Composing Microservices and Serverless for Load Resilience

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

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

Antonios Katsarakis
Huawei Research
Edinburgh, UK
antoniskatsarakis@yahoo.com

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

Rakesh Kumar
NTNU
Trondheim, Norway
rakesh.kumar@ntnu.no

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

ABSTRACT

Online services strive to maintain application responsiveness even when the traffic is unpredictable and fluctuating. Today's online services are commonly deployed as graphs of microservices, each microservice packaged as one or more containers inside a virtual machines (VMs). While performant and affordable when the load is steady, VM-based deployments are known to be slow to scale when the load spikes, resulting in degraded performance for end-users of the service. To avoid such performance degradations, service providers can over-provision their deployments; however, such a strategy is costly and inefficient, leaving resources heavily under-utilized for extended periods of time.

To address this challenge, we propose Hydra, a hybrid architecture that combines microservices with serverless computing. Hydra utilizes VMs to handle steady workloads cost-effectively and leverages serverless elasticity to absorb traffic spikes. When compared to an all-VM deployment with Kubernetes auto-scaling, Hydra achieves a 62.4% reduction in peak tail latency with a minimal 2.3% increase in cost.

1 INTRODUCTION

Today's online services are complex, tiered applications composed of multiple functionalities that must deliver a cost-effective and seamless end-user experience on a tight latency budget. Examples of such services include social networks, online stores and media portals. Due to the need for high scalability, availability and developer productivity, online services are typically developed and deployed as *microservices* – a collection of lightweight independent services that communicate via RPC.

Prior works have shown that it is common for online services to experience load fluctuation, both at regular intervals (e.g., higher load during a day and lower load at night-time) as well as less-predictable episodic spikes [27, 28, 5]. The latter may arise due to a major news event, an online flash sale, or the release of a suddenly popular digital media item. Regular load fluctuations are straight-forward to accommodate by provisioning capacity for the expected load ahead of time. In contrast, irregular changes in load may present a challenge, particularly if they are sudden and if the amplitude of the spike is large.

Maintaining a highly responsive service in the face of a load spike is a well-known challenge [35, 27, 1]. First, a spike must be detected and confirmed to be non-transient, and then additional resources must be provisioned and brought online before traffic can be redirected to them. In practice, these steps may take minutes or even tens of minutes. For instance, Unity [41], a service provider that hosts several e-stores that sometimes feature flash sales, has an online forum where users frequently complain about their inability to access a given store whenever a flash sale is in progress. In one such discussion thread, titled "Store Server Overloaded Resulting in Missed Flash Sales Purchases", a user writes: "In 12 minutes since the sale started the page has only pulled up once." [37].

To avoid the slow scale-out problem, services can be over-provisioned by deploying more instances than required for a given load level. However, our analysis of a week-long trace from Twitter [5] shows that the load spikes by over 2x on several occasions during the week. Having enough stand-by capacity to absorb such spikes would be prohibitively expensive, since the extra resources would have to be deployed and paid for continuously, even when they are not needed.

In theory, the bulk of an online service can be deployed using serverless, which is known to be highly scalable both in terms of time to launch an instance (typically just seconds or less) and the number of concurrent instances (hundreds or even thousands)[1]. In practice, though, the cost of running even a moderately popular online service on serverless

[1]Only services that are inherently stateless can be ported to serverless. Fortunately, these comprise the bulk of existing online services based on our analysis of DeathStartBench workloads [18].

would be prohibitive. Our calculations show that a representative hour-long steady-load fragment of the Twitter trace would cost 3.6x more to serve using serverless than VM-based microservices.

In this work, we observe that online services are best served by microservices when the load is steady or predictable, thus providing good performance in a cost-effective manner. However, we argue that effectively tolerating load spikes is fundamentally challenging in existing microservice architectures due to an inherent cost-performance trade-off. Overprovisioning resources to handle the worst-case load is costly, while provisioning new resources on the fly to handle a spike incurs a high latency overhead.

