Spot LB: Using Spot Instances in Multi-Cloud Environments with Hierarchical Load Balancing While Maintaining SLOs

Seoungdeok Jeon
Siebel School of Computing and Data Science
University of Illinois Urbana-Champaign
USA
sj65@illinois.edu

Mohit Sharma
Siebel School of Computing and Data Science
University of Illinois Urbana-Champaign
India
sharma83@illinois.edu

Adeel Siddiqui
Siebel School of Computing and Data Science
University of Illinois Urbana-Champaign
Singapore
adeelas2@illinois.edu

Cody Talmadge
Siebel School of Computing and Data Science
University of Illinois Urbana-Champaign
USA
codytt2@illinois.edu

ABSTRACT

While cloud computing has revolutionized how organizations utilize computing resources, many enterprises still face several challenges such as high latency, vendor lock-in, and suboptimal cost efficiency due to dependency on a single cloud provider. Multi-cloud computing, which utilizes services from multiple providers, addresses these issues by enhancing latency, availability, and cost management. Additionally, spot instances offer cost optimization but come with the risk of preemption. This paper studies the feasibility of achieving Service Level Objectives (SLOs) using only spot instances in a multi-cloud environment, without using more expensive on-demand or reserved instances. We propose a hierarchical load balancing architecture that dynamically manages spot instances across multiple providers, ensuring service continuity despite preemptions. Our approach combines predictive bidding with diversification strategies and real-time workload distribution, tested using chaos engineering and load injection to ensure resiliency in failure scenarios and under high traffic. We demonstrate that hierarchical load balancing, integrated with predictive bidding and proactive resource allocation, achieves reliable performance alongside significant cost savings using only spot instances. This provides a scalable, flexible, and cost-effective alternative to single-cloud and hybrid cloud approaches for latency-sensitive applications.

I. INTRODUCTION

Cloud computing provides access to a shared pool of computing resources provided by a cloud provider which can be configured with minimal effort [1]. This shift has revolutionized computing economics by enabling organizations to access computing power as a public utility. However, enterprises dependent on one cloud provider can face challenges of high latency, vendor lock-in, hindering optimal cost, reliability, and performance.

To address these issues, multi-cloud computing has emerged as a promising solution [4]. It leverages services from multiple vendors in a unified network infrastructure, avoiding platform complexity [2][7]. This strategy allows enterprises to optimize their performance by reducing latency, increasing availability, and avoiding vendor lock-in. Tools like Terraform further simplify the management of infrastructure across multiple vendors by providing a unified Domain-Specific Language (DSL) that enables DevOps teams to provision resources across different platforms [3].

Studies have shown that redundant web service deployments on different cloud providers can reduce network round trip times (RTT) by over 20% when compared to deploying only on a single cloud [5]. Naturally, these benefits have meant that many enterprises have adopted or are adopting multi-cloud. A survey [6] showed that up to 87% of respondents were using multiple cloud providers and this percentage is only expected to grow.

Despite its advantages, multi-cloud deployment introduces complexities in managing spot instances across providers, particularly when relying exclusively on these cost-effective but preemptible resources. Several multi-cloud deployment architectures exist including, but not limited to brokering architecture, reservoir model, layered multi-cloud, and container architecture [8]. Orchestration tools such as Kubernetes and service meshes such as Istio have made it easy to deploy applications on different cloud providers and connect them together as a single transparent cluster. For example, Istio provides a multi cluster mode that can install the Istio service mesh across multiple Kubernetes clusters [9].

II. SPOT INSTANCES IN MULTI-CLOUD ENVIRONMENTS

With the growth of multi-cloud computing, the adoption of spot instances has become an attractive choice for cloud cost-cutting. Cloud providers make underutilized cloud resources available as spot instances at much lower prices - up to 90% less than those of on-demand and reserved instances [10]. On-demand instances, while flexible, do not require any long-term commitments and, as such, present the most expensive option. In contrast, reserved instances provide a discount of up to 72%, but require a 1-3 year commitment [11]. The use of these spot instances, though cost-effective, comes with risks. They can be preempted - that is, taken away-by the cloud provider with very

little notice, especially during periods of high demand, thus making them less reliable for particular types of workloads.
This risk of preemption can be significant. Several studies have shown that AWS spot instances are frequently preempted due to high demand, with a mean time to failure (MTTF) ranging from a few hours to several days depending on the instance type and region [12]. Some operators (e.g., Google Cloud and Microsoft Azure), do not publish preemption frequency statistics, so a direct comparison of the risk of preemption between different cloud providers is not possible.

To manage these risks, several hybrid systems have emerged, such as SpotKube and SpotWeb, which combine spot instances with more reliable on-demand instances [13] [14]. SpotKube uses an elastic auto-scaler that scales up the on-demand instances dynamically upon preemption of spot instances, ensuring sufficient resources during spot instance failures. Similarly, SpotWeb deploys intelligent over-provisioning whereby it over-provisions extra capacity in anticipation of possible preemptions. It rebalances workloads to on-demand instances as needed to ensure continuity of service and meet Service Level Objectives.

While most of the recent works propose hybrid models combining spot instances with on-demand or reserved instances, our research investigates the feasibility of ensuring workload capacity (measured through SLO compliance) by using only spot instances in a multi-cloud environment. We monitor in real time the availability of spot instances and perform dynamic workload distribution across cloud providers with the goal of developing a resilient solution maintaining the service reliability and minimizing downtimes without incurring the higher costs related to on-demand instances.

In a single cloud environment, relying solely on spot instances can expose services to downtime risks, particularly for latency-sensitive applications. Our proposal is a multi-cloud approach, taking into consideration the independent probability of spot instance preemption across multiple cloud providers and regions. By dynamically distributing workloads across those providers, as spot instances are preempted workload can be shifted from to other clouds or regions that are not experiencing preemption. This allows for the delivery of continuous services without any disruption; it means an efficient solution reliably for microservices and other latency-sensitive applications.

As we treat the possibility of preemption across multiple providers and regions independently, increasing the diversity of clusters decreases the overall risk of workflow downtime. Our objective is to demonstrate that we can meet specific reliability SLOs by leveraging this independence without requiring on-demand/reserved instances or instance overprovisioning. This strategy reduces the potential for any downtime while still retaining the cost-efficient aspects of spot instances.

Our solution further leverages the fluctuating spot instance prices across diverse regions and providers to optimize costs for cost-optimizing multi-cloud approaches, ensuring that enterprises meet their service level objectives at minimal cost while mitigating risks associated with running spot instances.

III. Related Work

This idea of using a multi-cloud setup to increase the resilience of an architecture built on spot instances and reduce the need for over-provisioning or backing them up with on-demand instances is underexplored. Traditionally, organizations over-provision spot instance resources or use on-demand or reserved instance as a backup to ensure their service is up in case failures occur within those spot instances. In contrast, multi-cloud, multi-cluster architectures can provide an alternative by distributing tasks among several providers, thereby improving resilience without demanding unnecessary backup capacity.

Neto J.P.A. et al. addressed this problem through their MULTS architecture, which combines statistical models to predict preemptions and uses checkpointing to save job statuses [15]. By restarting preempted spot instances in another cloud provider or region, this architecture ensures minimal service disruption for long-running tasks. However, they specifically focused on pre-known scheduled tasks, as opposed to on-demand requests, limiting the application of the architecture described in their paper.

Similarly, Qu C. et al. explored the use of geographical load balancing to handle on-demand tasks in multi-cloud environments. Their approach distributed incoming requests to the closest data center, scaling resources as needed, and rerouting requests to other data centers when necessary [16]. While their strategies demonstrated the potential of multi-cloud architectures to provide resilience and minimize the risk of service interruptions, especially when using spot instances, they didn't test in a multi-cloud environment, nor did they focus on meeting specific SLOs with their solution.
As a result, how to exclusively use spot instances in multi-cloud, multi-region environments to meet SLOs remains largely unexplored. Our research uniquely targets meeting SLOs using only spot instances, combining dynamic workload distribution across cloud providers and continuous monitoring of spot instance availability to deliver a resilient, cost-effective solution for latency-sensitive applications.

IV. Proposed Solution

Our proposed solution combines three different pieces together: informed spot price bidding that balances price with preemption, hierarchial load balancing to distribute requests across a multi-cloud environment, and a system to handle overload scenarios when clusters experience capacity reductions due to preemptions.

