Spandana: Reconciling Strict SLOs with Low Cost under Fine-Grained Load Fluctuations

Dilina Dehigama
University of Edinburgh
Edinburgh, United Kingdom
dilina.dehigama@ed.ac.uk

Shyam Jesalpura
University of Edinburgh
Edinburgh, United Kingdom
s.jesalpura@gmail.com

Zeyu Xu
University of Edinburgh
Edinburgh, United Kingdom
zeyu.xu@ed.ac.uk

Marton Nemeth
University of Edinburgh
Edinburgh, United Kingdom
s2474972@ed.ac.uk

Shengda Zhu
The University of Edinburgh
Edinburgh, United Kingdom
shengda.zhu@ed.ac.uk

Marios Kogias
Imperial College London
London, United Kingdom
m.kogias@imperial.ac.uk

Boris Grot
University of Edinburgh
Edinburgh, United Kingdom
boris.grot@ed.ac.uk

Abstract

Cloud-based online services must contend with a load that exhibits significant fluctuations at sub-second granularity. Despite such load volatility, services must meet strict Service Level Objectives (SLOs), which forces a tradeoff between over-provisioning resources in order to meet SLO and achieving high resource utilization and cost efficiency. None of the existing approaches, which include reactive and proactive autoscalers, serverless (FaaS), and hybrid clusters combining virtual machines (VMs) and serverless workers, are able to reconcile this tension under fine-grained load fluctuation.

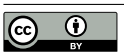
We introduce Spandana, an architecture that resolves this tension by decoupling SLO enforcement from cost optimization through a two-level control plane. A light-weight controller colocated with each application VM (the local level) steers each arriving request to the VM if the expected service time falls within the SLO target; otherwise, the request is forwarded to a stock FaaS layer (e.g., AWS Lambda). Doing so achieves high VM utilization and cost-efficiency while ensuring strict SLO compliance through the use of elastic serverless instances when needed. At the global level, Spandana's resource allocator determines the most-efficient VM provisioning considering both VM and serverless costs as well as traffic volatility. Our evaluation shows that Spandana exhibits extremely strict SLO adherence, consistently achieves CPU utilization in the range of 76-86%, and reduces costs by 5-44% over three SOTA baselines.

CCS Concepts

• **Computer systems organization → Cloud computing**.

Keywords

Autoscaling, Resource Provisioning, Cloud Computing



SoCC '26, Singapore, Singapore

ACM ISBN 979-8-4007-2735-1/26/11
https://doi.org/10.1145/3815789.3827950

ACM Reference Format:
Dilina Dehigama, Shyam Jesalpura, Zeyu Xu, Marton Nemeth, Shengda Zhu, Marios Kogias, and Boris Grot. 2026. Spandana: Reconciling Strict SLOs with Low Cost under Fine-Grained Load Fluctuations. In *ACM Symposium on Cloud Computing (SoCC '26), November 18–20, 2026, Singapore, Singapore.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3815789.3827950

1 Introduction

User-facing services running in the cloud are subjected to a highly volatile request load, with significant fluctuations at a second and sub-second granularity [15, 62]. A recent study on load balancing at YouTube reports that server load appearing relatively smooth at a minute granularity may fluctuate by over an order of magnitude at a second granularity [69]. Our analysis of a Twitter trace shows that second-scale load fluctuations exceed the average load by as much as 50%.

Such fine-grained load fluctuations make resource provisioning difficult. With cloud-based services typically deployed on VMs, system administrators are faced with conflicting requirements of keeping VM utilization low in order to meet SLO but also of minimizing the number of VMs in order to control costs. Reconciling these competing objectives is a real-world challenge. Since SLO objectives generally take priority for the sake of user experience, deployments are commonly overprovisioned relative to the average load so that fluctuations can be absorbed without violating SLO. As a result, industry reports point to CPU utilization in production clusters often well below 50%[21, 56].

To keep up with bursty loads, cloud deployments rely on autoscaling mechanisms that seek to adjust capacity in response to load fluctuations. Reactive autoscalers, such as the Kubernetes (k8s) Horizontal Pod Autoscaler (HPA) [10], are designed to detect cases when utilization is above a preconfigured threshold for a sustained period of time, in which case they react by bringing more VM capacity online. Problematically, detection and bringing new VMs online

An extended version containing additional studies is available on arXiv [25].

can take minutes [42], which means such autoscalers cannot respond effectively to short-lived bursts. Proactive autoscalers aim to predict demand and scale resources in advance[17, 41, 45, 52, 54, 57]. In practice, proactive systems operate at coarse time scales, such as minute-level intervals, to avoid instability. As a result, fine-grained load fluctuations are either missed or require overprovisioning of resources leading to high deployment cost and low utilization. What both reactive and predictive schemes lack is fine-grained elasticity that can naturally accommodate fine-grained load fluctuations.

Function-as-a-Service (FaaS), or serverless computing, offers tremendous resource elasticity, making it attractive for volatile loads. Alas, as our studies, and that of others [47, 55], show, serverless is several times more expensive than VMs for volume processing. A number of works have proposed augmenting VMs with serverless; the majority of these engage serverless instances only when faced with unexpected surges in load and/or to bridge the throughput gaps during new VMs being launched [14, 24, 34, 38, 48, 51, 64, 71]. Such schemes do not help with fine-grained load fluctuations. Libra [55] goes beyond these works by *continuously* offloading a portion of the traffic to serverless in order to balance SLO and cost. However, as our evaluation reveals, Libra's resource allocation strategy tries to provision for both SLO and cost in advance, resulting in a poor allocation under volatile traffic that compromises on cost in order to meet SLO.

Our work directly targets the tension between strict SLO, high resource utilization and low cost. We introduce Spandana, a fresh take on combining the cost-efficiency of conventional VMs with an elastic compute substrate (FaaS in this work, but other elastic resource pools are possible) to ensure strict latency SLO under highly volatile load.

The key insight exploited in Spandana's design is the decoupling of SLO enforcement from cost optimization. To that end, Spandana employs a two-level architecture. At the local level, each application instance (e.g., a VM) makes a local decision as to whether an arriving request can meet its SLO target based on the instance's current utilization level and queue length of pending requests. If the expected service time of the arriving packet exceeds the latency budget, the packet is immediately forwarded to the serverless plane. Crucially, in Spandana, a request is processed by a serverless instance *only if* it cannot be accommodated by a VM without violating SLO, thereby ensuring high VM utilization and cost efficiency with strict SLO adherence. The decision-making logic concerning whether to process a request on a VM or redirect to serverless has minimal CPU and memory footprint and is colocated with each application instance, thereby scaling naturally with deployment size and with no impact on existing intra-VM load balancing [11].

At the global level, a deployment-wide resource optimizer periodically examines the load distribution across VMs and serverless instances and adjusts VM allocation so as to minimize overall deployment cost taking into account the extent of traffic fluctuations and serverless costs. Notably, the optimizer allocates *only* for overall cost without concern for SLO, which is enforced at Spandana's local level.

We evaluate Spandana against k8s HPA, a state-of-practice autoscaler, as well as AutoBurst [35] and Libra [55], state-of-the-art schemes for cost- and SLO-aware cluster provisioning that combine VMs with burstable (AutoBurst) and serverless (Libra) instances. Our experiments show that Spandana comfortably meets strict latency SLO targets for all studied applications, and achieves desirable high CPU utilization of VM instances in the range of 76%-86% and reduces costs by 14-36% compared to HPA, 5-27% compared to AutoBurst and by 28-44% compared to Libra whenever AutoBurst and Libra meets or only slightly exceeds the SLO budget.

To summarize, we make the following contributions:

- Characterize how fine-grained load fluctuations present a unique challenge for VM-based application deployments, which must choose between cost-efficiency or SLO compliance.
- Existing resource allocation schemes force a choice between SLO adherence and cost-efficiency. Such schemes include both reactive and proactive autoscalers, serverless deployments and hybrid approaches combining VMs with serverless.
- Spandana decouples the concerns of SLO enforcement and cost efficiency into separate and dedicated control planes. SLO is enforced at each application instance by estimating whether an arriving packet can meet its latency SLO if it's processed by the instance; if not, it is redirected to FaaS, thus guaranteeing SLO adherence. Cost efficiency is achieved by a global controller that considers VM costs, serverless costs and traffic volatility.
- Spandana meets strict SLO targets while outperforming SOTA baselines on cost and CPU utilization.

2 Motivation

2.1 The Bursty Nature of Traffic

The user-triggered load on the services in the cloud is known to exhibit significant fine-grained fluctuations, a key feature of which are so-called microbursts [15, 62]. Microbursts are characterized by sudden and short-lived spikes in load with a duration of tens of milliseconds or less. In the presence of microbursts, the load on individual services deployed in the cloud can fluctuate greatly at sub-second time scales, resulting in a volatile request pattern in which the average load at coarser time scales fails to represent the highly-variable actual load.

To illustrate this behavior, we analyze a one-hour segment of a production workload trace [6] from Twitter (now, X), a popular user-facing cloud application. Figure 1 shows the request rate at two granularities – minute-scale (red) and second-scale (blue). At minute scale, computed as the rolling average over the previous 60 seconds, the load is quite stable, deviating from a trace-wide mean by at most 7.2% across all intervals. In contrast, at second scale, the trace shows deviations from the mean of as much as 50%.

Because the resolution of the data in the trace is limited to 1 second, we extrapolate the arrival rates within each 1-second interval following a Poisson process with an exponential distribution of inter-arrival times. The Poisson distribution is widely accepted as representative of human-generated traffic [27, 61]. Figure 1 shows the resulting load with significant fluctuations within each second-long interval. Fluctuating pattern of load is not unique to this workload, with similar patterns having been observed in other large-scale systems, such as YouTube's production clusters [69].

