Flare: Leveraging Serverless Elasticity to Absorb Microservice Load Spikes

Dilina Dehigama[1], Shyam Jesalpura[1], David Schall[2,*], Antonios Katsarakis[3,*],
Marios Kogias[4], Rakesh Kumar[5], Boris Grot[1]
[1]University of Edinburgh, UK [2]TU Munich, Germany [3]Huawei Research, UK
[4]Imperial College London, UK [5]NTNU, Norway

***Abstract*—Online services are commonly deployed as chains of microservices on virtual machines (VMs). VM deployments are cost-effective under steady load, but struggle with unexpected load spikes due to slow scale-out. In contrast, serverless functions (FaaS) can rapidly absorb load spikes; however, continuously running a microservice workload entirely on serverless can be prohibitively expensive.**

We propose Flare, a hybrid microservice architecture that combines VMs with serverless computing. Flare keeps steady traffic on VMs and uses serverless functions only when a spike overloads specific services. During a spike, Flare detects the overloaded services and shifts only the excess traffic for those services to serverless, minimizing cost overhead. Flare integrates with existing autoscaling and serverless infrastructure, requiring only control-plane changes and zero or minimal application changes. Compared to a VM-only autoscaling baseline, Flare reduces peak tail latency by 47.9% on average, while increasing cost by just 4.4% on average.

I. Introduction

Today's user-facing online services, such as social networks, online stores, and media portals, must deliver low latency despite large changes in user demand. Many such services are built as microservices: small services that communicate through remote procedure calls. Building online services as microservices offers many benefits, including modularity, ease of development, and independent scaling.

Load changes in online services include both regular, predictable patterns and sudden, less-predictable spikes [1, 2, 3]. Regular load fluctuations are straightforward to accommodate and can be handled by provisioning resources ahead of time. In contrast, irregular changes in load may present a challenge, especially if they are sudden and if the amplitude of the spike is large. An autoscaler must first detect that a spike is non-transient, and then provision new resources before traffic can be redirected to them. In practice, the detection and provisioning steps may take minutes or even tens of minutes, during which service quality may be compromised. Netflix reports that when traffic doubled within 10 seconds, autoscaling took 5 minutes to restore the latency Service Level Objective (SLO) [4].

To avoid the slow scale-out problem, services can be over-provisioned by deploying more instances than required for a given load level. However, our analysis of a week-long Twitter trace [3] shows several, sudden & unexpected load spikes, sometimes doubling the load over a short interval. Maintaining enough standby capacity to absorb such spikes would be prohibitively expensive, since the extra resources would have to be deployed and paid for continuously, even when they are not needed. Another option is to predict traffic fluctuations and scale resources proactively. While predictive scaling can accurately anticipate regular load fluctuations, real-world load can spike unexpectedly, rendering such techniques ineffective for sudden load increases [2, 5]. In theory, deploying online services as serverless functions[1](FaaS) can absorb spikes within seconds due to fast startup and elasticity. In practice, however, running a steady microservice workload entirely on serverless is expensive. In our measurements, a representative hour-long steady segment of the Twitter trace costs $2.4\times$ more on serverless than on VM-based microservices.

This creates a fundamental conundrum: existing VM-based deployments are cost-effective but slow to scale, while serverless platforms offer rapid elasticity at high cost. Prior hybrid systems combine VMs and serverless resources, but they do not address overload at individual services inside a multi-hop microservice chain [6, 7, 8, 9, 10, 11, 12]. To bridge the gap, we propose Flare, a hybrid microservice architecture that combines VM cost efficiency with serverless elasticity. Under normal load, Flare runs services on VMs. During a spike, Flare identifies the overloaded services and routes only the portion of their traffic that the current VM instances cannot serve to serverless functions while VM autoscaling catches up. Flare shifts traffic at the level of individual microservices rather than the whole application, requires only control-plane changes and zero or minimal application changes, and integrates with existing autoscaling and serverless infrastructure.

Our evaluation shows that Flare reduces peak tail latency by 47.9% on average compared to a VM-only autoscaling baseline. Flare eliminates all median-latency SLO violations and significantly reduces tail-latency violations, while increasing cost by only 4.4% on average.

The paper makes the following contributions:

- We show that reactive and predictive autoscaling struggle with unexpected load spikes, while fully serverless deployment handles load spikes but incurs substantial cost overhead when run continuously (Section II).

[*]The author was at the University of Edinburgh when this project started.

[1]We use the term serverless to specifically refer to Function-as-a-Service (FaaS) throughout this paper.

- We present Flare, a hybrid architecture that selectively shifts traffic beyond current VM capacity from overloaded microservices to serverless functions (Section III).
- We demonstrate that Flare improves spike resilience with modest cost overhead, reducing peak tail latency by 47.9% on average and adding only 4.4% average cost (Section VI).

II. Motivation

A. Modern Online Services and Autoscaling

Modern online services have evolved into complex systems that must be scalable, highly available, and responsive while meeting tight latency SLOs [13, 14]. Large service providers such as Airbnb, Netflix, LinkedIn, Uber, and Twitter address service scalability and availability with microservice architectures [15, 16, 17, 18, 19]. A microservice application is built from small services, each responsible for a narrow function and communicating with other services over the network. Many services are designed to be stateless, with persistent state kept in external data stores, which improves scalability [20, 21].

Microservices are commonly deployed as containers on VM clusters using orchestration platforms such as Kubernetes (K8s) [22, 23, 24]. In K8s, the basic deployment and scaling unit is a *pod*, which contains one or more containers. Pods run on VMs, and one VM can host several pods.

Autoscaling in K8s happens at two levels. First, at the pod level, the Horizontal Pod Autoscaler (HPA) creates new pods inside running VMs based on metrics such as CPU utilization or requests per second [25]. Second, at the VM level, if the cluster lacks capacity for new pods, the Cluster Autoscaler (CA) [26] provisions new VMs and adds them to the cluster, enabling HPA to place additional pods inside the newly-created VMs.

B. Load Variability

The load, or request arrival rate, of a service can vary significantly due to factors such as the time of day, day of the week, or season [3]. This is evident in online services such as a social networking platform facing higher loads during the day and lower loads at night or a shopping website encountering increased demand during the holiday season.