A. Spot Price Bidding

Bidding for spot instances requires careful consideration of various factors, including budget constraints, workload characteristics, and risk tolerance. Optimizing spot instance utilization involves employing strategies that balance these factors while adapting to dynamic pricing conditions. Our simulation implements predictive bidding through heuristic-based models that analyze historical price trends. These models

use the most recent spot price data, aggregating it over fixed intervals, to compute average prices and predict future fluctuations. The predictions drive bid adjustments dynamically, aiming to balance cost efficiency and availability. For instance, if the recent average spot price shows an increasing trend, the bidding strategy adjusts upward to reduce preemption risk, while downward trends lower bids to optimize costs [17][18][19]. This process ensures cost-effectiveness while reducing the likelihood of preemptions by adapting to real-time market changes.

Although AWS and GCP no longer support spot price bidding, this mechanism remains relevant for Azure instances, where bid prices play a critical role in instance allocation. Azure's spot market relies on the bid price to prioritize allocation among competing users, making predictive bidding an essential tool for managing resource availability.

Another strategy involves diversification, where workloads are distributed across multiple instance types, availability zones, regions, and cloud providers to reduce the impact of price spikes or interruptions in any single market. Research [20] highlights the importance of diversification and other techniques like checkpointing and fallback mechanisms to ensure application resilience.

By combining heuristic-based predictive bidding with diversification and robust interruption handling mechanisms, organizations can effectively leverage spot instances to achieve significant cost savings without compromising application availability. Our proposal integrates these aspects: heuristic-based predictive bidding to adjust bids dynamically and diversification to balance spot prices paid with preemption risk.

B. Hierarchical Load Balancing in Multi-Cloud Environments

Traditional load balancing methods, often relying on centralized approaches, face significant challenges in the dynamic landscape of geographically distributed systems and multi-cloud deployments. As highlighted in "An Edge DNS Global Server Load Balancing for Load Balancing in Edge Computing" [21], these centralized approaches can struggle to efficiently manage traffic across complex, dispersed setups. This limitation is further emphasized in "Dynamic Load Balancing Method Based on DNS for Distributed Web Systems" [22] which warns of potential bottlenecks arising from a single point of failure when handling many servers.

To address these challenges, a shift towards decentralized and hierarchical load balancing is essential. The paper "Global Server Load Balancing with Networked Load Balancers for Geographically Distributed Cloud Datacenters" [23] proposes a hierarchical system with networked load balancers operating at local, regional, and global levels. The paper's authors argue that this approach offers improved scalability, reliability, and reduced latency by directing users to nearby servers.

Hierarchical load balancing (geo → regional → zonal) offers a powerful approach to managing traffic flow in complex deployments, especially when dealing with the dynamic nature of spot instances. Firstly, it minimizes latency by directing users to the closest available resources. Secondly, it enhances availability by providing multiple fallback options in case of failures at any level. If a zone goes down, traffic can be redirected to another zone within the same region. If an entire region fails, traffic can be shifted to a different region. This multi-layered redundancy is crucial when relying on spot instances, which can be interrupted with short notice.

Fortunately, the capabilities of leading cloud providers facilitate this transition. AWS Route 53, Azure Traffic Manager, and Google Cloud DNS offer sophisticated DNS-based load balancing features, including health checks, failover mechanisms, and geolocation routing [24] [25].

Moreover, leveraging these services from multiple providers enhances resilience and avoids vendor lock-in. Google Cloud, for example, advocates for a multi-provider approach, utilizing tools like Terraform and OctoDNS to manage and automate DNS replication across different providers [26]. This strategy ensures that if one provider experiences an outage, the others can seamlessly take over, guaranteeing continuous availability for applications.

By strategically configuring DNS records and utilizing features like weighted routing and failover policies, organizations can create a highly available and responsive infrastructure that seamlessly directs users to the best-performing cloud region. This offers a cost-effective alternative to Anycast while maintaining the flexibility and scalability benefits of a multi-cloud strategy.

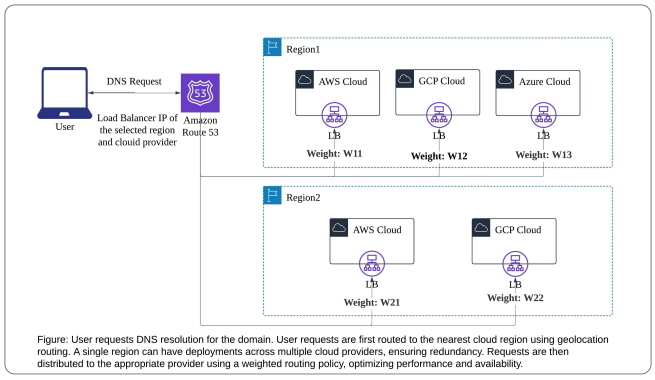

Figure 1: User requests DNS resolution for the domain. User requests are first routed to the nearest cloud region using geolocation routing. A single region can have deployments across multiple cloud providers, ensuring redundancy. Requests are then distributed to the appropriate provider using a weighted routing policy, optimizing performance and availability.

As proof of concept, we explore a novel multi-cloud hierarchical load balancing architecture designed to optimize the utilization of spot instances in a multi-cloud environment. This architecture uses DNS-based Global Server Load Balancing (GSLB) in AWS Route53 that uses hierarchical routing policies (Geolocation → Weighted). When a user makes a request, our system uses geolocation routing to identify their location and direct them to the nearest server region. This minimizes network distance and latency. Within each region,

we use weighted routing to distribute traffic across servers hosted on different cloud providers (like AWS, GCP, and Azure). This ensures we're not reliant on a single provider and can take advantage of the best each has to offer.

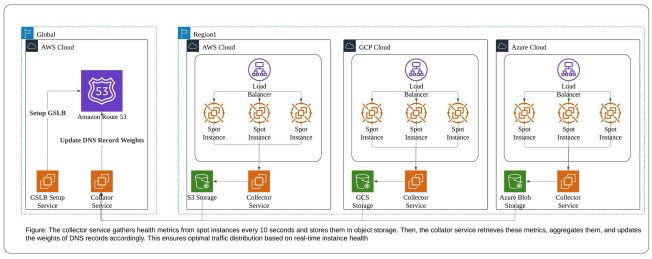

Figure 2: The collector service gathers health metrics from spot instances every 10 seconds and stores them in object storage. Then, the collator service retrieves these metrics, aggregates them, and updates the weights of DNS records accordingly. This ensures optimal traffic distribution based on real-time instance health.

The weights used in weighted routing policy are not static; they are dynamically adjusted based on the real-time health and performance of the underlying infrastructure. To achieve this dynamic weight adjustment, a collector service is deployed within each cloud provider's environment in every region. This service periodically gathers key health metrics from spot instances, including CPU utilization, RAM usage, storage capacity, ingress traffic, and the number of active instances. These metrics are then pushed to the respective cloud provider's object storage (e.g., AWS S3, GCP Cloud Storage, Azure Blob Storage).

A central collator service is responsible for aggregating these regional metrics from all object storage locations. It analyzes the collected data and computes new weights for the DNS records, ensuring that traffic is directed towards the most performant and cost-effective instances available. This dynamic adjustment process optimizes resource utilization, maximizes application performance, and minimizes costs by leveraging the strengths of each cloud provider and adapting to changing conditions in real-time.

C. Handling Overload and Capacity Reduction

In systems built on spot instances, a key challenge is handling overload scenarios when clusters experience capacity reductions due to preemptions. One challenge with DNS load balancers is the latency introduced by caching DNS results [27]. In a system built on spot instances, a significant number of instances within a single cluster could be preempted by the cloud provider simultaneously, significantly reducing that cluster's capacity. However, due to DNS record caching, for several minutes the DNS load balancer would continue to send load to this cluster assuming it had its original capacity. As a result, there is the need for the cluster to be able to handle overload situations itself that the DNS load balancer isn't yet aware of. While this is not a commonly researched problem, because it's uncommon to only deploy spot instances, a similar problem is often encountered in edge/fog computing.