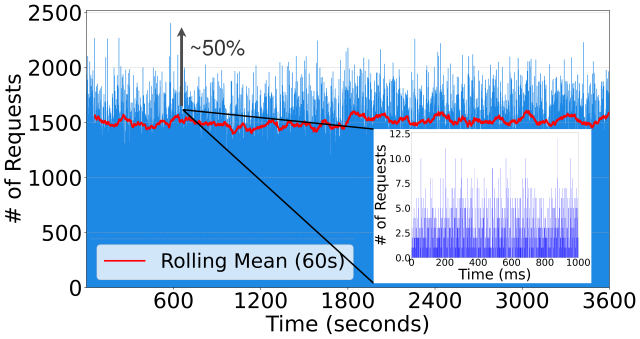


Figure 1: Production workload from Twitter showing fine-grained load fluctuations at 1-second granularity and zoomed-in figure showing fluctuations within a second

System	SLO Vio. (%)	Cost (¢)	Avg # VM	Max # VM
HPA-cost	3.5	9.2	12.1	15
HPA-slo	0.29	10.0	14.0	14
Serverless	0.02	30.9	-	-
Libra	0.84	12.7 (8.9 / 3.8)	12.4	13

Table 1: Comparison of SLO violations, cost, and number of VM instances across systems. For Libra, total cost is broken down as (VM / serverless).

2.2 Resource Provisioning for Fluctuating Load

For cloud applications, fine-grained fluctuations create a significant challenge of resource provisioning[1]. The challenge lies in balancing two competing objectives: allocating enough resources to meet strict SLOs, while simultaneously keeping resource utilization high to avoid the cost of idle capacity.

To showcase the provisioning challenge, we conducted an experiment using a 10-minute portion of the same load from Figure 1 applied to an online service, the *ratings-service* from the BookInfo application [7]. The *ratings-service* was deployed on a multi-node AWS Elastic Kubernetes Service cluster (EKS) with m5a.large VMs with 2 vCPUs and 8 GB of memory. One vCPU was allocated to each container instance of the service hosted on the VMs. From here on, we refer to each container instance of the service as an *application instance*. First, we determined the maximum throughput of a single application instance by benchmarking it with a uniform request load to find the highest request rate it could handle without violating its latency SLO target. The latency SLO target was defined as 10 times the processing time of a single request on an idle application instance.

Figure 2a shows the load trace (blue) along with the aggregate throughput of the application instances (dashed red lines). As the figure shows, the minimum cluster size needed to handle the average load (just over 1500 RPS) is 11 application instances. However, the frequent bursts clearly exceed the capacity of an 11 application-instance cluster, which leads to severe SLO violations. To handle every peak in the trace, a total of 16 application instances are needed which is 45% more than the 11 application instances required to sustain the average load. Figure 2b illustrates the direct consequence of provisioning more application instances on CPU utilization of the cluster. Provisioning for the average load with 11 application instances drives CPU utilization to a desirable high of 90%, but at the cost of an unacceptable 40% SLO violation rate. Conversely, provisioning for the peak load with 16 application instances reduces SLO violations to near-zero but at the cost of CPU utilization dropping to 60%, leaving a large fraction of CPU capacity idle. The experiment reveals the provisioning challenge for fluctuating load: clusters must either run at high utilization and violate SLOs, or they must be overprovisioned to meet SLOs at the cost of under-utilization.

In fact, the under-utilization problem is even more severe in large-scale production environments. Recent industry reports show that average cluster CPU utilization often remains well below 50%. For example, a recent report analyzing thousands of production clusters across major cloud providers found that clusters hosting production workloads reach only about 10% average CPU utilization [8]. Similarly, studies of Azure VM clusters show that majority of VMs have an average CPU utilization below 50% [21, 56].

Takeaway: With bursty workloads, clusters face a dilemma: provisioning for load peaks avoids SLO violations but requires over-provisioned, incurring cost overheads and low utilization; provisioning for the average load leads to frequent SLO violations.

3 State-of-the-Art Approaches Fall Short

Current cloud provisioning strategies struggle to reconcile the conflicting goals of strict SLO compliance, high resource utilization, and low operational cost when facing fine-grained load fluctuations. We categorize the existing solutions into three predominant classes: autoscaling with VMs, pure serverless deployments, and a hybrid approach that combines VMs and serverless [55]. Each category addresses part of the problem, but none handles fine-grained bursts without either sacrificing utilization or paying a high premium.

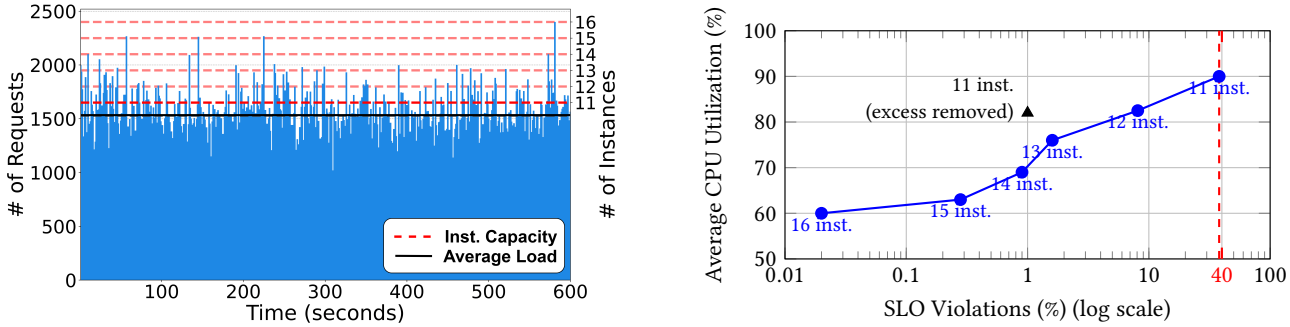
3.1 Limitations of VM Autoscaling

Standard cloud deployments rely on autoscaling to adjust the number of deployed VMs in response to demand. Reactive autoscalers, such as the k8s HPA, monitor resource usage over a window of time and add capacity when utilization exceeds a threshold [10]. A number of works have proposed more advanced autoscaling strategies [28, 49] but all are ultimately hamstrung by two limitations: (1) new VMs are slow to start, often requiring a minute or more [42]; and (2) the slow VM start times result in long decision intervals (10s of seconds) to avoid spinning up VMs for transient bursts.

Proactive autoscalers attempt to forecast demand in order to scale ahead of time, but such predictions typically operate at coarse, minute-level granularities to maintain stability [41, 58]. As a result, under highly fluctuating load at sub-second time scales, both reactive and proactive autoscalers must be tuned *either* for high SLO compliance, resulting in over-provisioning, low utilization and high cost, *or* for high utilization and low cost, leading to SLO violations due to microbursts.

We use HPA as a representative state-of-practice autoscaler and evaluate its effectiveness on the Ratings service from Bookinfo application [7] under the fluctuating load pattern from Figure 1.

[1]In today's clouds, resources are typically provisioned at the granularity of VMs. Within a VM, individual services are deployed as containers.



(a) Combined request handling capacities vs cluster sizes.

(b) CPU utilization vs SLO target violations (%)

Figure 2: The provisioning trade-off for bursty workloads.

The SLO is defined as a 99th-percentile tail latency target of 140 ms, set at 10× the median latency of the service under light load. See Section 7 for details of the methodology. Table 1 illustrates the cost and performance of these approaches. The following subsections discuss the other techniques shown in the table.

Tuning HPA for cost (HPA-cost) results in a significant rate of SLO violations of 3.5% while utilizing an average of 12.1 VMs and peaking at 15 VMs. Conversely, tuning HPA for SLO (HPA-slo) decreases the SLO violation rate to 0.29% but increases the average VM count to 14.0. The extra provisioning required to absorb fluctuations, increases the total cost to 10.03c compared to 9.18c for HPA-cost. Autoscaling approaches can easily reconcile resource utilization and SLO under load that varies only at coarser time intervals, such as minute-scale. However, when load fluctuates at sub-second granularity, which is the case in existing cloud services [69], autoscalers must choose between SLO and resource utilization, the latter directly correlated with cost. Lastly, we note that while our study focused on the state-of-practice HPA, other autoscalers face the same fundamental limitation due to the lack of fine-grained elasticity that prevents effective mitigation of sub-second load variance.

3.2 Serverless: SLO at high cost

Serverless computing, specifically Function-as-a-Service (FaaS) platforms like AWS Lambda, offers the fine-grained elasticity that VMs lack. Serverless[2] provides extremely fast (100s of ms) startup, on-demand autoscaling, and a pay-for-use billing model – features that are well-aligned with demands imposed by bursty loads.

To assess the viability of a pure serverless approach, we execute the fluctuating workload on stock AWS Lambda functions. We do not use any performance optimizations, such as pre-warming the functions or keeping them alive through periodic pings; as such, cold start latencies and other inherent performance variabilities of FaaS are reflected in our measurements.

As seen in Table 1, serverless comfortably meets the latency target, with 99.98% of requests completing within the SLO threshold (recall that the target is 99%). Such performance, however, comes at an exorbitant cost, exceeding SLO-compliant VM-based deployments by 3.1x. Thus, while serverless can address the SLO challenge, it fails the cost-efficiency requirement for high-volume services.

[2]We use FaaS and serverless interchangeably. That is, any usage of serverless refers strictly to FaaS.

3.3 Hybrid Compute to the Rescue?

As demonstrated above, serverless can provide high resource elasticity that is a good fit for fluctuating service loads and SLO targets. Meanwhile, VMs offer much better cost efficiency for volume processing. Not surprisingly, prior work has explored approaches that combine the two types of compute.

A number of papers advocate for using serverless as a fallback resource during VM scale-out and/or when the load surges [14, 18, 24, 34, 38, 48, 51, 64, 71]. The general theme is to serve the load on VMs, and engage serverless only when the load shifts and VMs cannot cope. Such schemes are effective for coarser-grained variations in load, but the high latency used to detect a load shift and the mechanisms used to redirect the traffic are not well-suited for fine-grained load fluctuations.

A more attractive approach for handling high load variability at sub-second time scales is to *continuously* use hybrid compute types to achieve both SLO compliance and cost-efficiency. The state-of-the-art in doing so is Libra [55]. In each time interval, Libra processes a pre-determined fraction of the traffic on VMs and sends the rest to serverless.

We identify three critical limitations in the Libra design that lead to suboptimal performance. First, Libra uses a fixed ratio the determines the fraction of traffic served by serverless. The user sets the parameter once with no mechanism to determine whether the chosen ratio is optimal at the current point in time. Second, when steering the requests to VMs or serverless, Libra follows a policy based on aggregated request volume that is unaware of the instantaneous load on the VMs. Thus, when a microburst arrives, Libra may overload the VMs, resulting in SLO violations. To account for this, Libra overprovisions VMs, failing to capitalize on the elasticity offered by serverless. Lastly, Libra's approach for determining the optimal number of VM instances from a cost/performance perspective only considers the total load in a previous time interval. As we show in the next section, such an approach is myopic and leads to a suboptimal resource allocation. Instead, the resource allocator must take load fluctuations into account since they affect SLO and the deployment must meet the SLO by having sufficient resources.