Figure 1 illustrates the load pattern of Twitter [3] over a week-long period. We observe a mostly consistent traffic pattern characterized by predictable periodic trends with minor fluctuations for a given time of day. However, the figure also shows multiple load spikes at various points in the week without a clear periodic trend. The zoomed-in portion of Figure 1 shows one spike starting at 07:10, when the load suddenly doubles from just under 4000 requests/min to 8000 requests/min. The example emphasizes that microservice systems must handle unexpected spikes without compromising user experience.

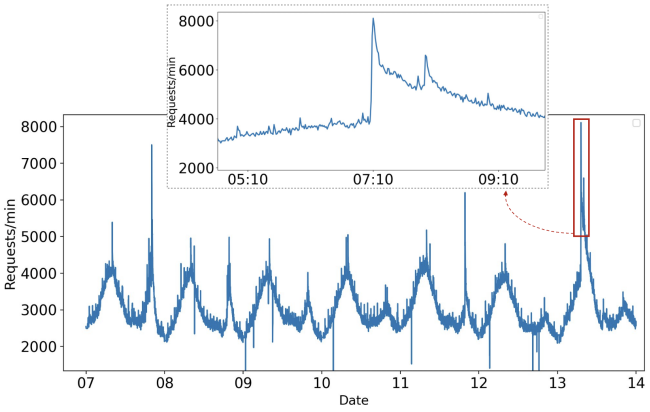

Fig. 1. Twitter load trace over one week. The highlighted region marks a four-hour interval from a day with an unexpected load spike.

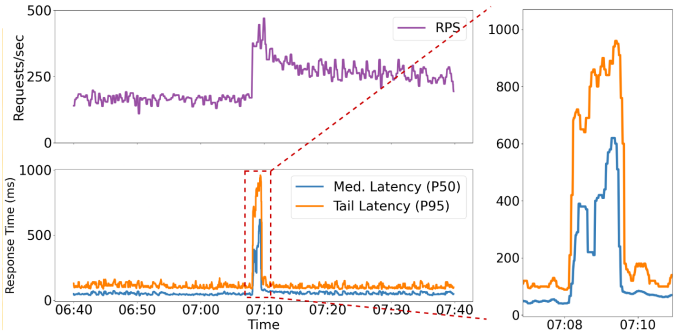

Fig. 2. Impact of a sudden load spike on a VM-based microservice application.

C. VM-based Microservices Meet Load Spikes

Unexpected load spikes pose a challenge for VM-based autoscaling due to two sources of delay: *detection lag* and *reaction lag*. Detection lag arises because autoscalers rely on metrics and thresholds to identify load spikes. Metrics such as CPU utilization or request rate are typically averaged over a time window to filter transient fluctuations and react only to sustained load increases. In common K8s deployments, detection lag is relatively small [27, 28]. Reaction lag is the time required to provision and deploy new instances, which can take minutes depending on the cloud provider and VM size [29]. During the combined lag, requests queue at existing instances and degrade user experience [30].

We quantify the impact of a sudden load spike on end-to-end latency using the BookInfo microservice application from Istio [31, 32]. We replay the highlighted one-hour load trace from Figure 1 on an AWS EKS cluster with HPA and CA enabled. Section V provides detailed information on the parameters used in this study.

Figure 2 presents the load and end-to-end latency over time. The top graph reveals an initially stable request rate around 200 RPS, then increases above 400 RPS at around 07:08. Median and tail latency remain low before the spike. During the spike, median latency peaks above 600ms and tail latency exceeds 900ms, corresponding to 8.2x and 9.6x increases

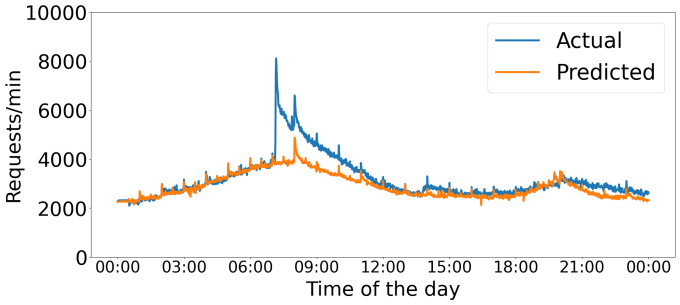

Fig. 3. Load prediction for a day with an unexpected load spike.

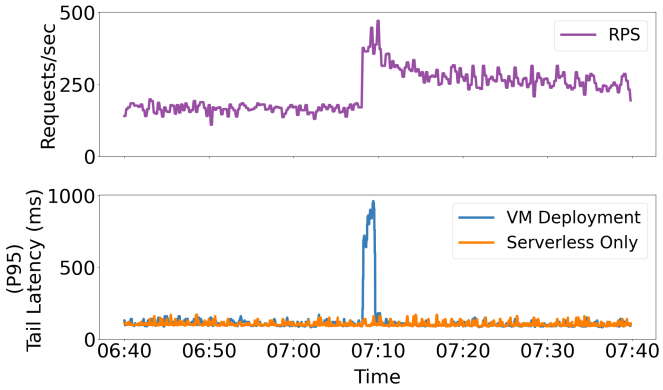

Fig. 4. Impact of a sudden load spike on tail latency (P95) on a serverless-only deployment compared to VM-based deployment.

over pre-spike latency, respectively. Latencies return to pre-spike levels just after 07:10, indicating about two minutes of combined detection and reaction time. Industry reports show similar behavior: Netflix reports that a traffic doubling within 10 seconds took 5 minutes to recover through autoscaling [4].

Service providers can avoid latency spikes by over-provisioning their clusters [33]. However, the spike in Figure 1 doubles the prior steady load. Absorbing the spike with VMs alone would therefore require roughly twice the VM capacity needed for the pre-spike load, leaving many resources idle outside the spike period and increasing cost.

D. Proactive Resource Provisioning

Instead of reacting to sudden load spikes, prior works have proposed proactive auto-scaling mechanisms that aim to predict future load patterns and provision resources ahead of time [34, 35, 36, 37, 38]. These approaches typically use machine learning models to analyze historical data and forecast future resource requirements [5, 2]. However, despite generally good accuracy, proactive auto-scaling approaches are challenged by sudden and unexpected load spikes.