In edge computing/fog computing, the system is often set up in a decentralized manner, and the capacity of edge computing servers is limited (often with no scaling available), causing them to be overloaded when flash crowds appear. As a result, they often need to solve the problem of consistent capacity and growing load. Although our problem sounds different at first glance (we are solving the problem of shrinking capacity and consistent load), the underlying problem of capacity being less than load and needing to handle the overload is the same in both cases.

Multiple approaches have been explored to address this problem. Puthal D. et al describe a system where edge data centers (EDCs) autonomously decide if they can handle the current request load, and if not, redirect tasks to other EDCs without going through a centralized load balancer [28]. When an EDC becomes overloaded, it broadcasts a load balancing request to its neighboring EDCs. Based on the responses it receives; it decides independently which neighboring EDCs have the capacity to handle additional tasks.

In a different approach, Li Y. et al describe a decentralized system where each cluster contains two task schedulers. First, an inter-cluster task scheduler decides which cluster should handle the request and sends it to that cluster's intra-cluster task scheduler [29]. The intra-cluster task scheduler then assigns the task to a particular VM/server within its cluster.

Tahmasebi-Pouya N. et al describe a simpler approach - a Blind Load-Balancing Algorithm (BLBA) where tasks by default are assigned to the geographically closest node [30]. By default, nodes handle a task if their load is below a threshold. Otherwise, they send the request to a randomly selected neighboring node, without checking the neighbor's load status, make it "blind". However, problems with this approach arise when multiple nodes are overloaded, potentially leading to a chain of rejections.

Finally, Beraldi R. et al [31] describe a solution where by default a fog node executes the jobs it receives directly and without any load balancing action. However, when its current workload is above a pre-defined threshold, the node probes other random nodes in the area and sends the job to the least loaded one, assuming that its workload is lower than the original node. They demonstrate how this approach provides acceptable results at a much lower latency and with fewer overhead costs than other algorithms.

Our approach models the approach described by Beraldi R. et al. We plan for each cluster to set a threshold to determine when it is overloaded, which may include predicting future overload based on upcoming preemptions. Once this threshold is met, the cluster's load balancer will probe a predefined number of other clusters nearby to determine whether they are overloaded. If they are not overloaded, it will redirect the request to the least-loaded of the F clusters.

V. Experimental Evaluation

To test our solution, we have run an experiment of it in both a cloud simulator as well as in the real world in AWS and

Google Cloud. The following sections describe the experiments done and their results.

A. *CloudSim*

CloudSim [44][45] is a toolkit for simulating and modeling cloud computing environments, offering capabilities to model data centers, VMs, container deployments, resource provisioning policies such as elastic autoscaling, and more. To evaluate SpotLB on CloudSim, we required additional features not yet available in CloudSim. Consequently, we have extended the existing CloudSim codebase to include the following enhancements:

Spot Instances: Spot instances are essential to our idea for reducing application costs while maintaining SLOs. We developed the SpotHddVm [46] class by extending the existing HddVm class, to support dynamic pricing updates for instances and termination notifications from the cloud provider when an instance is pre-empted.

Weighted Round Robin Load Balancer [47]: Each application load balancer uses a weighted round-robin policy to allocate requests to managed spot instances. As these instances can be heterogeneous, they are weighted by default according to RAM capacity. Users can also provide a custom comparator function to customize weighting policies.

Global (or DNS) Load Balancer: Implemented using CloudSim's IEntryPoint interface specification, this load balancer routes requests to the closest cloud provider based on latency and avoids providers with terminated or fully utilized spot instances. These capabilities were incorporated into the SpotLBEntryPoint [48] class.

Request Transfer for Terminating Instances: When a request arrives at a spot instance scheduled for termination, it can be transferred to the next optimal provider. Practically, we would use a shared memory store or some other mechanism so that VMs would know the other providers' load balancer DNS names; for simulation purposes, we achieved this by rescheduling the request back to the entry point.

Experimental Setup [52]:

1. We created three data centers across three continents, each representing different cloud providers.
2. Each datacenter contained a heterogeneous number of spot instances, each with 512 MB RAM, 1 vCPU at 250 MIPS, and 2 GB storage.
3. Three load balancers were deployed, each associated with a WebBroker under a different datacenter.
4. A global entry point was set up to route requests to the optimal cloud provider's load balancer, selecting the best one based on latency and availability of non-terminated spot instances. This entry point simulates AWS Route53 for DNS-based load balancing with additional features to deprioritize over-utilized or terminated instances.
5. We generated 300 requests (cloudlets in CloudSim) with random values for CPU, RAM, and I/O consumption, using a Gaussian distribution.
6. During the simulation, each provider's spot instances faced a 10% termination probability.

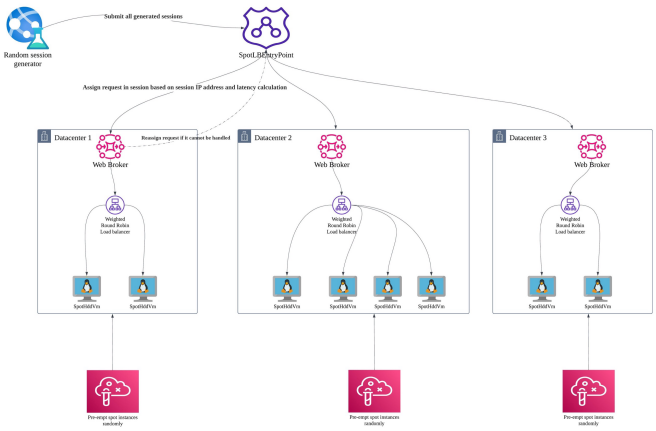

Figure 3: SpotLB design on CloudSim

Our preliminary evaluation indicates that this setup increases reliability, measured by the reduced number of failed requests, with minimal latency introduced (1 unit in terms of CloudSim's clock time) compared to configurations using only spot or regular instances on a single cloud provider.

The work we plan to do to make the simulation more comprehensive includes:

1. Scaling the number of requests to much larger than 300/second.
2. Add capability to the central load balancer (entry point) to choose the cloud provider with lower priced spot instances.
3. Provide knobs to the user to choose whether the central load balancer should optimize for request latency based on geo-proximity, cost, or both.
4. Fix up log collection in CloudSim and make it more granular so that we can have a better assessment of request latencies and failed requests.

B. *Proof of concept on AWS and Google Cloud*

In addition to demonstrating SpotLB in CloudSim, we also developed and implemented a test of SpotLB in the cloud. A true full-scale implementation of SpotLB would require multiple components, including a DNS-based load balancer with adjustable weights, intelligent spot price bidding capabilities, and several clusters spread across multiple cloud providers, each with their own per-cluster load balancer and multiple servers (running on spot instances). While we haven't built a full industry-scale example, we have created a proof-of-concept test version that includes the following components:

DNS-Based Load Balancer: This component uses DNS-based Global Server Load Balancing (GSLB) in AWS Route53 to route traffic based on user location and dynamically adjusted weights. The Route 53 configuration we propose is as follows:

1. Hosted Zone: A hosted zone is created on Route 53 for our domain.
2. Geolocation Record Sets: Separate record sets are created for each geographic region (e.g., us-east-1, us-west-1, etc.).
3. Weighted Record Set: Within each geolocation record set, multiple "A" records are created, each pointing to the static IP of a regional load balancer in that region. Weights are assigned to each record, allowing for fine-grained traffic distribution.

Mock DNS-Based Load Balancer: While we have done some preliminary testing with the AWS Route 53 DNS-base load balancer, for the sake of simplicity and ease of testing we have focused most of our testing on using a custom-built load balancer to simulate DNS-based load balancing. Our central load balancer is set up to evenly distribute requests between two different cluster load balancers by distributing incoming requests to the IP addresses of these two cluster load balancers randomly. This doesn't perfectly simulate AWS Route 53 for several reasons. For example, DNS results are cached by a client, so for AWS Route 53 a single client will continue to send requests to the same cluster load balancer until the cache expires. Additionally, our centralized load balancer is not set up to rebalance weights as servers are preempted from a cluster in the way that we plan for AWS Route 53 to do. However, we don't think these differences are substantial enough to make the results of our experiments invalid. For example, the fact that our centralized load balancer sends new requests to different servers even for the same client allows us to more easily simulate multiple different clients from one computer. Additionally, the inability of our centralized load balancer to rebalance also reflects the reality of cached DNS records (meaning the updated weights at AWS Route 53 won't immediately propagate through) allowing us to simulate how this DNS caching impacts performance.