Table 1 highlights the inefficiency of the allocation strategy employed by Libra. The system provisions an average of 12.4 VM instances compared to 14.0 for the HPA-slo configuration. The reduction of 1.6 instances fails to offset the cost overhead of serverless,

resulting in a total cost 1.3x higher than the HPA-slo configuration. With its rigid request distribution, limited awareness of microbursts, and myopic resource allocation, Libra fails to achieve both cost-efficiency and SLO compliance at once.

4 Reconciling Strict SLO with Low Cost

The appeal of combining VMs with serverless to effectively handle volatile load lies in the former's cost-efficiency and the latter's resource elasticity. However, as shown in the previous section, existing works have not been able to reconcile the objective of strict SLO with that of low cost. Send too much bursty traffic to VMs, and SLO suffers; offload excessively to FaaS, and costs spike.

We observe that a system perfectly reconciling strict SLO with low cost under volatile load must abide by the following principles:
P1. Any request that *can be* serviced on a VM without violating SLO *should be* serviced on a VM. This ensures cost-efficiency with SLO compliance.
P2. A corollary of the above is that any request that cannot be serviced on a VM within the latency SLO (and only such requests!) must be sent to serverless. Doing so ensures strict SLO compliance while minimizing cost overheads of FaaS.

A naive implementation of the above principles would simply overprovision the pool of VMs to fully absorb the fluctuations. As shown in Section 2.2, however, such a deployment would be plagued by low average VM utilization and cost inefficiency. Thus, we introduce the third, and final, principle:
P3. The number of allocated VMs must be provisioned in a way that minimizes total cost, taking serverless costs into account.

Guided by the principles above, we unveil three insights that naturally lead to a system architecture capable of achieving strict SLO compliance and high cost-efficiency under load variability.

Insight 1. Decision as to whether a request should be processed by a VM or a serverless instance must be per-request and on-demand. Coarse-grained decisions, such as Libra's interval-based approach to commit a particular amount of load to VMs and not engage serverless until that point, inevitably lead to SLO violations and/or inflated costs under volatile load (Section 3). Instead, principles P1 and P2 dictate that each request should have an opportunity to execute on a VM (for cost-efficiency), and only execute on serverless when SLO compliance is at risk.

Insight 2. Just because a VM is operating at its peak capacity at a certain point in time does not mean a request should be offloaded to serverless. Latency SLO is typically defined as a multiple of the average-case request service time in the absence of queueing and resource contention. By design, this implies that a certain degree of queueing is acceptable from the latency SLO standpoint. Thus, the mere fact that a VM is fully occupied at a given instant of time does not mean that an arriving request cannot be completed within its SLO budget. As dictated by queueing theory, it is the combination of the service rate (of the VM) and the number of pending (queued) requests that together determine the expected service time of the request if it were to wait. Only if the expected time to completion, including the queueing time, exceeds the SLO target, should the request be offloaded to serverless.

Insight 3. Knowledge of average or total load is insufficient to find a cost-optimal VM provisioning. For instance, Libra examines the number of requests received in a time interval, uses that to decide on a fraction to be processed by VMs, and from that, determines the number of VM instances needed. Problematically, under volatile load, such an approach results in a non-optimal number of allocated VMs ultimately leading to inflated cost. Instead, the extent of load variability must be considered for optimal VM provisioning.

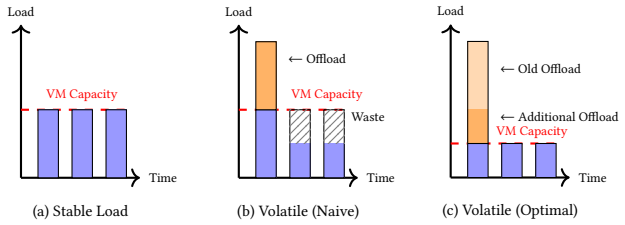


Figure 3: Cost minimization strategies. (a) Stable load allows perfect VM matching. (b) Provisioning for *average* volume under volatile load creates under-utilization (hatched) during lows. (c) Optimal allocation provisions VMs for the *base* load to eliminate waste, using serverless (orange) for the peaks.

To illustrate why total load volume is an insufficient metric for cost optimization, consider the scenarios in Figure 3. Scenario (a) is characterized by a load that does not vary in time. Under such load, a VM provisioned at the average request rate achieves 100% utilization and has no need for serverless. In scenarios (b) and (c), the load varies in time but the total number of requests is identical to scenario (a). Scenario (b) shows that applying that same "average-based" provisioning as used in Scenario (a) is sub-optimal. During the initial burst, the load exceeds VM capacity, forcing the rest to serverless. During the subsequent lows, the provisioned VMs sit idle, which is wasteful cost-wise. Scenario (c) shows a cost-optimal strategy, which is to provision fewer VMs to maximize their utilization and rely on serverless strictly for the bursts.

While the example above is simplistic (in practice, sending too much traffic to FaaS merely for the sake of maximizing VM utilization can be cost-inefficient), it helps intuit Insight 3: the optimizer must account for the distribution of the load, not just its volume.

5 Spandana

5.1 Overview

We introduce Spandana, a system that leverages the principles and insights above to reconcile strict SLO latency objectives with extreme cost efficiency. Spandana uses conventional VMs to cost-efficiently serve the bulk of the load while dynamically steering individual requests that are likely to violate SLO to serverless instances. A key challenge addressed by Spandana is how to split the incoming requests between VMs and elastic compute in real time so as to minimize cost by avoiding VM overprovisioning while meeting SLO targets.

To address this challenge, Spandana operates at two levels of granularity. At the instance level (pod or VM), Spandana decides on a per-request basis whether the instance can serve the request within the latency budget or if the level of queueing is such that an SLO violation is likely. In the latter case, the request is redirected to a stock serverless plane (e.g., AWS Lambda). At coarse grain,

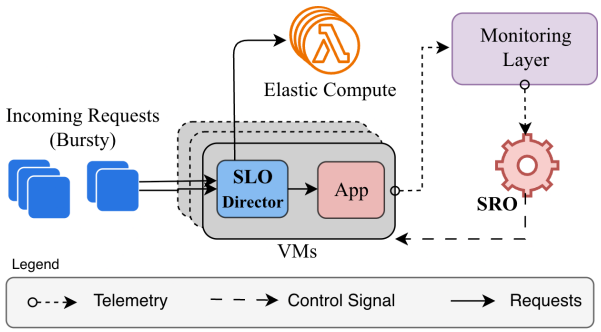


Figure 4: High-level architecture of Spandana

Spandana monitors the fluctuating load across the VM instance pool and periodically adjusts the VM allocation so as to maximize overall cost-efficiency taking into account both VM and serverless costs. The two-level architecture allows Spandana to be both SLO-compliant despite fluctuating load and economically efficient over longer time scales.

By design, Spandana separates the concerns of SLO enforcement from cost optimization. The former happens at the finest grain (instance-level, on-demand, per-request); the latter is performed at coarse grain (periodically, across the entire instance pool). This explicit separation of concerns is at the heart of Spandana's effectiveness, and differentiates it from prior works, such as Libra [55], which conflate the two.

Figure 4 illustrates the architecture of Spandana. At the instance level, a SLO Director is colocated with each application instance running on a VM. The SLO Director intercepts all incoming requests and makes a real-time decision to either serve the request locally (i.e., on the VM) or offload it to elastic compute. At the cluster level, the centralized Spandana Resource Optimizer (SRO) collects telemetry data from all instances via a monitoring layer. By analyzing the load across all instances and how effectively it is served across the two compute types, the SRO adjusts the number of provisioned VMs to optimize for cost. We next describe each component in detail.

5.2 SLO Director

The SLO Director acts as a distributed, per-instance decision point that determines whether each incoming request should be served by the colocated VM-based application container instance or offloaded to the elastic compute. The SLO Director shields the application instance from bursty demand by maintaining a bounded queue in front of the instance; when the queue is full, newly arriving requests are immediately offloaded to the elastic compute. All incoming requests are intercepted and passed through the SLO Director, enabling timely routing decisions without requiring any changes to clients or the application.

SLO Director's queue acts like a leaky bucket, shaping the traffic to the local instance to avoid overloading it (details in Section 6.2) with overflow redirected to serverless. Its bounded length facilitates service time estimation, which is needed to determine if an incoming packet can meet its SLO target; given the queue length of L_{Queue} and an average service rate of μ, the expected queueing time is L_{Queue}/μ.

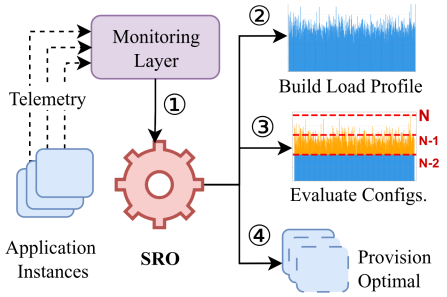


Figure 5: Spandana Resource Optimizer (SRO) in action

SLO Director's organization is informed by principles P1 and P2 (Section 4, which dictate that the default service point for all requests is a VM due to its cost-efficiency, unless SLO is in jeopardy, in which case the at-risk requests (and only such requests) should be served by serverless instances. SLO Director's specific design is derived from Insights 1 and 2 (Section 4), which argue for per-request decision making and the need for queueing in front of the instance to absorb small bursts without either violating SLO or using costly serverless compute.

5.3 Spandana Resource Optimizer (SRO)

While the SLO Director manages traffic in real time on individual instances, the *Spandana Resource Optimizer (SRO)* runs as a control loop for longer-term cluster-wide optimization aimed at maintaining high utilization and cost efficiency. Existing autoscalers typically determine cluster size based on aggregated load metrics over a recent window, scaling resources only when utilization crosses a predefined threshold. Such an approach fails to account for the fine-grained volatility of the workload. The SRO differentiates itself from these traditional mechanisms by seeking a global lowest cost configuration rather than simply reacting to load volume. As detailed in Insight 3 (Section 4), the SRO explicitly accounts for traffic fluctuation profile to find the optimal balance point.

