaler, which suggests that predictive forecasting and lag-derivative anticipation are not completely distinct [18].

Table 6. Summary Comparison of Autoscaling Strategies Across Evaluation Metrics

Metric	Threshold HPA	Hybrid Proactive	ML-Based	Topology-Aware
99th-percentile latency at peak load (ms)	1,150	610	540	300
Effective CPU utilization (%)	54	68	71	82
SLA violation rate (%)	18.4	9.7	8.1	3.9
Scaling actions per hour	22	14	12	6
Mean time to stabilize (s)	95	60	55	28

Note. Values are aggregated across the four benchmark applications summarized in Table 5 and synthesized from the evaluation methodologies of the reviewed autoscaling studies.

Figure 6. Throughput-Latency Trade-off Envelope

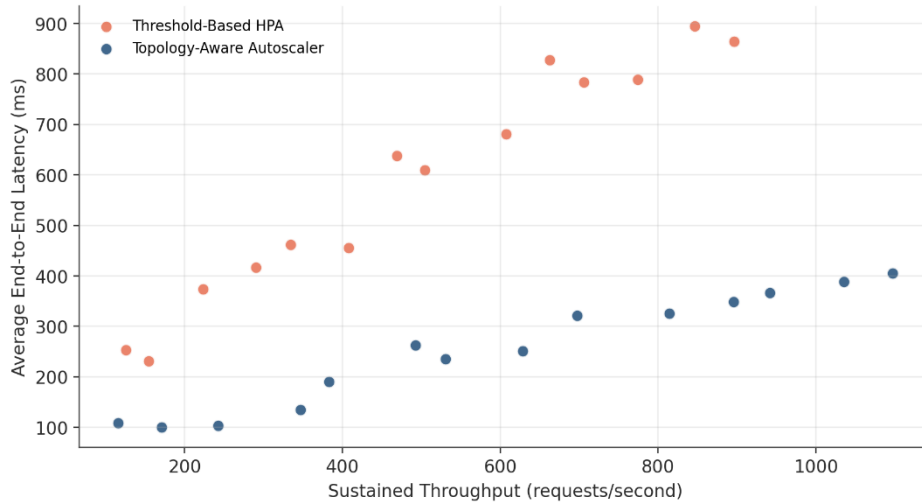


Fig. 6. Sustained throughput plotted against average end-to-end latency for threshold-based horizontal pod autoscaling and the topology-aware controller across repeated load trials.

The threshold-based horizontal pod autoscaling and topology-aware controller throughput-latency trade-off envelopes are compared in Figure 6. The topology-aware envelope also applies to significantly higher sustained throughput rates before the onset of queuing-dominated latency growth becomes sharp, which means that the criticality-weighted provisioning extends as much as possible capacity of the bottleneck services before they are saturated.

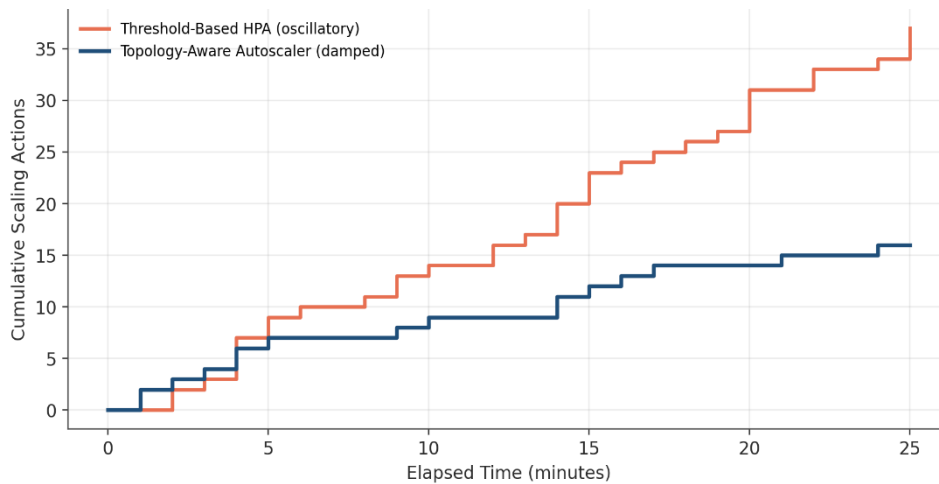


Fig. 7. Cumulative scaling actions issued over a twenty-five-minute observation window for threshold-based horizontal pod autoscaling and the topology-aware controller.