Cloud Clusters: Our experiments focused on testing the performance of two different clusters in four different scenarios:

1. Across the same Availability Zone in AWS
2. Across two different Availability Zones in AWS
3. Across two different regions in AWS
4. Across two different cloud providers (AWS and Google Cloud)

To get a working cloud clusters set up we created a VPC and a subnet for the clusters within both AWS and Google Cloud with the per-cluster load balancers and servers communicating within a subset using private IP addresses. Requests to a cluster are sent to the cluster load balancer by the mock DNS-based load balancer, and internal communication between the cluster load balancer and the servers happens only over private IP addresses.

To allow easily testing different situations and conditions, we created three different machine images: one for the mock DNS-based load balancer, one for the cluster load balancer, and one for the servers. We also set up a Terraform configuration file for each scenario that automatically starts up the required instances for that scenario. It starts three on demand instances: one for the mock DNS-based load balancer and one for each per-cluster load balancer as well as eight spot instances – for servers within each cluster.

Per-Cluster Load Balancer: The per-cluster load balancers were set up using a t4g.nano on demand EC2 instance (e2-micro on Google Cloud), chosen to minimize costs. On demand was chosen as the instance type as we are only planning to demonstrate how our algorithm handles preemptions of servers, not a preemption of load balancers. A custom Flask server, run using gunicorn with 10 workers and 200 threads automatically starts on the server using a Linux service. The Flask server has three different endpoints [49]:

- /server_status - This POST endpoint receives data from each connected server every five seconds containing the server's IP address, average CPU usage, average request duration, and last updated time. This data is stored in Redis with a fifteen second expiration, automatically removing unresponsive (e.g., preempted) servers. Redis is used to sync the status information across all gunicorn workers.
- /all_server_status - This GET endpoint reports the status of all connected servers using data from Redis, including their IP address, average CPU usage, average request duration, and last updated time and is used to monitor the load balancer by validating it is evenly spreading load across the cluster.
- / - This GET endpoint handles requests by sending them to one of the active servers in the Redis server list. If no servers are currently available, an error message is returned. Assuming servers are available, it selects two servers at random from the list, routing the request to the one with the lower load (based on the number of active requests). If both servers are above a configurable threshold (for example >5 active requests and >90% CPU utilization), then the cluster is determined to be overloaded and the request is routed to the other cluster.

The custom Flask load balancer was used to allow us to easily implement custom load balancing algorithms as we develop out our solution. We recognize that the implementation of this load balancer is likely very inefficient compared to a purpose-built load balancer, and if we were to implement this solution in the real world a more robust load balancer would likely be used. To demonstrating our solution, however, we think the custom-built load balancer is sufficient.

Server: Each server was set up using a t4g.micro EC2 spot instance (e2-highcpu-2). These instance types were chosen due to low cost and relatively low CPU/memory capabilities. For the sake of our demonstration, low CPU/memory capabilities are not an issue as they allow us to test server overload more easily. The server consists of two different pieces that are simultaneously running, a reporting system and a Flask server.

The server's reporting system [50] automatically calls the load balancer's /all_server_status endpoint described above with a POST request containing the server's IP address, average CPU usage, average request duration, and last updated time every five seconds.

The Flask server [51] runs on the / endpoint and handles requests. Load is simulated with a high CPU loop that generates the first x prime numbers, where x was chosen to take ~0.25 seconds on a single thread on the selected EC2 instance through empirical testing. This allows us to easily overload the server with a handful of requests per second, letting us demonstrate the effectiveness of load balancing with a relatively small number of active requests. Once the load has been simulated, the server responds to the load balancer with its IP address and the total number of requests it has handled.

In the real world an actual task would be the load-heavy part of the request, but this logic was included to allow us to quickly overload the server with multiple requests to demonstrate that the load balancer is able to distribute load across multiple servers.

VI. Results

The following sections describe the results of the experiments done and described above:

A. CloudSim

To make a fair comparison, we run the simulation in two broad scenarios:

1. A multi-cloud setup of 3 datacenters – each having its own load balancer (LB) and spot instances, with the request assignment to the instances managed by the load balancer. **This is our baseline.**
2. A multi-cloud setup of 3 datacenters with a global (DNS) load balancer and the capability for each LB in a datacenter to be able to transfer a request to a different LB in a different datacenter, apart from managing the request assignment to the instances under it. **This represents the SpotLB setup.**

In a practical setup, the 3 datacenters would represent application deployment on different cloud providers in the same or different geographic region.

Before we delve into the results of the simulation, the following are important points to note about the simulation:

1. The workload we choose is IO and memory heavy, rather than being CPU-intensive. This represents modern web/API workloads well as they are mostly data intensive.
2. When the memory utilization of any VM instance reaches 100%, CloudSim terminates that instance.
3. We do not use any autoscaling policies in the baseline or modified (SpotLB) scenario since they would not alter the core benefit offered by SpotLB - the ability to transfer requests to other load balancers, as autoscaling is finite.
4. We re-provision a new instance when it has been pre-empted by the cloud provider.

We ran the baseline & SpotLB versions with 2 significantly high requests per seconds (RPS) requirements: 1000 & 2000 with an SLO of 500 CloudSim internal clock time, on 3 configurations:

1. Total memory capacity: 4 GB
   a. Datacenter A: 2 spot instances with 512 MB, 2 vCPUs each
   b. Datacenter B: 4 spot instances with 512 MB, 2 vCPUs each
   c. Datacenter C: 2 spot instances with 512 MB, 2 vCPUs each
2. Total memory capacity: 8 GB
   a. Datacenter A: 2 spot instances with 1024 MB, 2 vCPUs each
   b. Datacenter B: 4 spot instances with 1024 MB, 2 vCPUs each
   c. Datacenter C: 2 spot instances with 1024 MB, 2 vCPUs each
3. Total memory capacity: 16 GB
   a. Datacenter A: 2 spot instances with 2048 MB, 2 vCPUs each
   b. Datacenter B: 4 spot instances with 2048 MB, 2 vCPUs each
   c. Datacenter C: 2 spot instances with 2048 MB, 2 vCPUs each

A background thread was responsible for pre-empting spot instances in each datacenter with a probability of 10%.

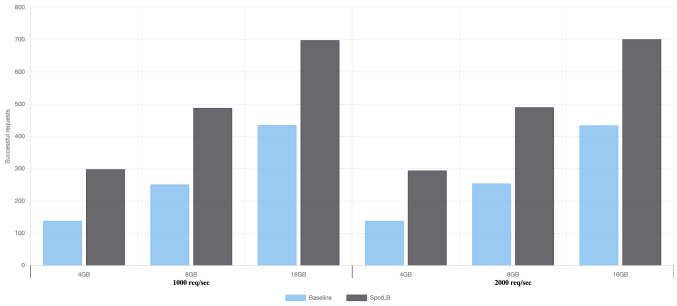

Figure 4: Comparing number of successful requests for 3 different configurations with 2 different RPS for the baseline and SpotLB versions on CloudSim.

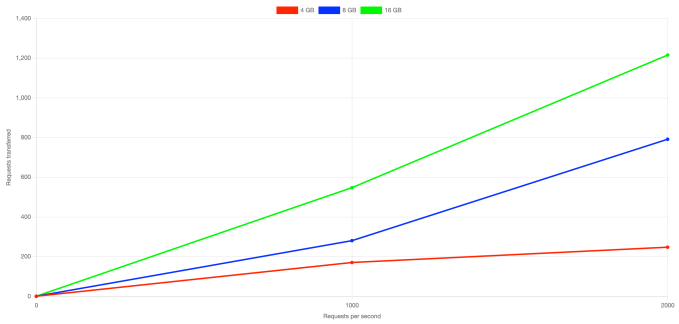

Figure 5: Number of requests transferred in SpotLB for different combinations of RPS and configurations

Version	RPS	Configuration	Successful requests	SLO violations	Requests transferred
Baseline	1000	4 GB	138	0	N/A
	1000	8 GB	251	0	N/A
	1000	16 GB	435	0	N/A
	2000	4 GB	138	0	N/A
	2000	8 GB	254	0	N/A
	2000	16 GB	434	0	N/A
SpotLB	1000	4 GB	298	0	170
	1000	8 GB	488	0	280
	1000	16 GB	698	0	547
	2000	4 GB	294	0	247
	2000	8 GB	490	0	791
	2000	16 GB	701	0	1215

Table 1: Simulation results comparing the baseline and SpotLB versions.