To identify the optimal provisioning level, the SRO performs a "what-if" analysis by simulating various provisioning configurations against the recent high-resolution load profile. For every potential cluster size, the SRO calculates the total projected cost, which combines the fixed cost of the provisioned VMs with the estimated cost of offloading the residual bursty traffic to FaaS.

Its core logic, formalized in Algorithm 1, is a continuous control loop triggered at a configurable coarse-grained (minutes-scale) interval. The algorithm proceeds as follows. First, the optimizer fetches the recent load history to build a load profile (line 2). Next, it proceeds to the main for-loop (lines 7-20), which implements the core "what-if" analysis. There, the optimizer simulates a range of provisioning levels by iterating through all possible instance counts, from one instance up to the maximum required to handle the historical peak load. For each simulated instance count (inst), it first calculates the fixed cost of the VMs (line 8). Next, to determine the offload portion, it calculates the residual load by iterating through the *load_profile* and subtracting the VM cluster's capacity at each time step (lines 11-13). The cost of handling this residual load on serverless is then calculated (line 14). After summing the VM and serverless costs to get a total projected cost, the algorithm identifies the configuration with the minimum cost (lines 16-19). Finally, if the

optimal instance count *(inst_opt)* differs from the current number of running instances, the optimizer scales the number of replicas of the target application's deployment (lines 21-23).

Figure 5 provides an overview of the SRO's control loop. ① The SRO begins each interval by fetching recent request-rate histories of application instances via the monitoring layer. ② Then it aggregates request-rate histories from the instances to construct a load profile (request rate vs. time), giving SRO a global view of the demand seen by the application. ③ Using this profile, the SRO evaluates a range of configurations, each defined by a different number of VM instances. Each configuration corresponds to some split between the stable *Base portion* handled by the VMs and the *Offload portion* offloaded to the elastic compute. For each choice, the SRO calculates the combined cost of running the VMs and offloading the offload portion. ④ The SRO then identifies and provisions the lowest-cost configuration[3]. This process repeats at each interval, ensuring that longer-term workload variations are accommodated by the VM provisioning, while short-term bursts are handled via the SLO Directors.

5.4 Discussion

Centralized vs. Distributed Implementation. The architecture of Spandana supports both centralized and distributed implementations. A centralized approach, which we evaluate as Spandana-C in Section 8, employs a global load balancer that routes traffic based on the aggregate state of all instance queues. Such a design benefits from global visibility, which allows the balancer to direct requests to the least-loaded instances [69]. However, the centralized approach requires a sophisticated load balancer tracking the state of the VMs across the deployment. The load balancer must also be present at each hop in a multi-service chain.

Conversely, the distributed design places an SLO Director alongside each application instance to make local offloading decisions. We select the distributed architecture as the default implementation for Spandana. The decentralized approach aligns naturally with modern cloud-native application deployments, allowing Spandana to integrate seamlessly into existing deployments without requiring changes to the cluster-wide ingress or load balancing infrastructure.

Feasibility of Serverless Offloading. Spandana maintains SLO compliance for offloaded traffic by leveraging the fact that modern FaaS latencies comfortably fit within the application's SLO target. Formally, the feasibility of offloading is governed by the inequality:

$$T_{redirect} + T_{cold} + T_{exec} < \text{SLO}$$

In practice, cold starts are rare (as shown in Section 8.1) because frequent offloading due to load volatility naturally keeps function instances warm. Furthermore, SLO slack (commonly 5-10x of the typical service time) provides a sufficient buffer to absorb both the minimal redirection overhead and the standard execution time in the serverless tier, ensuring robust compliance.

Generalizability of Compute Resources. While this work evaluates Spandana using a homogeneous pool of standard VMs and serverless functions as the elastic tier, the underlying architecture is agnostic to specific compute instance types. Spandana can readily adapt to alternative configurations, such as employing burstable instances as the elastic resource or orchestrating a heterogeneous cluster of mixed VM types to optimize steady-state costs. While incorporating these resource-level optimizations could further enhance cost-efficiency, they represent orthogonal improvements that do not alter the fundamental design principles or the key insights presented in this work.

[3]Note that SRO only provisions the VMs. The offload portion is dynamically redirected to elastic compute by SLO Directors. However, the cost of elastic compute is accounted for by the SRO.

Algorithm 1 Resource Optimizer Logic

Require:
n_{curr}: current number of application instances
C_{vm}: cost parameters for VMs
$C_{\text{serverless}}$: cost parameters for serverless backend
$RPS_{\max}$: max requests per second per instance

```
1:  function OptimizeResources
2:      lp ← FetchLoadProfile()
3:      rps_peak ← GetPeakRPS(lp)
4:      inst_max ← ⌈rps_peak/RPS_max⌉
5:      cost_min ← ∞
6:      inst_opt ← n_curr

7:      for inst ← 1 to inst_max do
8:          vm_cost ← VMCost(inst, C_vm)
9:          residual ← []
10:         vm_capacity ← inst × RPS_max
11:         for each time step t in lp do
12:             residual[t] ← max(0, lp[t] − vm_capacity)
13:         end for
14:         sl_cost ← ServerlessCost(residual, C_serverless)
15:         total ← vm_cost + sl_cost
16:         if total < cost_min then
17:             cost_min ← total
18:             inst_opt ← inst
19:         end if
20:     end for

21:     if inst_opt ≠ n_curr then
22:         ScaleDeployment(inst_opt)
23:     end if
24: end function
```

6 Spandana Under the Hood

6.1 Technology Stack

Container Orchestration. Spandana is implemented on top of k8s, which manages the deployment and lifecycle of the provisioned service instances. In k8s, the fundamental unit of deployment is a pod, which is a logical group of one or more containers that share storage and network resources. In our setup, each pod contains the main application container and the sidecar container similar to how most online services are deployed today [19, 75]. This co-location allows the two containers to be treated as a single, scalable unit.

Elastic Compute. Our current implementation uses AWS Lambda Functions as the elastic compute. Lambdas have the desirable features of fast start time (well under 1s [44, 65]), high scalability and pay-per-use billing, all of which make it well-suited for handling bursty components of load. Importantly, Spandana's design is not tied to Lambda; any future cloud service that offers fast scaling and pay-per-use billing could be used instead.

Monitoring. Our monitoring layer uses a standard cloud-native stack to provide metrics for the Resource Optimizer. We use Fluent Bit [2] as the log collection agent that runs on every cluster node. These agents gather logs from all sidecars and forward them to Loki [32], a centralized aggregation system. This pipeline enables the Resource Optimizer to query Loki for RPS metrics and construct the application's time-series load profile.

6.2 Implementing the SLO Director

To realize the SLO Director, we built a custom sidecar proxy that is deployed with each application pod. The proxy is written in Go and mimics the core functionality of production-grade proxies such as Envoy [1], namely traffic interception and forwarding. A custom proxy allowed us to embed SLO Director's logic without the challenge of modifying a more complex codebase. This is not a limitation, however, and another implementation could instead deploy SLO Director logic as part of an existing sidecar.

Queueing and Offloading Logic. The SLO Director, residing in a sidecar, maintains a bounded FIFO request queue *(RQ)* to buffer incoming load. The size of the RQ is configurable and sets the maximum local backlog tolerated before offloading. Any request that arrives when the RQ is at this maximum capacity is immediately offloaded by invoking an AWS Lambda function with the request payload. For offloaded requests, SLO Director subsequently receives the response and forwards it back to the client over the original connection. The offloading process is completely opaque to the client requiring no client-side changes.

Request Pacing and SLO Adherence. To create a smooth load on the local application instance, the SLO Director implements a leaky bucket-like pacing mechanism. It spawns a dedicated thread that dequeues and forwards requests to the local instance. The time between forwarded requests is governed by a *pacing interval*, calculated as $T_{\text{wait}} = 1/RPS_{\text{max}}$, where ($RPS_{\text{max}}$) is the application instance's maximum sustainable request handling capacity per-second, determined via profiling.

The controller continuously monitors the instance's response latency. If the response latency exceeds the SLO target of the deployed service, the controller dynamically increases the pacing interval (thus lowering the rate) to alleviate pressure on the application. In practice, we have found that the ability of the controller to adjust the pacing rate for their individual instance (rather than using a fixed value for all instances) is helpful because actual performance of different instances of the same type varies across the deployment. By adjusting its pacing rate, each instance can maximize throughput in an SLO-compliant manner.

Metric Exposure. To support cluster-level optimization by the SRO, the SLO Director records request-level metadata within each pod. Specifically, it logs arrival timestamps, response latencies, and whether each request was served locally or offloaded. These logs are written in a structured format such that they can be consumed by the monitoring layer without interfering with the application.

6.3 Implementing the SRO

Spandana Resource Optimizer (SRO) is implemented as a standalone **Go application** deployed as a standard k8s Deployment. This design ensures the SRO runs continuously alongside the application workload, leveraging k8s' native resilience features.

Data Ingestion and Profiling: To build the load profile required for optimization, the SRO interfaces with **Loki**, our centralized log aggregation system. At the start of every control interval (configured to 2 minutes), the SRO queries Loki to fetch aggregated RPS metrics from all active SLO Directors. We utilize Loki's LogQL query language to efficiently retrieve and build the load profile.

Control Loop Execution: The optimization loop runs in a dedicated goroutine triggered by a ticker. To ensure stability, we employ a configurable cooldown period between scaling actions. In our measurements, the optimizer consumes less than 1% of a single vCPU with a negligible memory footprint, making it lightweight.

6.4 Spandana in a Microservice Chain

Spandana's design extends naturally to applications composed of microservice chains. When an application container running on a VM handles a request and subsequently calls a downstream microservice, Spandana allows the request to follow the standard intra-cluster communication path. The request is routed using k8s' internal service discovery mechanisms [9] to an instance of the appropriate downstream microservice in the cluster. The SLO Director at the destination instance then intercepts the call and applies its queueing and offloading logic.

Once the serverless instance completes processing, the request is routed back to the VM cluster by configuring the elastic compute to direct downstream requests to the k8s service endpoint of the target microservice. Upon arrival, the request is intercepted by the SLO Director at the downstream microservice, which manages the request like any other incoming request. This approach ensures that each request at *every* hop in the microservice chain has an opportunity to run on a VM, hence ensuring high VM utilization and low cost, while retaining the ability to execute on elastic compute in order to meet SLO if the VM instance is at capacity.