In response, we propose Hydra, a hybrid architecture combining microservices and serverless. Under normal load, Hydra runs online applications using microservices, exactly the same as it is done in today's deployments. When the load spikes, Hydra engages a serverless component, rapidly shifting as much load as necessary to absorb the spike while new microservice instances are launched in the background. Once new microservice instances are available, the load is shifted onto them. Our results show that Hydra achieves the best of both serverless and microservice designs – high scalability, responsiveness *and* cost-efficiency.

2 MOTIVATION

2.1 Modern Online Services

Modern online services, extensively utilized by users, have evolved into complex systems. Given their interaction with users, they encounter varied and unpredictable usage patterns [5]. Nevertheless, meeting Service Level Objectives (SLO) and adhering to latency constraints is crucial, as any deviation significantly compromises the user experience [25]. To address these challenges, large service providers such as Airbnb, Netflix, LinkedIn, Uber, and Twitter have adopted a distributed microservice architecture [14, 38, 34, 40, 21].

In the microservices architecture, an application is composed of small, loosely coupled components , each responsible for handling specific isolated functions. These microservices are deployed as containers in virtual machines (VMs), allowing each component to scale independently. As a design principle, microservices are typically intended to be stateless in nature [22, 18], facilitating greater scalability.

2.2 Varying Load Patterns

The load, or request arrival rate, of a service can exhibit significant variability, influenced by factors such as the time of day, day of the week, or season [5]. This is evident in several online services, as demonstrated by Twitter, a social networking platform facing higher loads during the day and lower loads at night, or a shopping website encountering increased demand during the holiday season.

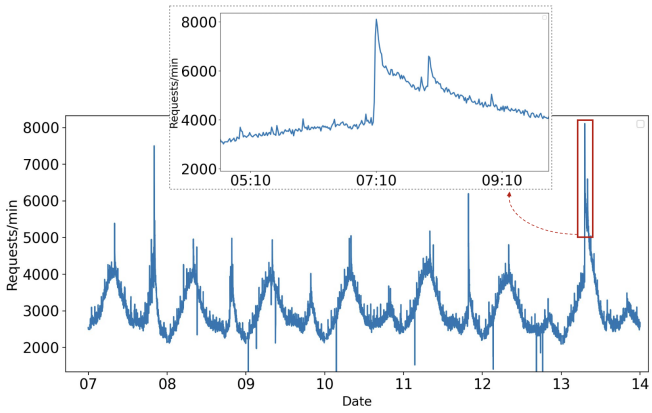

Figure 1: The load trace of Twitter over a week's timespan. The highlighted section represents a few hours from a specific day.

Figure 1 illustrates the load pattern of Twitter [5] over a week-long period. We observe a mostly consistent traffic pattern characterized by predictable periodic trends with minor fluctuations based on the time of day. However, the figure also shows multiple instances of load spiking at various points in the week without a clear periodic trend. The zoomed-in portion of Figure 1 shows one such spike, starting from 07:10 when the load suddenly increases from just under 4000 requests/min to 8000 requests/min – a doubling in load. This emphasizes that microservice systems must be capable of gracefully handling unexpected spikes without compromising service quality. The pivotal question is: Can existing microservice deployments and their autoscaling mechanisms effectively contend with such load spikes?

2.3 Microservice Deployment Strategies and Autoscaling

Microservices are commonly deployed as containers on top of a cluster of virtual machines (VMs) with the help of container orchestration platforms such as Kubernetes (K8s). [32, 2, 30]. Kubernetes streamlines the management of microservices by offering features like automated deployment, scaling, and monitoring. In the Kubernetes framework, the fundamental unit of deployment and scaling is a *pod*, which typically comprises a primary container and, if necessary, additional helper containers. One or more pods are deployed within a single VM on a physical node.