B. Proof of Concept on AWS and Google Cloud

As described above, our experiments focused on testing the performance of two different clusters in four different scenarios:

1. Across the same Availability Zone in AWS
2. Across two different Availability Zones in AWS
3. Across two different regions in AWS
4. Across two different cloud providers (AWS and Google Cloud)

For each scenario, we ran two different tests:

1. Non-overload-handoff scenario – The cluster load balancer overload capabilities are disabled, so it acts as a standard load balancer
2. Overload-handoff scenario – The cluster load balancer capabilities are enabled, demonstrating our scenario

During each test, two clusters of four servers were started and allowed to run to determine the steady-state response time with ~60 requests/second send to the mock DNS load balancer. Once steady-state response was reached, one of the servers from one of the clusters was manually disabled to simulate a spot instance preemption.

The mock-DNS load balancer continued to divide load between the two clusters, and so the cluster with the "preempted" server was forced to handle the load with only three servers instead of the original four servers. For "non-overload handoff" tests, requests continued to be split just between the three servers in that cluster, while in the overload-handoff tests the cluster load balancer was able to identify overload scenarios and send overload requests to the other cluster to handle.

In each figure below, the horizontal axis represents time (in seconds) since the start of the test, and the vertical axis represents response times.

Scenario 1 – Across the same Availability Zone in AWS

Both clusters were in AWS region us-west-2 (Oregon, USA) in the same availability zone. All requests were made from Oregon.

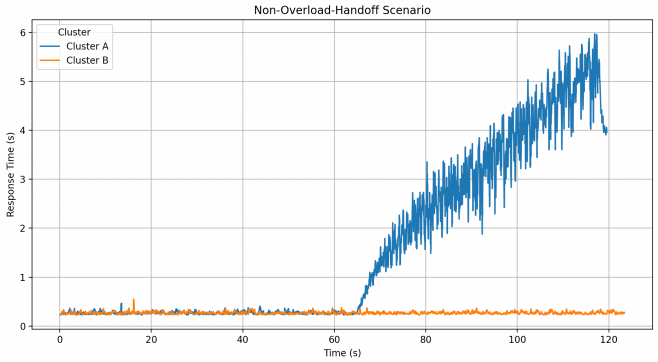

Figure 6: Scenario 1 response times for Non-Overload-Handoff

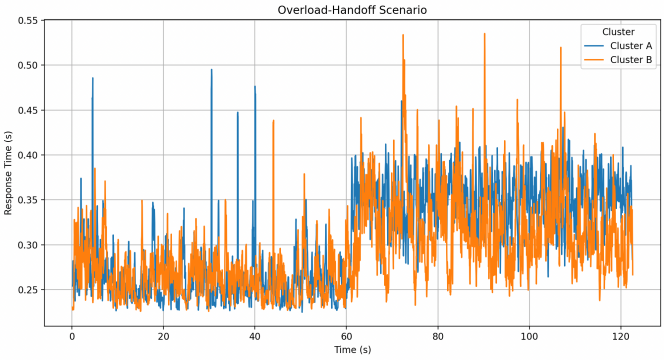

Figure 7: Scenario 1 response times for Overload-Handoff

Scenario 2 – Across two different Availability Zones in AWS

Both clusters were in AWS region us-west-2 (Oregon, USA) but in different availability zones. All requests were made from Virginia, USA.

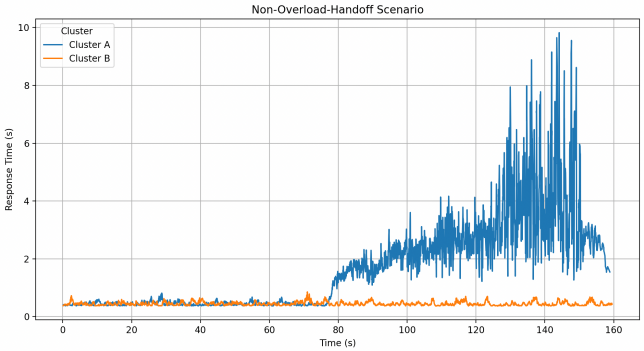

Figure 8: Scenario 2 response times for Non-Overload-Handoff

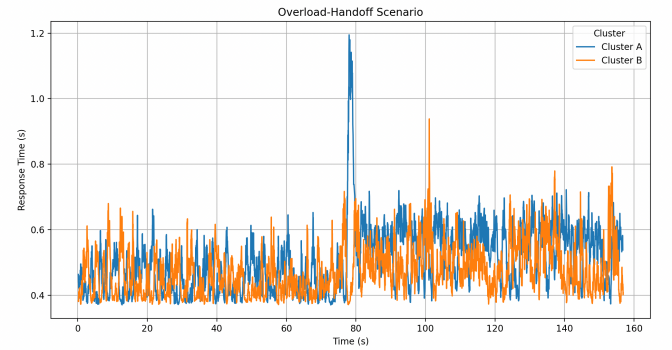

Figure 9: Scenario 2 response times for Overload-Handoff

Scenario 3 – Across two different regions in AWS

Cluster A and mock-DNS load balancer were in AWS region us-west-2 (Oregon, USA). Cluster B was in AWS region ap-southeast-2 (Sydney, Australia). All requests were made from Virginia, USA.

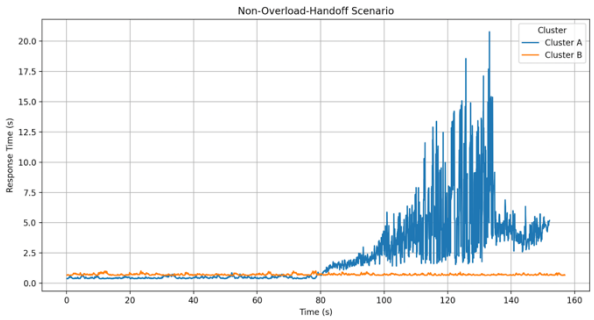

Figure 10: Scenario 3 response times for Non-Overload-Handoff

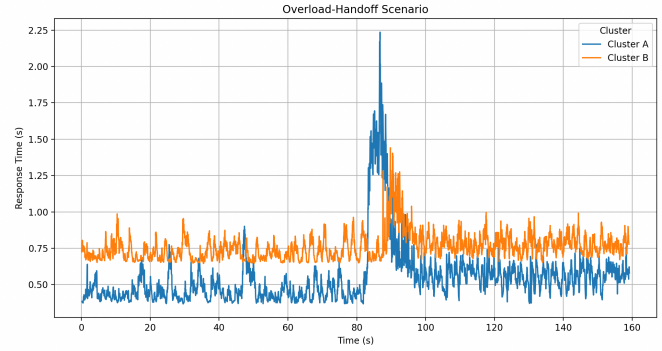

Figure 11: Scenario 3 response times for Overload-Handoff

Scenario 4 – Across two different cloud providers (AWS and Google Cloud)

Cluster A and mock-DNS load balancer were located in AWS region us-west-2 (Oregon, USA). Cluster B was located in Google Cloud region us-central1-a (Iowa, USA). All requests were made from Oregon.