To evaluate the effectiveness of proactive models, we use Seq2seq [39], a widely used machine learning model also used in a state-of-the-art proactive auto-scaling mechanism at Alibaba [2]. We train the Seq2seq model using three months of Twitter trace data to evaluate its prediction accuracy.

Figure 3 presents a day-long period with an unexpected load spike, comparing the load predicted by Seq2seq (orange line) with the actual load (blue line). During regular-load periods, specifically before 07:00 and after 12:00, Seq2seq achieves good prediction accuracy and maintains less than a 14% deviation. During the spike period, however, the predicted load deviates sharply from the actual load, with the maximum deviation exceeding 200%. The result highlights a key limitation of predictive models: they can follow regular load patterns, but miss sudden spikes that are not visible in historical trends.

E. Can Serverless Help?

Serverless computing, particularly Functions-as-a-Service (FaaS), has emerged as a widely adopted cloud deployment model for online applications due to its scalability and elasticity [40]. Serverless functions can scale from zero to thousands of active instances within seconds [41], which makes serverless a strong candidate for handling abrupt traffic increases.

To evaluate the suitability of serverless platforms for handling traffic spikes, we study the load spike highlighted in Figure 1 using the BookInfo benchmark from Section II-C. For this analysis, we deploy BookInfo fully on serverless. Figure 4 shows that tail latency in the serverless deployment remains stable during the spike, in contrast to the VM-only deployment.

To investigate the cost implications of running a microservice application entirely on serverless, we also estimate cost during a steady-load interval. For the 05:40–06:40 interval in Figure 1, estimated cost is $0.48 on serverless and $0.20 on VMs, while P95 latency remains similar (106 ms on serverless versus 113 ms on VMs). A serverless-only deployment is therefore 2.4x more expensive for this regular-load period.

Summary. VM deployments provide good cost efficiency for steady traffic but react slowly to unexpected spikes. Serverless deployments respond quickly to spikes but incur a high steady-state cost. An ideal system would combine VM cost-efficiency for steady traffic with rapid elasticity of serverless for unexpected load spikes.

III. Flare

A. Overview

Flare is a system that combines the cost efficiency of VM deployments with the responsiveness of serverless computing. The key idea is to route only the traffic that exceeds current VM capacity to serverless during load spikes, and only until newly provisioned VMs become ready. As a result, Flare reduces SLO violations during load spikes without the high steady-state cost of a fully serverless deployment or the resource overhead of maintaining large idle VM pools. Flare applies this idea at service granularity rather than at whole-application granularity. When a microservice chain contains an overloaded service, Flare shifts only requests that reach the overloaded service to serverless instances. Requests to services with sufficient VM capacity remain on VMs. Moreover, Flare's traffic-shifting decisions are opaque to clients and application

logic; neither observes whether a given request executes on a VM or a serverless instance.

To remain complementary to existing deployments, Flare separates the mechanism into three components: monitoring, load-balancing, and a control plane. The monitoring and load-balancing components rely on the default infrastructure provided by the deployment environment (K8s). The control plane adds the *Flare controller*, which reads load metrics and updates load-balancer rules when a service needs temporary serverless capacity.

B. System in Action

Figure 5 depicts how Flare responds to a spike for a single scalable microservice. In the K8s deployment, microservices run in pods, while an external load balancer, e.g., Envoy or AWS Application Load Balancer [42, 43], routes incoming traffic. The *Flare Controller* runs as a service within the cluster and continuously monitors RPS and CPU metrics ②.

During steady load, the load balancer directs all requests to pods running on VMs ①. Upon identifying a load spike, the Flare controller updates the load-balancer configuration so a fraction of requests is routed to serverless functions ③–④. This temporary traffic shift reduces pressure on the overloaded VMs while new VM capacity is being provisioned, limiting SLO violations during the scale-out interval. Importantly, Flare does not interfere with existing autoscaling mechanisms; for example, Cluster Autoscaler (CA) provisions new VMs for the increased load ⑤. After the new VMs become operational, the *Flare controller* shifts the serverless traffic back to VMs ⑥.

C. Flare Controller

The *Flare Controller* is the brain behind Flare and has two responsibilities. First, it monitors CPU and RPS metrics for all services deployed in the cluster. Second, based on observed metrics, it configures the load-balancing infrastructure to steer traffic only to VMs or to both VMs and serverless resources. Crucially, monitoring and load balancing happen for each individual service along the microservice chain.

Flare implements this split with weighted load balancing [44, 45]. Under steady load, the VM weight is 100% and the serverless weight is 0%. During a spike, the controller computes new weights from observed metrics so that the serverless weight corresponds to the excess traffic beyond current VM capacity. Once CA and HPA provision sufficient VM capacity, the controller restores the VM weight to 100%.

IV. Flare Under the Hood

A. Technologies Used

Flare is implemented on Kubernetes, with serverless functions running on Knative [46]. It uses Istio [47, 48] with Envoy sidecars [43, 49, 50] for service-to-service routing, and updates Istio Virtual Services to shift traffic between VM and serverless instances. For spike detection, Flare extends K8s CAdvisor [51] to collect pod CPU statistics every 1 second, while Prometheus [52] exposes CPU statistics and service-level RPS to the controller.

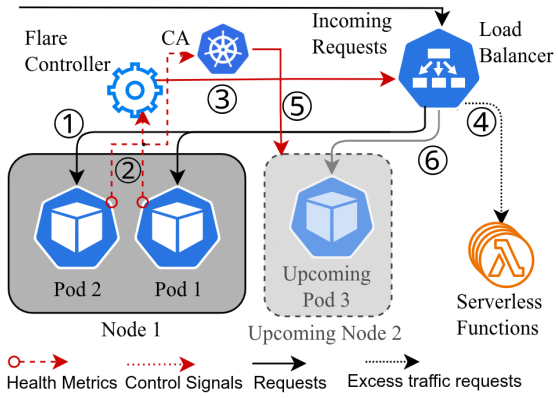

Fig. 5. Flare design overview.