6.5 Portability and Developer Effort

Spandana does not assume that all VM-hosted services can run unchanged on a FaaS platform. Porting to serverless is most straightforward for stateless services (or services whose state is already externalized to managed storage). Services with their own persistent state are treated as VM-only in Spandana. For stateless services, our experience shows that the porting effort is minimal. Serverless platforms such as AWS Lambda provide native support for container images [3], allowing the same container to run on both k8s (VM-backed) clusters and Lambda without repackaging. For applications initially designed for VMs, tools such as the AWS Lambda Web Adapter [4] and Serverless Adapter [5] translate Lambda's event-based model into standard HTTP requests, enabling applications to execute in Lambda without modifications. Any service that is difficult to port need not be ported; Spandana can be viewed as a best-effort optimization from the developer's standpoint. In a microservice graph, some services would then be "Spandana-fied" to reduce cost under SLO constraints, others would run as VM-only.

7 Experimental Methodology

Studied Applications. We evaluate Spandana on four cloud applications that span popular language runtimes and workload

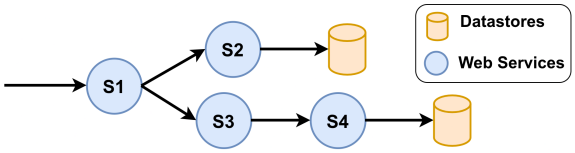


Figure 6: Service topology of BookInfo application

characteristics. The Ratings and Details web services are drawn from the BookInfo application [7]; they are I/O-bound services implemented in Node.js and Ruby, respectively. The other two, Image Processing and Compression, are CPU-bound data-processing Python applications adapted from SeBS [20]. All applications are configured to use managed cloud databases, reflecting common cloud deployment practices. Single-request latencies on an idle instance are 8ms (Details), 14ms (Ratings), 40ms (Image Processing), and 130ms (Compression).

We also evaluate an application with a chain of microservices. For this, we use the BookInfo application, which consists of four web services — Productpage (S1), Details (S2), Reviews (S3), and Ratings (S4) — arranged in a mix of sequential and fan-out dependencies, as shown in Figure 6. The leaf services in this chain are configured to use DynamoDB as their datastore backend. The SLO is defined with respect to the entire chain and is set to 10x the latency of a single request traversing an otherwise idle chain.

Load Trace & Replay Setup. For our main experiments, we use a one-hour Twitter trace [6] to drive a realistic load on a cloud service. Other public traces report load at even coarser granularity [61] hence obscuring fine-grained behavior, or are much older [67]. The chosen trace exhibits a steady load at coarse granularity with fine-grained fluctuations as discussed in Section 2.1, making it representative of an online service in steady state. We scaled up the trace [60] to generate sufficient load for the studied applications to run on a multi-node cluster with at least 10 instances per application. We replay the load trace on each application three times and report the run with the median SLO violation rate. To evaluate Spandana's ability to handle unexpected load spikes, we selected a different one-hour Twitter trace that includes a load surge peaking at 2× the mean load. We use k6 [33], a modern load generator, to replay the trace. K6 runs on a dedicated VM (4 vCPUs, 16 GB of RAM) in the same AWS region as our k8s cluster.

Studied Systems.

1) Kubernetes HPA - As a baseline representing common industry practice, we use HPA, the default autoscaling mechanism in k8s. The HPA is a simple yet widely-used autoscaling mechanism, with recent industry reports highlighting its popularity among organizations running k8s [23]. We study two configurations of HPA:
HPA-S (Standard): Configured to scale based on CPU utilization with a target threshold of 60%, a common production setting to balance performance and cost[31, 41].
HPA-O (Oracle): Configured to just meet the latency SLO (using a-priori knowledge of the entire trace), thus resulting in the most cost-efficient SLO-compliant HPA setup possible.

2) Libra - We compare against Libra [55], a state-of-the-art hybrid framework that is the closest related work to Spandana. Libra attempts to balance cost and SLO by proactively engaging FaaS alongside VMs. To ensure a fair comparison under high-throughput conditions, we implemented a high-performance version of Libra in Go. Consistent with our own setup, we use m5a.large EC2 VM instances for the IaaS tier and AWS Lambda for the FaaS component in our Libra implementation. We set its key parameter p to 80%, a value suggested by the original authors and empirically verified in our setup as the threshold that just meets the SLO target.

3) AutoBurst - We compare against AutoBurst [35], a SOTA research system that combines hybrid compute types in the form of low-cost burstable[4] instances and regular on-demand VMs. Stated goals for AutoBurst are to minimize cost while meeting SLO.

AutoBurst requires configuring several parameters, notably the initial numbers of on-demand and burstable instances at experiment start, and the *desired credit level* for burstables. Guided by communication with the work's authors, we size the on-demand pool to sustain the average load and configure the burstable pool to absorb demand beyond the provisioned on-demand capacity. We set the *desired credit level* based on the load fluctuations of our trace and include a warm-up period for burstable instances to accumulate credits to this level before experiments begin. For PD controller parameters, we adopt the defaults suggested by AutoBurst. Consistent with the original AutoBurst paper, we use *m5a.large* instances as the on-demand compute (same as what Spandana uses for its VMs) and *t3a.small* instances as the low-cost, burstable compute.

4) Spandana - Default is a decentralized design where the SLO Director is co-located with each VM to make local, per-request offloading decisions. We empirically tuned two parameters of Spandana: the SLO Director queue length and the SRO control loop interval. For SLO Director, the queue length must absorb short bursts on the VM without adding queueing delay that violates the latency SLO. For leaf services, we set the queue length to nine requests, allowing up to nine waiting requests while keeping the requests within the 10× latency SLO when served locally on the VM. For non-leaf services, we double the queue length to 18, because time spent waiting on calls to downstream services allows a larger backlog before CPU saturation. More generally, the queue length should be set per service based on its concurrency tolerance, such as I/O overlap or multi-threading and latency SLO target. For SRO, the control loop runs every two minutes, an interval chosen to account for the startup latency of new VMs and avoid scaling oscillations. This choice aligns with common autoscaling practice, where minute-scale intervals balance responsiveness and stability [41, 58].

We also evaluate a centralized variant of Spandana, denoted as **Spandana-C**. This variant uses a centralized load balancer at the cluster's entry point that maintains a global view of all individual instance queues. While such an omniscient load balancer is impractical in large-scale production clusters due to the overhead and complexity of maintaining real-time global state of all VMs, it serves as a useful "near-optimal" baseline for load balancing across all application instances. By forwarding traffic to the least-loaded instance using its perfect knowledge, Spandana-C allows us to isolate the cost and performance benefits of having global visibility for request placement, thus reducing the need for serverless through the ability to choose the least-loaded VM. Similar to the primary

[4]Burstable VMs are intended for low average CPU utilization (e.g., 20%) [12] but can burst to higher utilization on-demand. When utilization is below a threshold, burstable instances accumulate credits that can then be spent in high-utilization periods.

App	Spandana	Spandana-C	HPA-S	HPA-O	AutoBurst	Libra
Ratings	0.05	0.01	0.02	0.80	0.32	0.60
Details	0.34	0.14	0.14	0.54	**1.14**	**1.59**
Compr	0.01	0.01	0.02	0.57	0.15	0.07
Img Proc	0.01	0.00	0.02	0.74	0.23	0.20

Table 2: SLO violations (%). Red indicates violation of the 1% SLO target.

decentralized architecture, Spandana-C still leverages serverless offloading: if the load balancer detects that the queues on all active VMs are saturated, it routes incoming requests directly to FaaS.

Cluster. Our experiments were run on an AWS EKS cluster using m5a.large EC2 VM instances. For the elastic compute, we use AWS Lambda. Note that we do not rely on pre-warmed functions or attempt to keep functions warm using keep-alive (dummy) requests or similar mechanisms. Thus, all presented results intrinsically capture various Lambda-related artifacts including cold starts.

Evaluation Metrics. We compare Spandana against the baselines using CPU utilization, SLO adherence, and cost. Each application instance is provisioned with 1 vCPU, and we report the CPU utilization of the main application container to ensure a fair comparison across systems. For SLO, we define the latency target as 10x the latency of a single request on an idle application instance and adopt a strict SLO violation limit of 1%, meaning that at most 1% of requests may exceed the target latency. Finally, we report monetary cost based on AWS pricing as of September 2025 covering both VM instances (AWS EC2) and serverless invocations (AWS Lambda).

8 Evaluation

8.1 Main Results: SLO/Cost/Utilization

We first evaluate Spandana's ability to improve CPU utilization and reduce cost while adhering to strict SLO. We start with the latter, a firm requirement that must be met for good user experience.

SLO: Table 2 shows the SLO violation rates for all systems across the four applications. Both Spandana variants and both HPA variants meet the strict 1% SLO violation budget on all applications. Spandana-C achieves the lowest SLO violation rate of all studied systems as it uses its omniscient load balancer to place requests and quickly redirects to serverless if all instances experience queueing. As expected, HPA-O has a higher violation rate than HPA-S since the former is optimized for cost under SLO compliance using oracle of the trace. HPA-S must be overprovisioned to contend with unkown load, resulting in fewer violations but higher cost as shown below. AutoBurst & Libra exceeds the 1% SLO budget for *Details service*. The Details service presents the greatest SLO challenge among the evaluated applications due to its low typical service time, resulting in the lowest SLO target of just 70ms and making it highly sensitive to even minor delays. For other applications, both AutoBurst and Libra meet the SLO target but with higher violation rates than either HPA or Spandana. The reason for AutoBurst and Libra having higher violation rates is their inability to handle fine-grained bursts. Lacking a mechanism to offload traffic at per-request granularity, they continue routing fine-grained bursts to already-overloaded VMs, directly causing the queueing delays that result in SLO violations.

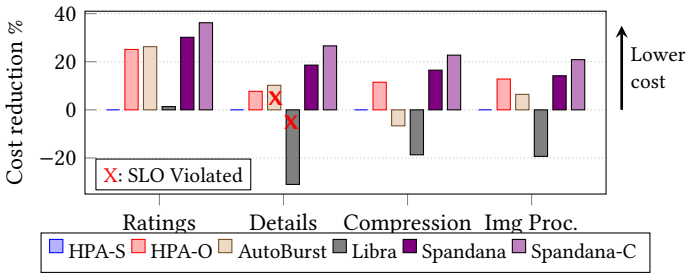


Figure 7: Cost reduction compared to HPA-S (higher is better).