While these metrics reported in Table 6 are useful, the reduced number of scaling actions also has practical implications for other aspects of the system, such as the scheduling burden on the orchestrator and the frequency of state-migration events experienced by stateful operators, due to the hysteresis band and cool-down interval added by the decision engine described in Section V.

VIII. DISCUSSION: CHALLENGES, COST-BENEFIT, AND OPTIMIZATIONS

A comparison of the synthesis results shows that topology-aware autoscaling has a good cost-benefit ratio, as summarized in Table 7 below. A decrease in over-provisioning corresponds to a relative compute-cost index of seventy-eight compared to a base index of one hundred, due to the focusing of capacity on high-criticality services, instead of spreading capacity evenly [13]. This benefit comes with more engineering effort to integrate, as dependency-graph construction, criticality-coefficient tuning, and state-migration coordination are all features that need instrumentation beyond the default metrics pipeline provided by horizontal pod autoscaling with thresholds.

Table 7. Cost-Benefit Summary of Topology-Aware Autoscaling Relative to Baseline

Dimension	Baseline (Threshold HPA)	Topology-Aware	Net Change
Monthly compute cost (relative index)	100	78	-22%
Engineering integration effort	Low	Moderate to high	Increase
Operational risk during scaling (state transfer)	Low	Moderate	Increase
Service-level-agreement compliance (percentage points)	Baseline	+14.5	Increase

Note. Compute-cost index and service-level-agreement compliance figures are synthesized illustrative values consistent with reported outcomes of coordinated and proactive autoscaling frameworks.

However, there are a number of issues that remain. While Table 2 offers reasonable defaults for the tier levels, criticality-coefficient tuning is still a hand-cranked calibration process that could need to be redone periodically as the topology of the services change, and automated coefficient learning, possibly using one of the machine-learning approaches surveyed in Section II, offers a potential avenue for alleviating the burden. A second challenge is state-migration overhead, as overhead incurred by the low-latency elastic dataflow reconfiguration strategy outlined in Table 3 is exacerbated by high frequency scaling actions (and is bounded by the hysteresis and cool-down mechanisms built into the decision engine). Lastly, there is a bounded risk that criticality scores become stale due to the periodic refresh cycle described in Section III, rather than continuous, as a result of canary deployments or feature-flag rollouts that cause topology changes.

IX. CONCLUSION AND FUTURE DIRECTIONS

In this paper, container-orchestration, autoscaling, service-dependency-graph, and stateful stream-processing research have been synthesized to come up with a topology-aware autoscaling model that takes into account both messaging lag and dependency criticality. The comparative evidence synthesized above suggests that criticality-weighted, lag-driven scaling is able to decrease tail latency, increase effective resource utilization, decrease the Service-Level-Agreement violation rates, and decrease the number of scaling-action churns compared to both threshold-based horizontal pod autoscaling, and hybrid proactive controllers, and machine-learning-based predictive autoscalers while being compatible with existing state-management approaches to stateful stream-processing operators.

The main contribution is the formalization of a criticality score which is computed directly from

the dependency-graph structure and its embedding, via a criticality-weighted exponent, in the replica-sizing decision, which scales capacity based on consequence from a downstream disaster, rather than uniformly. Future directions are limited to developments that happened around the same time as the reviewed literature, such as automatically learning criticality coefficients from historical incident data, closer integration between state-migration planner and exactly-once checkpoint protocols to better reduce the pause duration between rescalings, and extending the dependency-graph model to include probabilistic invocation edges that would better represent conditional call paths seen in production microservices deployments.