B. Flare Controller

Integration with the Autoscaler. K8s autoscaling commonly uses CPU utilization to decide when additional pod and VM capacity is needed [25]. However, once Flare shifts part of a load spike to serverless, the CPU utilization measured on VMs no longer reflects the full service load. The autoscaler may then observe steady VM-side CPU utilization, preventing further VM scale-out. Flare addresses this issue by tracking total RPS for each service across both VM and serverless instances and providing this signal to the autoscaler for scale-out decisions. This integration preserves CPU-based scaling behavior while allowing VM scale-out to proceed normally, even during Flare's temporary traffic shifting to serverless.

Weight Calculation. Flare performs weighted load balancing using W_v and W_s, which denote the traffic fractions sent to VMs and serverless instances, respectively. At timestamp t, Flare computes W_v as the ratio of *service throughput capacity*, i.e., the maximum collective RPS of all active pods, to the estimated load at $t+1$. For $\Delta RPS = RPS_t - RPS_{t-1}$, the next-step load estimate is $RPS_t + \Delta RPS$ when load is increasing, and RPS_t otherwise:

$$W_v = \frac{\textit{Service Throughput Capacity}}{\begin{cases} RPS_t + \Delta RPS & \text{if } \Delta RPS > 0 \\ RPS_t & \text{if } \Delta RPS \leq 0 \end{cases}}, \quad W_s = 1 - W_v$$

C. Handling Traffic Shifts in Chained Microservices

In a microservice application, requests may traverse multiple services that form a service chain. Flare shifts traffic only at overloaded services, as described in Section III. After a request enters the serverless path, the implementation must decide whether downstream calls should return to VMs or remain in serverless. Returning to VMs can reduce cost, but it requires serverless instances to know the current load of downstream VM services or requires additional load-balancing stages after each service. Both options increase implementation complexity and may add latency. Our prototype therefore keeps a request in the serverless environment once the request enters the serverless path. The design may execute some

non-overloaded downstream services on serverless, but the evaluation shows that the added cost is small. The choice is not fundamental to Flare and can be revisited in future implementations.

D. Service Portability and Adaptation Effort

The adoption of containers as the standard unit of deployment has significantly simplified the migration of workloads between VMs and serverless environments [22, 46]. Most serverless platforms support container-image-based deployments, enabling the same image to run in both the VM-based K8s cluster and serverless functions. Although the deployment artifact remains consistent, protocol and invocation models may still differ across platforms; for example, Knative uses gRPC and HTTP, while AWS Lambda has a provider-specific invocation model [53]. Lightweight adapters, such as the AWS Lambda Web Adapter [54, 55, 56], bridge such differences without changing application logic.

V. Experimental Methodology

Load trace. Load is generated from a Twitter trace [3], using the hour-long segment containing the load spike shown in Figure 3. Since the original trace is sampled, we scale the trace to trigger CA-level autoscaling [57]. Locust [58] replays the trace from a VM with 4 vCPU cores and 16 GB of RAM in the same AWS region as the cluster.

Benchmarks. We use three microservice benchmarks: BookInfo, Online Boutique, and Hotel Reservation [32, 59, 21]. The benchmarks include multiple service chains, diverse communication protocols, and several widely used programming languages. One BookInfo service (*reviews*) is ported from Java to Go to reduce Java's high cold-start latency on serverless [60]. We replay the load trace on each application three times and report the run with the median latency across runs.

Configurations. Flare is compared against the K8s default baseline (*Baseline*), which uses an EKS cluster with default HPA and CA parameters. HPA triggers scaling when average CPU utilization reaches 50% [2]. The average RPS threshold is configured to match the RPS at 50% CPU utilization.

Cluster. The K8s cluster runs on Amazon EKS [61], with EC2 *t3.xlarge* nodes that provide 4 vCPU cores and 16 GB of RAM. Each VM costs 0.1670 USD per hour [62]. Knative deployments use dedicated VMs within the same K8s cluster. Since production serverless platforms such as AWS Lambda and Azure Functions have cold-start times about 10× lower than Knative [63], Knative functions are kept warm to approximate production serverless performance. The effect of warm and cold starts is evaluated separately on AWS Lambda in Section VI-C.

Cost Estimate. Serverless cost is estimated from the execution time and allocated vCPU of each Knative function. Because AWS Lambda ties vCPU allocation to provisioned memory and does not allow direct CPU selection, we map each Knative vCPU allocation to the Lambda memory configuration with comparable CPU capacity [64]. The mapped memory configuration is then used to calculate cost [65].

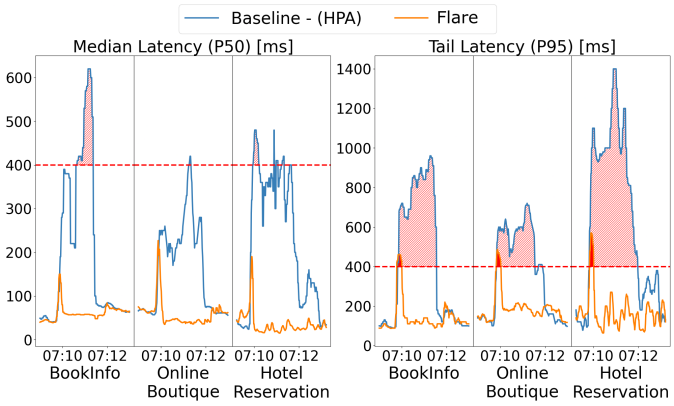

Fig. 6. Latency comparison. Red dashed line marks the 400ms SLO.

Application	Median (P50)		Tail (P95)	
	Base	Flare	Base	Flare
BookInfo	36s	–	1.45m	12s
Online Boutique	3s	–	1.6m	10s
Hotel Reservation	29s	–	2.3m	13s

TABLE I
SLO violation duration for Baseline (Base) and Flare.

VI. Evaluation

We evaluate Flare along three dimensions: latency under load spikes, cost overhead, and performance with AWS Lambda as a production serverless backend.

A. Impact on Latency

We compare the median (P50) and tail (P95) latencies of Flare against the VM-only baseline. The SLO target for the studied microservices is 400ms [66]. Since Flare is activated only during the spike, baseline and Flare latencies are similar both before and after the spike. The analysis therefore focuses on the spike interval from 7:10 to 7:12.