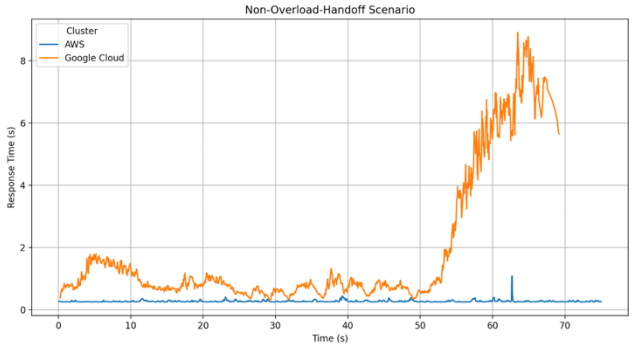

Figure 12: Scenario 4 response times for Non-Overload-Handoff

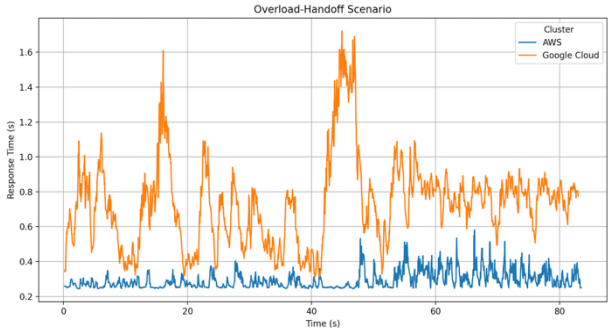

Figure 13: Scenario 4 response times for Overload-Handoff

VII. Analysis Of Results

A. CloudSim

The simulation results support our hypothesis that the SpotLB setup is better than the standard multi-cloud setup with no overload handover between load balancers. The number of successful requests in the SpotLB setup is, on average, 90% higher than the baseline setup while maintaining no violations with an SLO of 500 (CloudSim unit time). Some insights about the simulation results are:

1. Since the workload is memory intensive, increasing the RPS doesn't have a significant effect on the number of successful requests in both the baseline and SpotLB setups.
2. The number of requests handled successfully depends on the overall memory available across all the cloud providers. This is in line with point 1 above.
3. For any multi-cloud configuration, the number of requests transferred between load balancers increases proportionately with the RPS. This is caused by the instances being overutilized much earlier and each LB trying to pass the request assigned to it to the other LBs.
4. For a fixed RPS, if we keep increasing the memory of individual instances or implement an autoscaling policy, it will increase the number of successful in both the baseline and SpotLB versions and reduce the number of requests transferred in the SpotLB version.

Hence, our hypothesis about SpotLB offering better reliability still holds.

B. *Proof of Concept on AWS and Google Cloud*

The experimental results above demonstrated that the SpotLB overload management system kept each cluster from developing runaway response times, while the tests without SpotLB developed runaway response times in each scenario.

For example, in the non-SpotLB scenario shown in Figure 6, for the first ~65 seconds (before the simulated preemption), requests had a consistent response time of approximately 0.25 – 0.30 seconds. After the preemption, while the response times for cluster B remained similar, the response times for cluster A increased to ~5 seconds, demonstrating cluster overload and violating a reasonable SLO.

On the other hand, when Spot LB was implemented for the same scenario in Figure 7, response times averaged 0.25 – 0.30 seconds before the simulated preemption and after the preemption response times increased slightly (averaging 0.30 – 0.40 seconds across both clusters), but Cluster A never became overloaded because it was able to send overload requests to Cluster B.

The SpotLB solution was able to work even in scenarios where the clusters were geographically very far apart. For example, Figure 10 and 11 show a scenario where Cluster A was located in Oregon, USA, while Cluster B was located in Sydney, Australia, over 7,000 miles apart. As can be see in Figure 11, this distance caused response times from Cluster B to be ~0.25 seconds longer than from Cluster A on average. Despite this distance and the added response time, however, comparing Figure 10 to Figure 11 shows that Spot LB was able to deliver significantly better results in overload scenarios than a system which did not send overload requests to another cluster.

Finally, SpotLB even worked across multiple different cloud providers where the clusters and servers were running on hardware with different capabilities. For example, Figure 12 and 13 show a scenario where one cluster is located on AWS and the other cluster is located on Google Cloud. While Figure 13 shows that the average response time (and response time variability) is different between the two different clusters and sets of hardware, the response times with SpotLB in Figure 12 are significantly lower post-simulated preemption than the response times without SpotLB.

Despite these small differences between the different scenarios, the benefits of SpotLB held true across all scenarios, demonstrating Spot LB's ability to work in a variety of different configurations: across a single AZ, across multiple AZs, across multiple regions, and across multiple cloud providers.

VIII. Future Work

There are two key areas we see potential future enhancement in: further exploring improved cost efficiency through spot price bidding, and developing a more thorough simulation of the SpotLB model.

Currently, our spot price bidding strategy uses predictive bidding and diversification. However, enhancements can still be made. More efficient machine learning processes can improve the quality of spot price forecasts and enhance the optimization of bidding decisions. Further, by analyzing other possible bidding strategies such as proactive bidding and value-based bidding, additional savings could be obtained while maintaining availability.

To further refine predictive bidding, we suggest considering an ARIMA model blended with the simple moving average. For example, these models would let one identify seasonality or a period in a time series and conduct better forecasts by capturing all repeated patterns in spot prices.

Additionally, we propose developing a ladder pricing strategy for spot instance bidding. This would require both expected spot prices and their confidence intervals, then developing a normal distribution of different prices to bid for each instance based on the expected likelihood of each instance being preempted. This strategy could reduce the risk of all instances being preempted simultaneously because of price spikes but would require trade-off between the expected cost and probability of instance losses. Using this combination of approaches could significantly make the process of spot price more resilient.

Finally, the collection of spot pricing data from cloud providers like Azure or Alibaba for future analysis and its use in enhancing predictive bidding models could ensure more accurate forecasts. By leveraging this data, predictive bidding could adapt to real-time market trends while optimizing cost efficiency and reliability.

Under the topic of developing a more thorough simulation of the SpotLB model, we see potential enhancements in using more thorough DNS load balancer, developing more rigorous overload handling, and testing using true spot instance preemptions instead of simulated preemptions.

Apart from the geolocation and weighted routing options we explored, Route 53 also offers latency-based routing, failover routing, and multi-value answer routing that could further distribute traffic while increasing fault tolerance. Additionally, Route 53's traffic steering could route traffic to different endpoints depending on the user's location, device type, and context of conditions in networks.

Additionally, our overload handling developed for the cluster load balancers depends on basic threshold-based approaches. More sophisticated overload detection and mitigation mechanisms could be implemented to create better identification of overload situations, improving overload handling.

Finally, our proof-of-concept testing involved simulating spot instance preemptions by manually shutting down server instances. In this testing, we only preempted instances from a singular cluster under the assumption that preemptions would be independent in clusters spread across a variety of different regions and cloud providers. Testing SpotLB for a longer period of time with server shutdowns caused by true preemptions as opposed to simulated preemptions would verify

this assumption and demonstrate SpotLB's capabilities in a more robust and realistic scenario.

IX. Conclusion

Our research explored the feasibility of achieving reliability Service Level Objectives (SLOs) using only spot instances in a multi-cloud environment. Through the simulation of SpotLB and a proof-of-concept demonstration on AWS, we have shown that hierarchical load balancing combined with intelligent cluster-based overload handoff can effectively manage spot instances while minimizing the risk of failures due to instance terminations.

Our simulated results demonstrated that SpotLB reduces failed requests and maintains acceptable latency even under fluctuating resource conditions, supporting the potential for cost optimization in cloud environments. Additionally, our proof-of-concept demonstration on AWS and Google Cloud showed that when using a delayed centralized load balancer (as is the case with DNS load balancers due to caching) along with standard cluster load balancers, server preemption can cause server overloads. These overloads can result in response times well exceeding SLOs. However, by sending overload requests from one cluster load balancer to another, while using the same centralized load balancer and at the same request rates, response times can be kept within SLOs.

Finally, while cost-based request routing is yet to be implemented, future work could focus on integrating machine learning models to refine predictive bidding strategies. Leveraging real-world cloud provider data rather than heuristic trends can enhance accuracy, improving the ability to adapt to dynamic spot price fluctuations and optimize bidding decisions.

References

[1] Mell P, Grance T. The NIST definition of cloud computing. Gaithersburg: Nation- al Institute of Standards and Technology; 2011 https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-145.pdf

[2] H. A. Imran et al., "Multi-Cloud: A Comprehensive Review," 2020 IEEE 23rd International Multitopic Conference (INMIC), Bahawalpur, Pakistan, 2020, pp. 1-5, doi: 10.1109/INMIC50486.2020.9318176.