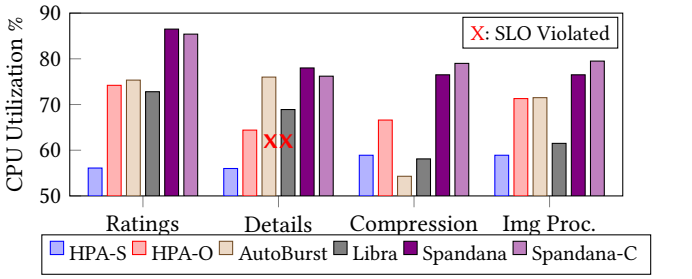


Figure 8: CPU utilization (higher is better).

Cost-efficiency: Figure 7 presents the cost reduction for the studied systems relative to HPA-S. Note that higher is better in the figure, while negative values indicate a cost increase relative to HPA-S. AutoBurst reduces costs for three applications by 6-26% (though the savings on the Details service come at the cost of an SLO violation) and incurs a cost increase of ≈7% for the Compression service. Libra performs poorly, increasing costs by 19-31% on three out of four applications. We find that the inefficiency of Libra stems from sub-optimal load balancing decisions that offload a significant volume of requests to expensive FaaS resources (request & cost distribution shown in Figure 9). In contrast, both variants of Spandana consistently reduce costs across all workloads while strictly adhering to SLOs. Spandana achieves cost reductions of 14-30% (averaging 19.9%), consistently outperforming even Oracle HPA (HPA-O). Spandana-C further optimizes efficiency, achieving reductions of 21-36% (averaging 26.6%) by leveraging the central load balancer's omniscient knowledge of VMs' load to make better request routing decisions.

CPU utilization: Figure 8 plots the average CPU utilization achieved by the schemes. HPA-S consistently runs at low utilization (approximately 56-59%) due to its conservative scaling threshold. AutoBurst and Libra achieve average utilization of 54-76% and 58-73%, respectively. While mostly better than HPA-S, both are significantly inferior to Spandana. Both Spandana variants constantly achieve the highest utilization ranging from 76-86% across all applications with strict SLO compliance (unlike AutoBurst and Libra). Spandana pushes utilization to 76.5-86.5%, significantly outperforming HPA-S and surpassing even HPA-O on all workloads. Spandana-C performs comparably. Both Spandana variants safely operate resources near saturation without compromising SLO.

FaaS: Figure 9 shows the fraction of requests served by FaaS and the corresponding cost proportion for Libra and variants of

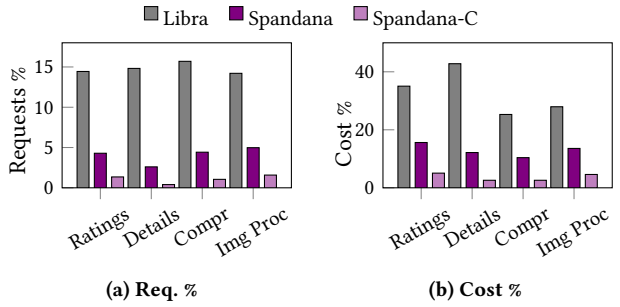


Figure 9: Breakdown of FaaS request volume (a) and cost contribution (b) across applications.

	Spandana	HPA-O	HPA-S
SLO Violations. (%) ↓	0.05	0.45	**0.00**
Total Cost ($) ↓	**1.89**	1.94	2.29
Avg. CPU Utilization (%) ↑	**77.0**	70.6	59.3

Table 3: End-to-end SLO violations (%), Total Cost ($), and Average CPU Utilization (%) for BookInfo service chain.

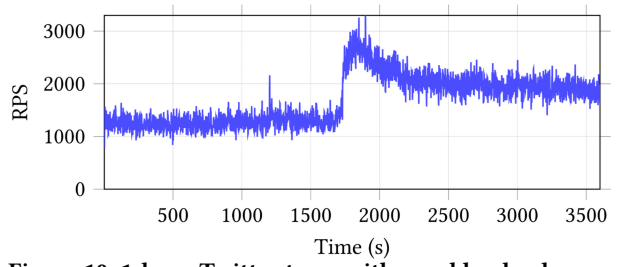


Figure 10: 1-hour Twitter trace with a sudden load surge.

Spandana. Across all applications, Spandana offloads only a small fraction of requests, ranging from 2.6% to 5%, which limits the FaaS portion of the total cost to between 10% and 15%. Spandana-C lowers both numbers, offloading merely 0.4-1.6% of requests and keeping FaaS costs below 6% for all workloads, thanks to its omniscient request routing. In contrast, Libra consistently offloads more requests to FaaS (roughly 15% of the total), resulting in FaaS charges consuming a large share of the overall cost at 25-43%.

We also assess the impact of FaaS cold starts on Spandana. Recall that Spandana offloads only a tiny fraction of total traffic (2.6–5%) to FaaS. Within this already small volume, we find that cold starts affect a negligible percentage of requests: 0.03% for *Ratings*, 0.13% for *Details*, 0.10% for *Compression*, and just 0.01% for *Image Processing*. The low cold start rate partly explains why SLO is not compromised despite cold starts in the critical path of the request latency. The other reason why cold starts do not affect SLO is in the case of compute-intensive workloads (*Compression* and *Image Processing*), whose longer processing times result in a higher SLO target that is large enough to absorb the overhead of a cold start.

8.2 Service Chains

We next evaluate Spandana on service chains using the BookInfo application, shown in Figure 6. Spandana works naturally for chains because the distributed SLO Director logic applies independently

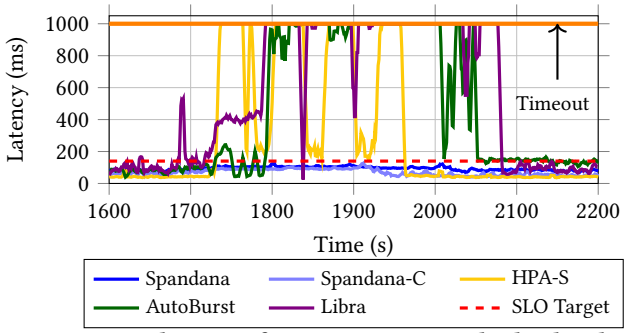


Figure 11: P99 latency of *Ratings* service under load spike

	Spandana	Spandana-C	HPA-S	AutoBurst	Libra
SLO vio.(%) ↓	0.08	**0.02**	**5.96**	**7.87**	**9.43**
Cost (¢) ↓	74.4	**57.5**	81.8	90.8	95.4
CPU Util. (%) ↑	**92.0**	87.0	58.0	71.1	73.5

Table 4: CPU utilization (%), SLO violations (% of requests above SLO target), and cost (¢) under a load spike.

at each hop and uses existing cloud infrastructure. In contrast, Spandana-C, AutoBurst and Libra do not "natively" support chains, requiring significant modifications and additional costs (in the form of centralized routing and decision-making components *at each hop*). Thus, we compare only to HPA as the only system that, like Spandana, natively works for chains.

For this study, SLO violations are calculated from end-to-end chain latency, while cost and utilization are aggregated across the deployment. Results are summarized in Table 3. Spandana comfortably meets the SLO target with a violation rate of just 0.05% while delivering the lowest total cost, undercutting HPA-S by 17% and the Oracle HPA by 2.6%. This cost efficiency is driven by better resource usage: Spandana maintains an average CPU utilization of 77.0% across all services in the chain, significantly higher than both HPA-S (59.3%) and HPA-O (70.6%).

These results demonstrate that Spandana's benefits extend beyond individual applications to entire chains of microservices, where Spandana consistently increases utilization and lowers cost while remaining within the SLO budget.

8.3 Handling Load Spikes

To evaluate Spandana under sudden demand surges, we use a Twitter trace that includes a sharp load spike, shown in Figure 10. The spike arrives at ≈1700s and eventually settles at a higher load level than the pre-spike load level. We use the Ratings service for this experiment and compare Spandana variations against HPA-S, AutoBurst and Libra.

Figure 11 shows the tail (P99) latency during the surge (1600–2200s). During the surge, AutoBurst's tail latencies remain at the 1s timeout for several minutes, while HPA-S and Libra oscillate between recovery and repeated timeouts. Such sustained periods of high tail latency degrade user experience. In contrast, both variants of Spandana keep P99 latency stable and within the SLO throughout the surge, showing no visible impact from the load spike.

Offload level	CPU (%)	Mem. (MB)
No-offload / Forward-only	8.1	7.3
10% offload	9.6	8.4
50% offload	13.6	9.8

Table 5: SLO Director sidecar resource use *per application instance* (averages across instances over the run).

We summarize average CPU utilization, SLO violations, and cost in Table 4. Both Spandana and Spandana-C achieve desirable high CPU utilization of 92% and 87% respectively, significantly outperforming HPA-S (58%), AutoBurst (71%), and Libra (74%). Furthermore, the Spandana setups nearly eliminate SLO violations (limiting them to just 0.02%-0.08%) while simultaneously maintaining the lowest total cost among all compared systems.

8.4 Overhead Analysis

SLO Director Overhead. To evaluate the overhead introduced by the offloading logic in the SLO Director, we measure the resource usage of the sidecar container colocated with each application container by driving a constant load that keeps ten instances of the Ratings service at ~80% utilization. Table 5 reports both the average CPU usage (as a percentage of one vCPU) and memory usage of the sidecar across three setups: a baseline "forward-only" proxy that simply forwards traffic to the application container without any offloading logic, and two offloading configurations. The first is 10% offload, which serves as a baseline for typical operation, as Spandana offloads at most 5% of requests in practice (see Section 8.1). The second is 50% offload, capturing worst-case instantaneous spikes under the trace in Figure 1.

Relative to the baseline, the SLO Director drives CPU usage of the sidecar container from 8.1% to 9.6% at 10% offload (+1.5%) and from 8.1% to 13.6% at 50% offload (+5.5%). Memory overhead is +1.1 MB at 10% offload and +2.5 MB at 50% offload. We did not observe any measurable increase in request latency across these setups, indicating that the offloading logic introduces no noticeable delay beyond the application's normal processing time. These results show that the SLO Director's overhead scales with offloaded volume but remains very small in absolute terms.