Figure 6 illustrates the end-to-end latency impact on median and tail latency for the three benchmarks. The VM-only baseline suffers from significant SLO violations during the spike. Median and tail latencies increase by 5.9-19.2x and 4.2-12.9x over pre-spike levels, respectively, exceeding the 400ms SLO by 5-55% and over 80% across the three applications.

With Flare, excess load is rapidly shifted to serverless instances, reducing end-to-end latency. Median latency increases by 1.6-7.6x but never violates the SLO. Tail latency increases by 2.8-5.3x for a short period and exceeds the SLO by at most 32%. Table I summarizes all SLO violations for the VM-only baseline and Flare. Flare still violates the tail SLO for 5% of requests, but the longest violation lasts only 13 seconds, compared to 1.45-2.3 minutes under the VM-only baseline.

In summary, Flare eliminates all median SLO violations and limits tail SLO violations to at most 13 seconds, while reducing peak tail latency by 32-59% (47.9% on average) compared to the VM-only baseline.

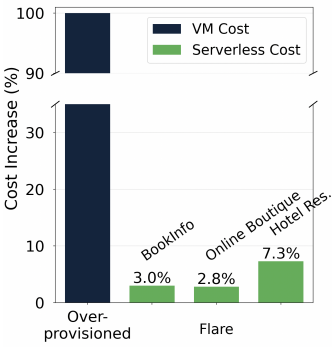

Fig. 7. Cost comparison.

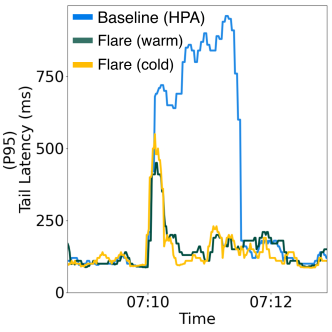

Fig. 8. Efficacy of Flare on AWS VMs & Lambda.

B. Cost Overhead

Next, we evaluate the cost overhead of using serverless capacity during the spike. Flare is compared against the VM-only baseline and an over-provisioned VM-only cluster (*over-provisioned*) with twice the steady-load VM capacity, which is sufficient to absorb the load spike. Figure 7 reports costs normalized to the VM-only baseline. Compared to the baseline, Flare increases cost by 3%, 2.8%, and 7.3% for BookInfo, Online Boutique, and Hotel Reservation, respectively. The average cost increase for Flare is 4.4% compared to the baseline, while remaining 47.8% cheaper than the over-provisioned cluster.

C. Performance with AWS Lambda

Finally, we evaluate Flare using AWS Lambda, a popular production serverless offering. The experiment uses BookInfo and the same load spike described in Section V, and studies two cases: warm Lambda instances and cold Lambda instances that are provisioned on demand.

Figure 8 shows that warm AWS Lambda instances perform similarly to the Knative-based evaluation in Section VI-A. With warm instances, Flare reduces peak tail latency by 53% and absorbs the spike after 9 seconds. With cold instances, Flare sees a 1.19x tail-latency increase compared to warm Lambda instances because Lambda must provision instances when spike traffic arrives. However, cold and warm Lambda instances recover at similar speeds: Flare returns to pre-spike latency within 10-16 seconds, compared to 1.45 minutes for the VM-only baseline.

VII. Related Work

Cloud auto-scaling. Cloud auto-scaling is commonly divided into reactive and proactive schemes [67, 68, 69, 70]. Reactive schemes, including public-cloud rule-based autoscalers [71, 72, 73], ATOM [74], and Microscaler [75], adjust resources after observing load or performance changes. They remain limited by detection delay and VM initialization latency during sudden spikes. Proactive schemes predict future load from historical data [70, 2, 76, 77], reducing detection delay for regular patterns but remaining less effective for unprecedented or non-periodic spikes. Flare complements these approaches by serving as a safety net when unexpected load exceeds the VM capacity selected by reactive or predictive scaling.

Fast-booting infrastructure. Fast-booting infrastructure, including MicroVMs [78], Unikernels [79, 80], and snapshot-based restoration [81, 82], reduces the startup delay of new compute instances. For example, AWS Lambda uses Firecracker MicroVMs [78] to support rapid serverless elasticity. Flare is agnostic to the underlying compute substrate and only requires a secondary tier that can provision capacity quickly. While the evaluation uses serverless functions due to their widespread availability, Flare's control plane can also use other fast-booting compute types if available.

Hybrid systems with serverless. Prior work has used serverless in various contexts to improve elasticity and handle short-lived resource demand [6, 7, 83, 8, 9, 10, 11, 12, 84]. Systems such as SplitServe [6], Cackle [7], Sponge [83], and Mashup [8] use serverless functions in data analytics, stream processing, and high-performance computing workloads. These systems target workloads with explicit job, query, operator, or workflow structure, unlike microservice request paths that vary by request and service behavior.

Other works explore hybrid provisioning for machine learning inference-as-a-service [9, 10] and general cloud applications with strict SLAs [11, 12, 84]. MArk [9] and Spock [10] use serverless instances to serve ML inference requests during spikes or scale-out. LIBRA [11] and FEAT [12] route whole application requests between VMs and serverless resources at a gateway or load-balancer level. BeeHive [84] offloads parts of a monolithic web application to FaaS. These systems are distinct from Flare, which shifts traffic for specific bottlenecked services in a multi-hop microservice chain and integrates with service meshes used in microservice deployments.

VIII. Conclusion

Sudden load spikes pose a considerable challenge for VM-based microservices due to the delay of reactive scaling and the limited ability of predictive scaling to anticipate unexpected surges. Although serverless functions provide rapid elasticity, a serverless-only deployment incurs high cost for steady traffic. To address this cost-performance trade-off, we propose Flare, a hybrid architecture that keeps steady traffic on VMs and shifts traffic beyond current VM capacity to serverless functions only at overloaded services. Across three microservice applications, Flare reduces peak tail latency by 47.9% on average, eliminates all median SLO violations, and limits tail SLO violations to at most 13 seconds with an average cost increase of 4.4%. An extended version of this paper is available on arXiv [85].

Acknowledgment

We thank anonymous reviewers and EASE Lab members at the University of Edinburgh for their valuable feedback. This research was generously supported by the University of Edinburgh, and EASE Lab's industry partners and sponsors, including Huawei, Intel, Arm and Cisco.