[3] https://www.terraform.io/

[4] Hong, J., Dreibholz, T., Schenkel, J.A., Hu, J.A. (2019). An Overview of Multi-cloud Computing. In: Barolli, L., Takizawa, M., Xhafa, F., Enokido, T. (eds) Web, Artificial Intelligence and Network Applications. WAINA 2019. Advances in Intelligent Systems and Computing, vol 927. Springer, Cham. https://doi.org/10.1007/978-3-030-15035-8_103

[5] Zhe Wu and Harsha V. Madhyastha. 2013. Understanding the latency benefits of multi-cloud webservice deployments. SIGCOMM Comput. Commun. Rev. 43, 2 (April 2013), 13–20. https://doi.org/10.1145/2479957.2479960

[6] https://info.flexera.com/CM-REPORT-State-of-the-Cloud

[7] V. Sharma, "Managing Multi-Cloud Deployments on Kubernetes with Istio, Prometheus and Grafana," 2022 8th International Conference on Advanced Computing and Communication Systems (ICACCS), Coimbatore, India, 2022, pp. 525-529, doi: 10.1109/ICACCS54159.2022.9785124.

[8] A Survey and Comparative Study on Multi-Cloud Architectures: Emerging Issues And Challenges For Cloud Federation. arXiv:2108.12831 [cs.DC]

[9] https://istio.io/latest/docs/setup/install/multicluster/

[10] Qiping She, Qing Li, and Juan Deng. 2016. The study of multi-instance purchase decision-making for minimising customers' cost under fluctuating cloud demands. Int. J. High Perform. Comput. Netw. 9, 1–2 (2016), 127–133. https://doi.org/10.1504/ijhpcn.2016.074665

[11] https://community.ibm.com/community/user/cloud/blogs/arindam-dasgupta/2024/09/18/aws-ec2-spot-ondemand-reserved-high-level

[12] JCS Kadupitige, Vikram Jadhao, and Prateek Sharma. 2020. Modeling The Temporally Constrained Preemptions of Transient Cloud VMs. In Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '20). Association for Computing Machinery, New York, NY, USA, 41–52. https://doi.org/10.1145/3369583.3392671

[13] Edirisinghe, Dasith & Rajapakse, Kavinda & Abeysinghe, Pasindu & Rathnayake, Sunimal. (2024). Cost-Optimal Microservices Deployment with Cluster Autoscaling and Spot Pricing. 10.48550/arXiv.2405.12311.

[14] Ahmed Ali-Eldin, Jonathan Westin, Bin Wang, Prateek Sharma, and Prashant Shenoy. 2019. SpotWeb: Running Latency-sensitive Distributed Web Services on Transient Cloud Servers. In Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '19). Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3307681.3325397

[15] Jose Pergentino Araujo Neto, Donald M. Pianto, Célia Ghedini Ralha, MULTS: A multi-cloud fault-tolerant architecture to manage transient servers in cloud computing, Journal of Systems Architecture, Volume 101, 2019, 101651, ISSN 1383-7621, https://doi.org/10.1016/j.sysarc.2019.101651.

[16] Qu C, Calheiros RN, Buyya R. Mitigating impact of short-term overload on multi-cloud web applications through geographical load balancing. Concurrency Computat: Pract. Exper. 2017; 29:e4126. https://doi.org/10.1002/cpe.4126

[17] Liu, W., Wang, P., Meng, Y. et al. Cloud spot instance price prediction using kNN regression. Hum. Cent. Comput. Inf. Sci. 10, 34 (2020). https://doi.org/10.1186/s13673-020-00239-5

[18] D. Kong, S. Liu and L. Pan, "Amazon Spot Instance Price Prediction with GRU Network," 2021 IEEE 24th International Conference on Computer Supported Cooperative Work in Design (CSCWD), Dalian, China, 2021, pp. 31-36, doi: 10.1109/CSCWD49262.2021.9437881.

[19] Nezamdoust, S.S., Pourmina, M.A. & Razzazi, F. Optimal prediction of cloud spot instance price utilizing deep learning. J Supercomput 79, 7626–7647 (2023). https://doi.org/10.1007/s11227-022-04970-x

[20] Liduo Lin, Li Pan, Shijun Liu, Methods for improving the availability of spot instances: A survey, Computers in Industry, Volume 141, 2022, 103718, ISSN 0166-3615, https://doi.org/10.1016/j.compind.2022.103718.

[21] Pawar, C.S., Ganatra, A., Nayak, A., Ramoliya, D., Patel, R. (2021). Use of Machine Learning Services in Cloud. In: Pandian, A., Fernando, X., Islam, S.M.S. (eds) Computer Networks, Big Data and IoT. Lecture Notes on Data Engineering and Communications Technologies, vol 66. Springer, Singapore. https://doi.org/10.1007/978-981-16-0965-7_5

[22] Moon, JB., Kim, MH. (2005). Dynamic Load Balancing Method Based on DNS for Distributed Web Systems. In: Bauknecht, K., Pröll, B., Werthner, H. (eds) E-Commerce and Web Technologies. EC-Web 2005. Lecture Notes in Computer Science, vol 3590. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11545163_24

[23] Gajbhiye, Amit & Singh, Shailendra. (2017). Global Server Load Balancing with Networked Load Balancers for Geographically Distributed Cloud Data-Centres. International Journal of Computer Science and Network. 6. 1-5.

[24] https://docs.aws.amazon.com/whitepapers/latest/real-time-communication-on-aws/cross-region-dns-based-load-balancing-and-failover.html

[25] https://learn.microsoft.com/en-us/azure/load-balancer/cross-region-overview

[26] https://cloud.google.com/blog/products/networking/how-cloud-load-balancing-supports-hybrid-and-multicloud

[27] H. Lin, L. Yang, H. Guo and J. Cao, "Decentralized Task Offloading in Edge Computing: An Offline-to-Online Reinforcement Learning

Approach," in IEEE Transactions on Computers, vol. 73, no. 6, pp. 1603-1615, June 2024, doi: 10.1109/TC.2024.3377912.

[28] Deepak Puthal, Rajiv Ranjan, Ashish Nanda, Priyadarsi Nanda, Prem Prakash Jayaraman, Albert Y. Zomaya, Secure authentication and load balancing of distributed edge datacenters, Journal of Parallel and Distributed Computing, Volume 124, 2019, Pages 60-69, ISSN 0743-7315, https://doi.org/10.1016/j.jpdc.2018.10.007.

[29] Yang Li, Bo Lei, Zhaojiang Li, Zheyan Qu, Xing Zhang, and Wenbo Wang. 2023. Task Offloading with Multi-cluster Collaboration for Computing and Network Convergence. In Proceedings of the 29th Annual International Conference on Mobile Computing and Networking (ACM MobiCom '23). Association for Computing Machinery, New York, NY, USA, Article 94, 1–3. https://doi.org/10.1145/3570361.3614064

[30] Tahmasebi-Pouya, Niloofar, Sarram, Mehdi-Agha, Mostafavi, Seyedakbar, A Blind Load-Balancing Algorithm (BLBA) for Distributing Tasks in Fog Nodes, Wireless Communications and Mobile Computing, 2022, 1533949, 11 pages, 2022. https://doi.org/10.1155/2022/1533949

[31] R. Beraldi and G. P. Mattia, "Power of Random Choices Made Efficient for Fog Computing," in IEEE Transactions on Cloud Computing, vol. 10, no. 2, pp. 1130-1141, 1 April-June 2022, doi: 10.1109/TCC.2020.2968443.

[32] A. Basiri, et al.,"Chaos Engineering" in IEEE Software, vol. 33, no. 03, pp.35-41, 2016.doi: 10.1109/MS.2016.60, https://doi.ieeecomputersociety.org/10.1109/MS.2016.60

[33] Chaos Engineering Upgraded, (Jul. 2020), https://netflixtechblog.com/chaos-engineering-upgraded-878d341f15fa

[34] Netflix Chaos Monkey Upgraded, (Jul. 2020): https://netflixtechblog.com/netflix-chaos-monkey-upgraded-1d679429be5d

[35] Hiller, M., Jhumka, A., Suri, N.: Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation. Springer (2003), https://doi.org/10.1007/b105828

[36] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter and V. Sekar, "Gremlin: Systematic Resilience Testing of Microservices," 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS), Nara, Japan, 2016, pp. 57-66, doi: 10.1109/ICDCS.2016.11.

[37] ChaosMesh: A chaos engineering platform for Kubernetes (2023). https://github.com/chaos-mesh/chaos-mesh

[38] Brady, James F., and Neil J. Gunther. "How to emulate web traffic using standard load testing tools." arXiv preprint arXiv:1607.05356 (2016).