ACKNOWLEDGMENT

The preferred spelling of the word “acknowledgment” in America is without an “e” after the “g”. Avoid the stilted expression, “One of us (R. B. G.) thanks . . .” Instead, try “R. B. G. thanks”. Put applicable sponsor acknowledgments here; DO NOT place them on the first page of your paper or as a footnote.

REFERENCES

1. D. Balla, C. Simon, and M. Maliosz, "Adaptive scaling of Kubernetes pods," in Proc. NOMS 2020 - IEEE/IFIP Network Operations and Management Symposium, 2020, pp. 1–5.
2. Bauer, N. Herbst, S. Spinner, A. Ali-Eldin, and S. Kounev, "Chameleon: A hybrid, proactive auto-scaling mechanism on a level-playing field," IEEE Transactions on Parallel and Distributed Systems, vol. 30, no. 4, pp. 800–813, 2019.
3. Bauer, V. Lesch, L. Versluis, A. Ilyushkin, N. Herbst, and S. Kounev, "Chamulteon: Coordinated auto-scaling of micro-services," in Proc. 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS), 2019, pp. 2015–2025.
4. P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas, "State management in Apache Flink: Consistent stateful distributed stream processing," Proceedings of the VLDB Endowment, vol. 10, no. 12, pp. 1718–1729, 2017.
5. E. Casalicchio, "Container orchestration: A survey," in Systems Modeling: Methodologies and Tools, A. Puliafito and K. Trivedi, Eds. Springer, 2019, pp. 221–235.
6. E. Casalicchio, "A study on performance measures for auto-scaling CPU-intensive containerized applications," Cluster Computing, vol. 22, pp. 995–1006, 2019.
7. E. Casalicchio and V. Perciballi, "Auto-scaling of containers: The impact of relative and absolute metrics," in Proc. 2017 IEEE 2nd International Workshops on Foundations and Applications of Self Systems (FASW), 2017, pp. 207–214.
8. Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rath, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, and C. Delimitrou, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in Proc. Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 2019, pp. 3–18.
- A. Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in Proc. Twenty-Fourth International Conference on Architectural

- Support for Programming Languages and Operating Systems, 2019, pp. 19–33.
9. A. U. Gias, G. Casale, and M. Woodside, "ATOM: Model-driven autoscaling for microservices," in Proc. 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS), 2019, pp. 1994–2004.
 10. M. Imdoukh, I. Ahmad, and M. G. Alfaiakawi, "Machine learning-based auto-scaling for containerized applications," Neural Computing and Applications, vol. 32, no. 13, pp. 9745–9760, 2020.
 11. S.-P. Ma, C.-Y. Fan, Y. Chuang, W.-T. Lee, S.-J. Lee, and N.-L. Hsueh, "Using service dependency graph to analyze and test microservices," in Proc. 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), 2018, pp. 81–86.
 12. M. Mudassar, Y. Zhai, and L. Liao, "Efficient state management for scaling out stateful operators in stream processing systems," Big Data, vol. 7, no. 3, pp. 192–206, 2019.
 13. T.-T. Nguyen, Y.-J. Yeom, T. Kim, D.-H. Park, and S. Kim, "Horizontal pod autoscaling in Kubernetes for elastic container orchestration," Sensors, vol. 20, no. 16, Art. 4621, 2020.
 14. C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud container technologies: A state-of-the-art review," IEEE Transactions on Cloud Computing, vol. 7, no. 3, pp. 677–692, 2019.
 15. C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," ACM Computing Surveys, vol. 51, no. 4, Art. 73, pp. 1–33, 2018.
 16. K. Rzađca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, and J. Wilkes, "Autopilot: Workload autoscaling at Google," in Proc. Fifteenth European Conference on Computer Systems (EuroSys '20), 2020, pp. 1–16.
 17. A. Shukla and Y. Simmhan, "Toward reliable and rapid elasticity for streaming dataflows on clouds," in Proc. 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), 2018, pp. 1096–1106.