Code and artifacts are available at https://github.com/ease-lab/Flare

References

[1] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu, "Characterizing microservice dependency and performance: Alibaba trace analysis," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 412–426. [Online]. Available: https://doi.org/10.1145/3472883.3487003

[2] S. Luo, H. Xu, K. Ye, G. Xu, L. Zhang, G. Yang, and C. Xu, "The power of prediction: Microservice auto scaling via workload learning," in *Proceedings of the 13th Symposium on Cloud Computing*, ser. SoCC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 355–369. [Online]. Available: https://doi.org/10.1145/3542929.3563477

[3] "Archive Team: The Twitter Stream Grab," Jan. 2024, [Online; accessed 9. Jan. 2024]. [Online]. Available: https://archive.org/details/twitterstream

[4] "How Netflix Ensures Highly-Reliable Online Stateful Systems," Dec. 2022, [Online; accessed 25. Apr. 2024]. [Online]. Available: https://www.infoq.com/articles/netflix-highly-reliable-stateful-systems/

[5] Z. Wang, S. Zhu, J. Li, W. Jiang, K. K. Ramakrishnan, Y. Zheng, M. Yan, X. Zhang, and A. X. Liu, "Deepscaling: microservices autoscaling for stable cpu utilization in large scale cloud systems," in *Proceedings of the 13th Symposium on Cloud Computing*, ser. SoCC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 16–30. [Online]. Available: https://doi.org/10.1145/3542929.3563469

[6] A. Jain, A. F. Baarzi, G. Kesidis, B. Urgaonkar, N. Alfares, and M. Kandemir, "Splitserve: Efficiently splitting apache spark jobs across faas and iaas," in *Proceedings of the 21st International Middleware Conference*, ser. Middleware '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 236–250. [Online]. Available: https://doi.org/10.1145/3423211.3425695

[7] M. Perron, R. Castro Fernandez, D. DeWitt, M. Cafarella, and S. Madden, "Cackle: Analytical workload cost and performance stability with elastic pools," *Proc. ACM Manag. Data*, vol. 1, no. 4, dec 2023. [Online]. Available: https://doi.org/10.1145/3626720

[8] R. B. Roy, T. Patel, V. Gadepally, and D. Tiwari, "Mashup: making serverless computing useful for hpc workflows via hybrid execution," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 46–60. [Online]. Available: https://doi.org/10.1145/3503221.3508407

[9] C. Zhang, M. Yu, W. Wang, and F. Yan, "Mark: exploiting cloud services for cost-effective, slo-aware machine learning inference serving," in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '19. USA: USENIX Association, 2019, p. 1049–1062.

[10] J. R. Gunasekaran, P. Thinakaran, M. T. Kandemir, B. Urgaonkar, G. Kesidis, and C. Das, "Spock: Exploiting serverless functions for slo and cost aware resource procurement in public cloud," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, 2019, pp. 199–208.

[11] A. Raza, Z. Zhang, N. Akhtar, V. Isahagian, and I. Matta, "Libra: An economical hybrid approach for cloud applications with strict slas," in *2021 IEEE International Conference on Cloud Engineering (IC2E)*, 2021, pp. 136–146.

[12] J. H. Novak, S. K. Kasera, and R. Stutsman, "Cloud functions for fast and robust resource auto-scaling," in *2019 11th International Conference on Communication Systems and Networks (COMSNETS)*, 2019, pp. 133–140.

[13] J. Liu, Q. Wang, S. Zhang, L. Hu, and D. Da Silva, "Sora: A latency sensitive approach for microservice soft resource adaptation," in *Proceedings of the 24th International Middleware Conference*, ser. Middleware '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 43–56. [Online]. Available: https://doi.org/10.1145/3590140.3592851

[14] R. Buyya, S. K. Garg, and R. N. Calheiros, "Sla-oriented resource provisioning for cloud computing: Challenges, architecture, and solutions," in *2011 International Conference on Cloud and Service Computing*, 2011, pp. 1–10.

[15] T. Currie, "Airbnb's 10 Takeaways from Moving to Microservices — thenewstack.io," https://thenewstack.io/airbnbs-10-takeaways-moving-microservices/, [Accessed 08-01-2024].

[16] W. Team, "Microservices at netflix: Lessons for architectural design," Jan 2023. [Online]. Available: https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/

[17] "Q&A with Jim Brikman: Splitting Up a Codebase into Microservices and Artifacts." [Online]. Available: https://www.linkedin.com/blog/engineering/archive/q-a-with-jim-brikman-splitting-up-a-codebase-into-microservices

[18] "The Opportunities Microservices Provide at Uber Engineering," Apr. 2016, [Online; accessed 8. Jan. 2024]. [Online]. Available: https://www.uber.com/en-GB/blog/building-tincup-microservice-implementation

[19] J. Q. Hylbert and S. Cosenza, "Rebuilding twitter's public api," 12 August 2020, [Online; accessed 8. Jan. 2024]. [Online]. Available: https://blog.twitter.com/engineering/en_us/topics/infrastructure/2020/rebuild_twitter_public_api_2020

[20] Z. Jia and E. Witchel, "Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices," in *Proceedings of the 26th ACM International Conference on Architectural Support for*

Programming Languages and Operating Systems, ser. ASPLOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 152–166. [Online]. Available: https://doi.org/10.1145/3445814.3446701

[21] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, 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, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 3–18. [Online]. Available: https://doi.org/10.1145/3297858.3304013

[22] "Production-Grade Container Orchestration," Jan. 2024, [Online; accessed 9. Jan. 2024]. [Online]. Available: https://kubernetes.io

[23] "Swarm mode overview," Dec. 2023, [Online; accessed 11. Jan. 2024]. [Online]. Available: https://docs.docker.com/engine/swarm

[24] "Nomad | HashiCorp Developer," Jan. 2024, [Online; accessed 11. Jan. 2024]. [Online]. Available: https://developer.hashicorp.com/nomad

[25] "Horizontal Pod Autoscaler - Kubernetes," Jan. 2024, [Online; accessed 14. Jan. 2024]. [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[26] "Cluster Autoscaler - Kubernetes," Jan. 2024, [Online; accessed 14. Jan. 2024]. [Online]. Available: https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler

[27] "AWS EKS Horizontal Pod Autoscaler Sync Interval," Jan. 2024, [Online; accessed 9. Jan. 2024]. [Online]. Available: https://github.com/aws/containers-roadmap/issues/1809

[28] "Google Kubernetes Engine Horizontal Pod Autoscaler Sync Interval," Jan. 2024, [Online; accessed 10. Jan. 2024]. [Online]. Available: https://cloud.google.com/kubernetes-engine/docs/concepts/horizontalpodautoscaler

[29] J. Hao, T. Jiang, W. Wang, and I. K. Kim, "An empirical analysis of vm startup times in public iaas clouds: An extended report," 2021.