[39] https://groups.google.com/g/mechanical-sympathy/c/icNZJejUHfE/m/BfDekfBEs_sJ

[40] Apache JMeter: https://jmeter.apache.org/

[41] Locust: https://locust.io/

[42] Gatling: https://gatling.io/

[43] wrk2: https://github.com/giltene/wrk2

[44] Calheiros, R.N., Ranjan, R., Beloglazov, A., De Rose, C.A.F. and Buyya, R. (2011), CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Softw: Pract. Exper., 41: 23-50. https://doi.org/10.1002/spe.995

[45] https://github.com/Cloudslab/cloudsim

[46] https://gitlab.engr.illinois.edu/adeelas2/cs598-spotlb-on-cloudsim/-/blob/master/modules/cloudsim/src/main/java/org/cloudbus/cloudsim/EX/disk/SpotHddVm.java

[47] https://gitlab.engr.illinois.edu/adeelas2/cs598-spotlb-on-cloudsim/-/blob/master/modules/cloudsim/src/main/java/org/cloudbus/cloudsim/web/WeightedRoundRobinLoadBalancer.java

[48] https://gitlab.engr.illinois.edu/adeelas2/cs598-spotlb-on-cloudsim/-/blob/master/modules/cloudsim/src/main/java/org/cloudbus/cloudsim/web/workload/brokers/SpotLBEntryPoint.java

[49] https://gitlab.engr.illinois.edu/adeelas2/cs598-ccc-spotlb/-/blob/main/cluster-load-balancer/cluster-load-balancer.py

[50] https://gitlab.engr.illinois.edu/adeelas2/cs598-ccc-spotlb/-/blob/main/server/report.py

[51] https://gitlab.engr.illinois.edu/adeelas2/cs598-ccc-spotlb/-/blob/main/server/server.py

[52] https://gitlab.engr.illinois.edu/adeelas2/cs598-spotlb-on-cloudsim/-/blob/master/modules/cloudsim-examples/src/main/java/org/cloudbus/cloudsim/examples/CloudSimMultiCloudLB.java

Conference Sumission

Below are five conferences we're considering submitting to:

A. CCGrid 2025

Why we may have a shot at acceptance

Our paper could have a shot at acceptance because it combines two of the six tracks that CCGRID looks for papers in. Track 2, includes papers focused on "resource management and scheduling" (which our unique multi-tier hierarchical load balancing system as well as spot instance cost savings fits under), and track 3, includes "use of ML/AI applications to enhance the performance of cluster, cloud and Internet-computing systems" which we are exploring for SpotLB pricing. With more refinement, our paper could propose significant cost savings across a wide variety of different cloud use cases compared to existing solutions, demonstrating the value that SpotLB provides.

Paper submission deadline

December 23, 2024

Paper submission requirements:

Paper should not exceed 10 letter size pages (8.5 x 11, IEEE format for conference proceedings), including figures, tables and references. Paper cannot be simultaneously sent to another conference for review. Submissions need to be double-blinded. Authors should not reveal their identity in their submissions

B. IEEE Cloud Summit 2025

Why we may have a shot at acceptance

Our paper aligns well with Track 3: Algorithm and Software Innovations. This includes "Advanced algorithms for cloud and fog computing" which our combination of a multi-tier hierarchical load balancing system and overload-handling cluster load balancers could be considered, as well as "Quality of service (QoS) and resource optimization", which our focus on maintaining SLOs could be considered. Additionally, if we expand on our spot pricing bidding algorithm, we could also meet this track's criteria for "Big data analytics and machine learning in cloud environments".

Paper submission deadline

March 1, 2025

Paper submission requirements

Up to 6 pages (up to 2 additional pages allowed with an extra fee). Paper cannot be simultaneously sent to another conference for review. Paper should follow the IEEE double column format. At least one author must register and present at the conference.

C. HPDC 2025

Why we may have a shot at acceptance

Our paper closely aligns with many of the topics that HPDC lists, including "datacenter, HPC, cloud, serverless, and edge/IoT computing platforms" - we're focused on load balancing across multiple different datacenters and clouds, "resource management and scheduling" - we're focused on reducing resource cost through the use of spot instances, and "fault tolerance, reliability, and availability" and "operational guarantees, risk assessment, and management" - our plan is to demonstrate that even using spot instances you can guarantee high reliability and availability. Given the close alignment of our paper with many of the topics this conference focuses on, and the limited existing research in our specific area, we think we have a shot at acceptance. Also, similar papers - most notably SpotWeb: Running Latency-sensitive Distributed Web Services on Transient Cloud Servers, have been previously accepted at HPDC under the High Performance Distributed Systems section.

Paper submission deadline

February 6, 2025

Paper submission requirements

Submissions need to be double-blinded. Authors should not reveal their identity in their submissions. At most 11 pages in PDF format, excluding references. Papers should be formatted in the ACM Proceedings Style. Submitted papers must be original work that has not appeared in and is not under consideration for another conference or a journal.

D. IEEE Cloud 2025

Why we may have a shot at acceptance

Our paper is well aligned with the call for papers of the IEEE CLOUD 2025 regarding several key topics, such as "multi-cloud and cloud federation" challenges. It directly addresses key challenges in cloud computing by emphasizing the exclusive use of spot instances to achieve SLOs in multi-cloud environments. The emphasis on leveraging only spot instances makes our approach both unique and impactful for the cloud computing community.

Paper submission deadline

March 3, 2025

Paper submission requirements

Paper Length: 7 to 10 pages for the main contents (including all text, footnotes, figures, tables and appendices) with additional pages for appropriate references. US Letter; Two-column format in the IEEE style. Double-blind reviewing process; author names and affiliations should not appear in the paper.

E. IC2E 2025

Why we may have a shot at acceptance

The IC2E conference is a little different than the conferences described above and seems to be more focused on the application of engineering ideas to cloud computing. Given our paper's focus on developing a system that allows the economical use of spot instances across multiple cloud providers, we think it could fit into this conference's intended topics. While the 2025 conference tracks haven't been published yet, in 2024 the tracks including "cloud management and engineering, from single (micro-)services to complete system landscapes" and "Non-technical aspects of Cloud Computing, e.g., Cloud governance or cloud economics" aligned closely with our paper (our paper being focused on developing a unique cloud landscape based on spot instances and focused on cloud economics for the end user). The more practical focus on this conference could be a good fit for our paper given our paper's focus on cost savings and SLOs.

Paper submission deadline

April 17, 2025 (Abstract), April 24, 2025 (Paper)

Paper submission requirements

Not yet listed for 2025, but from 2024 - Full papers (including figures and tables) should not exceed 9 double-column pages; references can be additional pages. Submitted papers must be original work that has not appeared in and is not under consideration for another conference or a journal. Authors are strongly encouraged to publish software and data sets as open-source/data.

F. Work Before Submission

Before submitting to any of the conferences listed above, to increase the strength of our submission and the likelihood of being accepted by the conference, there are a few areas (aligned with much of what we discussed in the future work section) that we plan to explore further. In particular, we would want to further explore the improved cost efficiency through spot price bidding, and develop a more thorough simulation of the SpotLB model.

For the spot price bidding, we would collect data around the active price of spot instances at cloud providers that support spot price bidding (e.g., Azure and Alibaba) and use this data to develop a ARIMA model which would help identify any time series in the model. We would combine this with the idea of developing a ladder pricing strategy for spot instance bidding, where each spot instance is bid at a different price so that as instance prices increase we're more likely to lose one spot instance at a time as opposed to losing all spot instances simultaneously.

Additionally, to better prove the concept of SpotLB, we'd want to run testing using the developed predictive model across multiple clouds, running the tests for long enough that instances are shut down due to true spot instance preemption as opposed to our simulated preemptions. This would allow us to demonstrate that our model can handle real-world preemption

scenarios and frequency as opposed to the conceptual single preemption situations that our current solution demonstrates.

The additional work on spot price bidding would help to demonstrate the novelty of our solution, while the real world testing of SpotLB across multiple clouds with real preemptions would demonstrate the robustness of our solution. Combined together, these two enhancements would increase the strength of our paper against other conference submissions.