SRO Overhead. The centralized SRO executes every two minutes, with each iteration completing within a few hundred milliseconds and consuming less than 1% of a vCPU with negligible memory footprint. This overhead is comparable to other cluster-level autoscaling components such as the k8s HPA controller. Because it runs outside the critical request path, the SRO's runtime cost is operationally invisible.

9 Related work

Hybrid Frameworks with Serverless. A number of works have sought to combine VMs with FaaS. Some target data query tasks [14, 38, 51, 64], while others serve cloud workloads [34, 48, 71, 74]. All of these use serverless functions only to process unexpected large load spikes, or to bridge throughput gaps during new VM spin-up. As these systems treat serverless functions only as a fail-safe, they are unable to (1) accommodate fine-grained load variations, and (2) seize the cost-saving opportunities through the use of serverless functions to avoid over-provisioning VMs.

Libra [55] goes beyond the aforementioned works in that it continuously offloads to serverless. However, as our work shows, Libra is unable to accommodate fine-grained load variations because its policies conflate provisioning for cost and SLO, ultimately falling short on both fronts.

UnFaaSener [59] is a conceptual opposite of Spandana. It offloads some serverless functions to VMs when the VM utilization is predicted to be low. Unlike Spandana, UnFaaSener is not concerned with SLO and tries to lower cost opportunistically rather than systematically.

Hybrid Frameworks with VMs. Numerous prior works seek to optimize cost and performance by combining standard VMs with alternative VM types, such as burstable instances [13, 22, 35] or spot instances [43, 46, 63, 68, 70]. Given a highly volatile load, burstable VMs experience frequent use and hence are unable to accumulate credits. To accumulate credits, burstable instances must be under-utilized, which increases cost (Section 8). Spot VMs can be revoked at any time. While cheaper than regular VMs, they share the same limitations as regular VMs when it comes to achieving high utilization under SLO (Section 3.1). Despite the limitations of both spot and burstable VM types, Spandana can leverage them to further lower deployment cost, provide resilience if a VM goes down, and guarantee SLO under fluctuating load.

Resource allocation for online services. Prior research has explored resource management for cloud services [16, 26, 29, 30, 36, 37, 39, 40, 50, 53, 66, 72, 73] to balance performance and efficiency. These works focus on efficient distribution of resources (typically, VMs) among a set of services. Spandana operates on top of such cluster-scale resource allocators by ensuring that individual services can meet SLO in the face of fluctuating load.

10 Conclusion

To address the sub-second load volatility and the resulting trade-off between SLO compliance and cost efficiency, this work introduced Spandana, a two-level architecture that decouples SLO enforcement from cost optimization. At the VM instance level, a lightweight controller steers requests to the VM if it can meet the SLO target; otherwise, the request is redirected to FaaS. At the global level, Spandana's resource allocator optimizes the entire deployment for cost taking into account VMs, serverless and the shape of the load. Spandana provides strict SLO adherence, high CPU utilization on the VMs and lower cost than SOTA schemes. An extended version of the paper containing additional studies is available on arXiv [25].

Acknowledgments

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. Marios Kogias is supported by an ERC Starting Grant (ID: 101220079)

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

References

[1] [n. d.]. Envoy Proxy. https://www.envoyproxy.io/.

[2] [n. d.]. Fluent Bit. https://fluentbit.io/.

[3] 2021. AWS Lambda – Container Image Support. https://aws.amazon.com/blogs/aws/new-for-aws-lambda-container-image-support/. Accessed 10. Sep. 2025.

[4] 2022. AWS Lambda Web Adapter. https://github.com/awslabs/aws-lambda-web-adapter. Accessed 10. Sep. 2025.

[5] 2022. Serverless Adapter. https://github.com/H4ad/serverless-adapter.

[6] 2024. Archive Team: The Twitter Stream Grab. https://archive.org/details/twitterstream. Accessed 9. Jan. 2024.

[7] 2024. "BookInfo Application". https://istio.io/latest/docs/examples/bookinfo/. Accessed 20. May. 2024.

[8] 2025. 2025 Kubernetes Cost Benchmark Report. https://cast.ai/kubernetes-cost-benchmark/. Accessed 19. Aug. 2025.

[9] 2025. Discovering Services. https://kubernetes.io/docs/concepts/services-networking/service/#discovering-services. Accessed: 18.Sept.2025.

[10] 2025. Horizontal Pod Autoscaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. Accessed: 18.Sept.2025.

[11] 2025. Services, Load Balancing, and Networking. https://kubernetes.io/docs/concepts/services-networking/.

[12] Amazon Web Services. 2025. Burstable Performance Instances and CPU Credits. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-credits-baseline-concepts.html. Accessed: 18.Sept.2025.

[13] Ataollah Fatahi Baarzi, Timothy Zhu, and Bhuvan Urgaonkar. 2019. BurScale: Using Burstable Instances for Cost-Effective Autoscaling in the Public Cloud. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 126–138. doi:10.1145/3357223.3362706

[14] Haoqiong Bian, Tiannan Sha, and Anastasia Ailamaki. 2023. Using Cloud Functions as Accelerator for Elastic Data Analytics. *Proc. ACM Manag. Data* 1, 2, Article 161 (jun 2023), 27 pages. doi:10.1145/3589306

[15] Peter Bodik, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. 2010. Characterizing, modeling, and generating workload spikes for stateful services. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 241–252. doi:10.1145/1807128.1807166

[16] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. 2014. Apollo: scalable and coordinated scheduling for cloud-scale computing. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 285–300.

[17] Jiadong Chen, Xiao He, Hengyu Ye, Fuxin Jiang, Tieying Zhang, Jianjun Chen, and Xiaofeng Gao. 2025. Online ensemble transformer for accurate cloud workload forecasting in predictive auto-scaling. *arXiv preprint arXiv:2508.12773* (Aug. 2025). arXiv:2508.12773 [cs.LG]

[18] Jiagan Cheng, Yilong Zhao, Zijun Li, Quan Chen, Weihao Cui, and Minyi Guo. 2023. Microless: Cost-efficient Hybrid Deployment of Microservices on IaaS VMs and Serverless. In *2023 IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2303–2310.

[19] Cloud Native Computing Foundation. 2022. *Service Meshes Are on the Rise — But Greater Understanding and Experience Are Required.* Technical Report. Cloud Native Computing Foundation. Accessed: 2025-09-18. https://www.cncf.io/wp-content/uploads/2022/05/CNCF_Service_Mesh_MicroSurvey_Final.pdf

[20] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. 2021. SeBS: a serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference* (Québec city, Canada) *(Middleware '21)*. Association for Computing Machinery, New York, NY, USA, 64–78. doi:10.1145/3464298.3476133

[21] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 153–167. doi:10.1145/3132747.3132772

[22] Jaime Dantas, Hamzeh Khazaei, and Marin Litoiu. 2021. BIAS Autoscaler: Leveraging Burstable Instances for Cost-Effective Autoscaling on Cloud Systems. In *Proceedings of the Seventh International Workshop on Serverless Computing (WoSC7) 2021* (Virtual Event, Canada) *(WoSC '21)*. Association for Computing Machinery, New York, NY, USA, 9–16. doi:10.1145/3493651.3493667

[23] Datadog. 2025. Container Report. https://www.datadoghq.com/container-report/. Accessed: 18.Sept.2025.

[24] Dilina Dehigama, Shyam Jesalpura, Antonios Katsarakis, Marios Kogias, Rakesh Kumar, and Boris Grot. 2024. Composing microservices and serverless for load resilience. 1–8. The 2nd Workshop on SErverless Systems, Applications and MEthodologies, SESAME 2024 ; Conference date: 22-04-2024 Through 22-04-2024. https://sesame2024.github.io/

[25] Dilina Dehigama, Shyam Jesalpura, Zeyu Xu, Marton Nemeth, Shengda Zhu, Marios Kogias, and Boris Grot. 2026. Spandana: Reconciling Strict SLOs with Low Cost under Fine-Grained Load Fluctuations. arXiv:2606.30533 [cs.DC] https://arxiv.org/abs/2606.30533

[26] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. 2012. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12)*. Association for Computing Machinery, New York, NY, USA, 99–112. doi:10.1145/2168836.2168847

[27] Robert G. Gallager. 2013. *Stochastic Processes: Theory for Applications.* Cambridge University Press. https://books.google.co.uk/books?id=ERLrAQAAQBAJ

[28] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A. Kozuch. 2012. AutoScale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers. *ACM Trans. Comput. Syst.* 30, 4, Article 14 (Nov. 2012), 26 pages. doi:10.1145/2382553.2382556

[29] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant resource fairness: fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA) *(NSDI'11)*. USENIX Association, USA, 323–336.

[30] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. 2016. Firmament: fast, centralized cluster scheduling at scale. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 99–115.

[31] Google Cloud. 2025. Scalable Apps and Autoscaling in Google Kubernetes Engine. https://cloud.google.com/kubernetes-engine/docs/learn/scalable-apps-autoscale. Accessed: 18.Sept.2025.

[32] Grafana Labs. 2025. *Grafana Loki: Like Prometheus, but for logs.* Grafana Labs.

[33] Grafana Labs. 2025. k6 Documentation. https://grafana.com/docs/k6/latest/.

[34] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Mahmut Taylan Kandemir, Bhuvan Urgaonkar, George Kesidis, and Chita Das. 2019. Spock: Exploiting Serverless Functions for SLO and Cost Aware Resource Procurement in Public Cloud. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. 199–208. doi:10.1109/CLOUD.2019.00043

[35] Rubaba Hasan, Timothy Zhu, and Bhuvan Urgaonkar. 2024. AutoBurst: Autoscaling Burstable Instances for Cost-effective Latency SLOs. In *Proceedings of the 2024 ACM Symposium on Cloud Computing* (Redmond, WA, USA) *(SoCC '24)*. Association for Computing Machinery, New York, NY, USA, 243–258. doi:10.1145/3698038.3698530

[36] Md Rajib Hossen, Mohammad A. Islam, and Kishwar Ahmed. 2022. Practical Efficient Microservice Autoscaling with QoS Assurance. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing* (Minneapolis, MN, USA) *(HPDC '22)*. Association for Computing Machinery, New York, NY, USA, 240–252. doi:10.1145/3502181.3531460

[37] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. 2009. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 261–276. doi:10.1145/1629575.1629601

[38] Aman Jain, Ata F. Baarzi, George Kesidis, Bhuvan Urgaonkar, Nader Alfares, and Mahmut Kandemir. 2020. SplitServe: Efficiently Splitting Apache Spark Jobs Across FaaS and IaaS. In *Proceedings of the 21st International Middleware Conference* (Delft, Netherlands) *(Middleware '20)*. Association for Computing Machinery, New York, NY, USA, 236–250. doi:10.1145/3423211.3425695

[39] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Íñigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. 2016. Morpheus: towards automated SLOs for enterprise clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 117–134.