[30] "Traffic Shedding against Stampeding Herd Effect from the Mobile App," Dec. 2022, [Online; accessed 25. Apr. 2024]. [Online]. Available: https://www.infoq.com/news/2023/10/monzo-app-traffic-shedding/

[31] "Istio," May 2024, [Online; accessed 22. May 2024]. [Online]. Available: https://istio.io

[32] "BookInfo Application," May 2024, [Online; accessed 20. May. 2024]. [Online]. Available: https://istio.io/latest/docs/examples/bookinfo/

[33] A. F. Baarzi, T. Zhu, and B. Urgaonkar, "Burscale: Using burstable instances for cost-effective autoscaling in the public cloud," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 126–138. [Online]. Available: https://doi.org/10.1145/3357223.3362706

[34] M. Carvalho, W. Cirne, F. Brasileiro, and J. Wilkes, "Long-term slos for reclaimed cloud computing resources," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1–13. [Online]. Available: https://doi.org/10.1145/2670979.2670999

[35] L. Ma, D. Van Aken, A. Hefny, G. Mezerhane, A. Pavlo, and G. J. Gordon, "Query-based workload forecasting for self-driving database management systems," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 631–645. [Online]. Available: https://doi.org/10.1145/3183713.3196908

[36] A. Mahgoub, A. M. Medoff, R. Kumar, S. Mitra, A. Klimovic, S. Chaterji, and S. Bagchi, "OPTIMUSCLOUD: Heterogeneous configuration optimization for distributed databases in the cloud," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 189–203. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/mahgoub

[37] K. Rzadca, 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 *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3342195.3387524

[38] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "Cloudscale: elastic resource scaling for multi-tenant cloud systems," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, ser. SOCC '11. New York, NY, USA: Association for Computing Machinery, 2011. [Online]. Available: https://doi.org/10.1145/2038916.2038921

[39] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in Neural Information Processing Systems*, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, Eds., vol. 27. Curran Associates, Inc., 2014. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2014/file/a14ac55a4f27472c5d894ec1c3c743d2-Paper.pdf

[40] R. Krishnamurthi, A. Kumar, S. S. Gill, and R. Buyya, *Serverless Computing: Principles and Paradigms*, 05 2023.

[41] "Lambda function scaling - AWS Lambda," Jan. 2024, [Online; accessed 11. Jan. 2024]. [Online].

Available: https://docs.aws.amazon.com/lambda/latest/dg/lambda-concurrency.html
[42] "Application Load Balancer | Elastic Load Balancing | Amazon Web Services," May 2024, [Online; accessed 22. May 2024]. [Online]. Available: https://aws.amazon.com/elasticloadbalancing/application-load-balancer
[43] "Envoy proxy - home," Feb. 2024, [Online; accessed 2. Feb. 2024]. [Online]. Available: https://www.envoyproxy.io
[44] "Envoy Proxy — Load Balancing," Jan. 2024, [Online; accessed 27. Jan. 2024]. [Online]. Available: https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/upstream/load_balancing/load_balancers
[45] "NGINX Docs — NGINX Load Balancing," Jan. 2024, [Online; accessed 27. Jan. 2024]. [Online]. Available: https://nginx.org/en/docs/http/load_balancing.html#nginx_weighted_load_balancing
[46] "Home - Knative," Jan. 2024, [Online; accessed 24. Jan. 2024]. [Online]. Available: https://knative.dev/docs
[47] "The Istio service mesh," Jan. 2024, [Online; accessed 24. Jan. 2024]. [Online]. Available: https://istio.io/latest/about/service-mesh
[48] I. Team, "Case studies from istio," 2024, [Online; accessed 30. Jan. 2024]. [Online]. Available: https://istio.io/latest/about/case-studies/
[49] R. Bhattacharya, Y. Gao, and T. Wood, "Dynamically balancing load with overload control for microservices," *ACM Trans. Auton. Adapt. Syst.*, vol. 19, no. 4, Nov. 2024. [Online]. Available: https://doi.org/10.1145/3676167
[50] "AWS App Mesh," May 2024, [Online; accessed 01. May. 2024]. [Online]. Available: https://aws.amazon.com/app-mesh/
[51] "cadvisor," Jan. 2024, [Online; accessed 24. Jan. 2024]. [Online]. Available: https://github.com/google/cadvisor
[52] Prometheus, "Prometheus - Monitoring system & time series database," Jan. 2024, [Online; accessed 25. Jan. 2024]. [Online]. Available: https://prometheus.io
[53] "Creating event-driven architectures with Lambda," https://docs.aws.amazon.com/lambda/latest/dg/concepts-event-driven-architectures.html, Jan. 2025, accessed 10. Dec. 2025.
[54] "AWS Lambda Web Adapter," https://github.com/awslabs/aws-lambda-web-adapter, Jan. 2022, accessed 10. Sep. 2025.
[55] "Serverless Adapter," https://github.com/H4ad/serverless-adapter, Jan. 2022, accessed 10. Sep. 2025.
[56] "GRPC Gateway," https://github.com/grpc-ecosystem/grpc-gateway, Jan. 2025, accessed 10. Dec. 2025.
[57] S. M. Sajal, T. Zhu, B. Urgaonkar, and S. Sen, "Traceupscaler: Upscaling traces to evaluate systems at high load," in *Proceedings of the Nineteenth European Conference on Computer Systems*, ser. EuroSys '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 942–961. [Online]. Available: https://doi.org/10.1145/3627703.3629581
[58] "Locust.io," Jan. 2024, [Online; accessed 27. Jan. 2024]. [Online]. Available: https://locust.io
[59] "Online Boutique Microservice Application," May 2024, [Online; accessed 20. May. 2024]. [Online]. Available: https://github.com/GoogleCloudPlatform/microservices-demo
[60] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, analysis, and optimization of serverless function snapshots," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 559–572. [Online]. Available: https://doi.org/10.1145/3445814.3446714
[61] "Amazon EKS Customers | Managed Kubernetes Service | Amazon Web Services," Jan. 2024, [Online; accessed 25. Jan. 2024]. [Online]. Available: https://aws.amazon.com/eks
[62] "AWS VM Instances cost," Jan. 2024, [Online; accessed 27. Jan. 2024]. [Online]. Available: https://aws.amazon.com/ec2/instance-types/t3/
[63] N. Daw, U. Bellur, and P. Kulkarni, "Xanadu: Mitigating cascading cold starts in serverless function chain deployments," in *Proceedings of the 21st International Middleware Conference*, ser. Middleware '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 356–370. [Online]. Available: https://doi.org/10.1145/3423211.3425690
[64] "Optimizing Lambda Cost with Multi-Threading," Jan. 2024, [Online; accessed 27. Jan. 2024]. [Online]. Available: https://web.archive.org/web/20220629183438/https://www.sentiatechblog.com/aws-re-invent-2020-day-3-optimizing-lambda-cost-with-multi-thread
[65] "AWS Lambda pricing calculator," Jan. 2024, [Online; accessed 27. Jan. 2024]. [Online]. Available: https://calculator.aws/#/createCalculator/Lambda
[66] J. Brutlag, "Speed Matters for Google Web Search," Google, Inc., Tech. Rep., Jun. 2009. [Online]. Available: https://services.google.com/fh/files/blogs/google_delayexp.pdf
[67] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch, "Autoscale: Dynamic, robust capacity management for multi-tier data centers," *ACM Trans. Comput. Syst.*, vol. 30, no. 4, nov 2012. [Online]. Available: https://doi.org/10.1145/2382553.2382556
[68] F. Cruz, F. Maia, M. Matos, R. Oliveira, J. a. Paulo, J. Pereira, and R. Vilaça, "Met: workload aware elasticity for nosql," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 183–196. [Online]. Available: https://doi.org/10.1145/2465351.2465370
[69] M. Straesser, J. Grohmann, J. von Kistowski, S. Eismann, A. Bauer, and S. Kounev, "Why is it not solved yet? challenges for production-ready autoscaling," in

Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering, ser. ICPE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 105–115. [Online]. Available: https://doi.org/10.1145/3489525.3511680

[70] N. Roy, A. Dubey, and A. Gokhale, "Efficient autoscaling in the cloud using predictive models for workload forecasting," in *2011 IEEE 4th International Conference on Cloud Computing*, 2011, pp. 500–507.

[71] "Autoscaling - Amazon EKS," May 2024, [Online; accessed 20. May 2024]. [Online]. Available: https://docs.aws.amazon.com/eks/latest/userguide/autoscaling.html

[72] schaffererin, "Cluster autoscaling in Azure Kubernetes Service (AKS) overview - Azure Kubernetes Service," Apr. 2024, [Online; accessed 20. May 2024]. [Online]. Available: https://learn.microsoft.com/en-us/azure/aks/cluster-autoscaler-overview

[73] "Use node auto-provisioning," May 2024, [Online; accessed 20. May 2024]. [Online]. Available: https://cloud.google.com/kubernetes-engine/docs/how-to/node-auto-provisioning

[74] A. U. Gias, G. Casale, and M. Woodside, "Atom: Model-driven autoscaling for microservices," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 2019, pp. 1994–2004.

[75] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Cost-effective scaling for microservice applications in the cloud with an online learning approach," *IEEE Transactions on Cloud Computing*, vol. 10, no. 2, pp. 1100–1116, 2022.

[76] S. Pothu and S. Kailasam, "Hybrid workload prediction for improved autoscaling in iaas clouds: An arima-olstm approach," *Ingénierie des systèmes d information*, vol. 30, pp. 961–970, 04 2025.

[77] F. Rossi, V. Cardellini, F. L. Presti, and M. Nardelli, "Dynamic multi-metric thresholds for scaling applications using reinforcement learning," *IEEE Transactions on Cloud Computing*, vol. 11, no. 2, pp. 1807–1821, 2023.

[78] A. Agache, M. Brooker, A. Florescu, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: lightweight virtualization for serverless applications," in *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*, ser. NSDI'20. USA: USENIX Association, 2020.

[79] A. Bratterud, A.-A. Walla, H. Haugerud, P. E. Engelstad, and K. Begnum, "Includeos: A minimal, resource efficient unikernel for cloud services," in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, 2015, pp. 250–257.

[80] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici, "My vm is lighter (and safer) than your container," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 218–233. [Online]. Available: https://doi.org/10.1145/3132747.3132763

[81] L. Ao, G. Porter, and G. M. Voelker, "Faasnap: Faas made fast using snapshot-based vms," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 730–746. [Online]. Available: https://doi.org/10.1145/3492321.3524270

[82] W. Weng, Y. Zhao, R. van Nieuwpoort, and A. Uta, "Serverless snapshot-resume performance in the real-world," in *2024 IEEE/ACM 17th International Conference on Utility and Cloud Computing (UCC)*, 2024, pp. 356–365.

[83] W. W. Song, T. Um, S. Elnikety, M. Jeon, and B.-G. Chun, "Sponge: Fast reactive scaling for stream processing with serverless frameworks," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. Boston, MA: USENIX Association, Jul. 2023, pp. 301–314. [Online]. Available: https://www.usenix.org/conference/atc23/presentation/song

[84] Z. Zhao, M. Wu, J. Tang, B. Zang, Z. Wang, and H. Chen, "Beehive: Sub-second elasticity for web services with semi-faas execution," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 74–87. [Online]. Available: https://doi.org/10.1145/3575693.3575752

[85] D. Dehigama, S. Jesalpura, D. Schall, A. Katsarakis, M. Kogias, R. Kumar, and B. Grot, "Flare: Leveraging serverless elasticity to absorb microservice load spikes," 2026. [Online]. Available: https://arxiv.org/abs/2605.23707