[40] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. 2019. GrandSLAm: Guaranteeing SLAs for Jobs in Microservices Execution Frameworks. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 34, 16 pages. doi:10.1145/3302424.3303958

[41] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Guodong Yang, and Chengzhong Xu. 2022. The Power of Prediction: Microservice Auto Scaling via Workload Learning. In *Proceedings of the 13th Symposium on Cloud Computing* (San Francisco, California) *(SoCC '22)*. Association for Computing Machinery, New York, NY, USA, 355–369. doi:10.1145/3542929.3563477

[42] Ming Mao and Marty Humphrey. 2012. A Performance Study on the VM Startup Time in the Cloud. In *2012 IEEE Fifth International Conference on Cloud Computing*. 423–430. doi:10.1109/CLOUD.2012.103

[43] Ziming Mao, Tian Xia, Zhanghao Wu, Wei-Lin Chiang, Tyler Griggs, Romil Bhardwaj, Zongheng Yang, Scott Shenker, and Ion Stoica. 2025. SkyServe: Serving AI Models across Regions and Clouds with Spot Instances. In *Proceedings of the Twentieth European Conference on Computer Systems* (Rotterdam, Netherlands)

(EuroSys '25). Association for Computing Machinery, New York, NY, USA, 159–175. doi:10.1145/3689031.3717459

[44] Maxday. 2025. Lambda Performance Benchmark. https://maxday.github.io/lambda-perf/. Accessed: 18.Sept.2025.

[45] Chunyang Meng, Haogang Tong, Tianyang Wu, Maolin Pan, Yang Yu, and Yi Jiang. 2024. BASE: Burst-adaptive autoscaling via stacked ensembles for SLO assurance and cost efficiency. *arXiv preprint arXiv:2402.12962* (Feb. 2024). arXiv:2402.12962 [cs.SE]

[46] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. 2024. SpotServe: Serving Generative Large Language Models on Preemptible Instances. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 1112–1127. doi:10.1145/3620665.3640411

[47] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 115–130. doi:10.1145/3318464.3389758

[48] Joe H. Novak, Sneha Kumar Kasera, and Ryan Stutsman. 2019. Cloud Functions for Fast and Robust Resource Auto-Scaling. In *2019 11th International Conference on Communication Systems & Networks (COMSNETS)*. 133–140. doi:10.1109/COMSNETS.2019.8711058

[49] Pradeep Padala, Kai-Yuan Hou, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. 2009. Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM European Conference on Computer Systems* (Nuremberg, Germany) *(EuroSys '09)*. Association for Computing Machinery, New York, NY, USA, 13–26. doi:10.1145/1519065.1519068

[50] Jun Woo Park, Alexey Tumanov, Angela Jiang, Michael A. Kozuch, and Gregory R. Ganger. 2018. 3Sigma: distribution-based cluster scheduling for runtime uncertainty. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 2, 17 pages. doi:10.1145/3190508.3190515

[51] Matthew Perron, Raul Castro Fernandez, David DeWitt, Michael Cafarella, and Samuel Madden. 2023. Cackle: Analytical Workload Cost and Performance Stability With Elastic Pools. *Proc. ACM Manag. Data* 1, 4, Article 233 (dec 2023), 25 pages. doi:10.1145/3626720

[52] Satya Nagamani Pothu and Swathi Kailasam. 2025. Hybrid workload prediction for improved autoscaling in IaaS clouds: An ARIMA-OLSTM approach. *Ing. Syst. D Inf.* 30, 04 (April 2025), 961–970.

[53] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 805–825. https://www.usenix.org/conference/osdi20/presentation/qiu

[54] Haoran Qiu, Weichao Mao, Chen Wang, Hubertus Franke, Alaa Youssef, Zbigniew T. Kalbarczyk, Tamer Başar, and Ravishankar K. Iyer. 2023. AWARE: Automate Workload Autoscaling with Reinforcement Learning in Production Cloud Systems. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 387–402. https://www.usenix.org/conference/atc23/presentation/qiu-haoran

[55] Ali Raza, Zongshun Zhang, Nabeel Akhtar, Vatche Isahagian, and Ibrahim Matta. 2021. LIBRA: An Economical Hybrid Approach for Cloud Applications with Strict SLAs. In *2021 IEEE International Conference on Cloud Engineering (IC2E)*. 136–146. doi:10.1109/IC2E52221.2021.00028

[56] Benjamin Reidys, Pantea Zardoshti, Íñigo Goiri, Celine Irvene, Daniel S. Berger, Haoran Ma, Kapil Arya, Eli Cortez, Taylor Stark, Eugene Bak, Mehmet Iyigun, Stanko Novakovic, Lisa Hsu, Karel Trueba, Abhisek Pan, Chetan Bansal, Saravan Rajmohan, Jian Huang, and Ricardo Bianchini. 2025. Coach: Exploiting Temporal Patterns for All-Resource Oversubscription in Cloud Platforms. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 164–181. doi:10.1145/3669940.3707226

[57] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. 2011. Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting. In *Proceedings of the IEEE International Conference on Cloud Computing*. IEEE, 159–166.

[58] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. 2020. Autopilot: workload autoscaling at Google. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 16, 16 pages. doi:10.1145/3342195.3387524

[59] Ghazal Sadeghian, Mohamed Elsakhawy, Mohanna Shahrad, Joe Hattori, and Mohammad Shahrad. 2023. UnFaaSener: Latency and Cost Aware Offloading of Functions from Serverless Platforms. In *Proceedings of the 2023 USENIX Annual Technical Conference*. USENIX Association, Boston, MA, USA. https://www.usenix.org/conference/atc23/presentation/sadeghian

[60] Sultan Mahmud Sajal, Timothy Zhu, Bhuvan Urgaonkar, and Siddhartha Sen. 2024. TraceUpscaler: Upscaling Traces to Evaluate Systems at High Load. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 942–961. doi:10.1145/3627703.3629581

[61] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad

[62] Danfeng Shan, Fengyuan Ren, Peng Cheng, and Ran Shu. 2016. Microburst in Data Centers: Observations, Implications, and Applications. arXiv:1604.07621 [cs.NI] https://arxiv.org/abs/1604.07621

[63] Prateek Sharma, David Irwin, and Prashant Shenoy. 2017. Portfolio-driven Resource Management for Transient Cloud Servers. *Proc. ACM Meas. Anal. Comput. Syst.* 1, 1, Article 5 (June 2017), 23 pages. doi:10.1145/3084442

[64] Won Wook Song, Taegeon Um, Sameh Elnikety, Myeongjae Jeon, and Byung-Gon Chun. 2023. Sponge: Fast Reactive Scaling for Stream Processing with Serverless Frameworks. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 301–314. https://www.usenix.org/conference/atc23/presentation/song

[65] Dmitrii Ustiugov, Theodor Amariucai, and Boris Grot. 2021. Analyzing Tail Latency in Serverless Clouds with STeLLAR. In *Proceedings of the 2021 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE.

[66] Zibo Wang, Pinghe Li, Chieh-Jan Mike Liang, Feng Wu, and Francis Y. Yan. 2024. Autothrottle: a practical bi-level approach to resource management for SLO-targeted microservices. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation* (Santa Clara, CA, USA) *(NSDI'24)*. USENIX Association, USA, Article 9, 17 pages.

[67] WikiBench Project. 2025. WikiBench: A Distributed Wikipedia Access Benchmark. http://www.wikibench.eu/?page_id=60. Accessed: 18.Sept.2025.

[68] Zhanghao Wu, Wei-Lin Chiang, Ziming Mao, Zongheng Yang, Eric Friedman, Scott Shenker, and Ion Stoica. 2024. Can't be late: optimizing spot instance savings under deadlines. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation* (Santa Clara, CA, USA) *(NSDI'24)*. USENIX Association, USA, Article 11, 19 pages.

[69] Bartek Wydrowski, Robert Kleinberg, Stephen M. Rumble, and Aaron Archer. 2024. Load is not what you should balance: Introducing Prequal. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 1285–1299. https://www.usenix.org/conference/nsdi24/presentation/wydrowski

[70] Fangkai Yang, Lu Wang, Zhenyu Xu, Jue Zhang, Liqun Li, Bo Qiao, Camille Couturier, Chetan Bansal, Soumya Ram, Si Qin, Zhen Ma, Íñigo Goiri, Eli Cortez, Terry Yang, Victor Rühle, Saravan Rajmohan, Qingwei Lin, and Dongmei Zhang. 2023. Snape: Reliable and Low-Cost Computing with Mixture of Spot and On-Demand VMs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 631–643. doi:10.1145/3582016.3582028

[71] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. 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* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 1049–1062.

[72] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G. Edward Suh, and Christina Delimitrou. 2021. Sinan: ML-based and QoS-aware resource management for cloud microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 167–181. doi:10.1145/3445814.3446693

[73] Yanqi Zhang, Zhuangzhuang Zhou, Sameh Elnikety, and Christina Delimitrou. 2024. Analytically-Driven Resource Management for Cloud-Native Microservices. arXiv:2401.02920 [cs.DC] https://arxiv.org/abs/2401.02920

[74] Ziming Zhao, Mingyu Wu, Jiawei Tang, Binyu Zang, Zhaoguo Wang, and Haibo Chen. 2023. 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* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 74–87. doi:10.1145/3575693.3575752

[75] Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, XiongChun Duan, Peng He, Arvind Krishnamurthy, Matthew Lentz, Danyang Zhuo, and Ratul Mahajan. 2023. Dissecting Overheads of Service Mesh Sidecars. In *Proceedings of the 2023 ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '23)*. Association for Computing Machinery, New York, NY, USA, 142–157. doi:10.1145/3620678.3624652