At the initial scaling level, operations are conducted at the pod granularity, and the Horizontal Pod Autoscaler (HPA) plays a pivotal role in this process. In the scaling-out process,

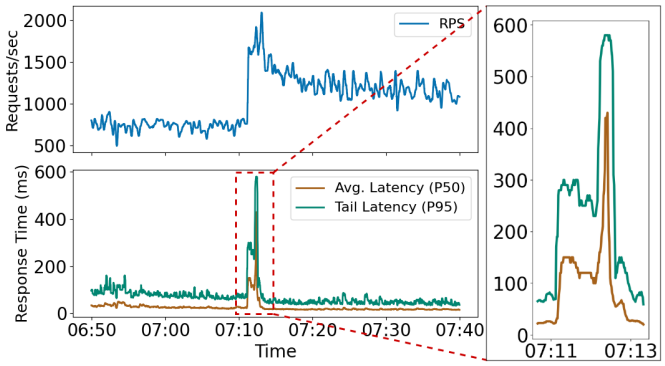

Figure 2: Impact of a sudden spike in load on a microservices-based application.

the HPA orchestrates the creation of new pods within a running VM based on observed metrics, such as CPU utilization or metrics like requests per second (RPS).

In scenarios where the existing cluster of VMs lacks the necessary capacity to accommodate new pods, the Cluster Autoscaler (CA) serves as the second level of autoscaling. The CA dynamically provisions new VMs and seamlessly integrates them into the cluster, thereby enabling HPA to place additional pods inside the newly-created VMs.

Problematically, cluster scaling operations are not without their limitations. For instance, HPA incurs a detection lag, which is 15 seconds by default and cannot be further reduced in current production Kubernetes clusters, such as Amazon Elastic Kubernetes Service (EKS) [6] and Google Kubernetes Engine (GKE) [19].

On top of the extra latency for detection, initiating new pods usually requires multiple seconds, while provisioning new VMs can extend to minutes [4]. In short, in case of a major load spike, when existing VMs are not sufficient to absorb the load, new VMs must be provisioned and the autoscaling process can take minutes to complete, during which time, the service quality will inevitably suffer.

2.4 Microservices Meet Load Spikes

Handling sudden load spikes poses a significant challenge for current microservice autoscaling systems. The difficulty lies in the time it takes for the autoscaler to recognize that a bona-fide spike is in progress *(detection lag)* and the subsequent time needed to scale-out instances *(reaction lag)*. The combined detection and reaction lag can lead significant request queuing within existing instances, adversely affecting system performance and user experience. In essence, autoscalers struggle to swiftly adapt to sharp spikes in workload, resulting in performance bottlenecks and delays during periods of higher demand.

We now evaluate the impact of the load spikes on the end-to-end latency, defined as the total time taken from the moment a client's request is sent to when the response is received. This evaluation is conducted by replaying a one-hour-long load trace from the day highlighted in Figure 1 on an application deployed on Kubernetes in an AWS EKS cluster. The application comprises two microservices in a sequential chain. The initial service handles client requests, subsequently invoking the second service. Once responses are obtained from the second service, the first service forwards them back to the client. Our infrastructure integrates both Horizontal Pod Autoscaler (HPA) and Cluster Autoscaler (CA), representing state-of-the-art production-grade autoscaling. Section 5 provides detailed information on the parameters used in this study.

In the portion of the trace outside of the spike (i.e., before 7:10 and after 7:14), we observe minimal load fluctuation, resulting in consistently low and stable latency. Starting from 07:11, there is a notable surge in the arrival rate of requests, causing a significant rise in response time. As shown in Figure 2, the average latency increases by 14x and the tail latency by 8.3x.

As discussed earlier, slow scaling can overload the system, resulting in high latency for end-users and violations of Service Level Agreements (SLAs). In attempts to address this issue, service providers often over-provision clusters.[10] The load spike, illustrated in Figure 1, is identified as being twice the regular load. Therefore, even with a marginal over-provisioning strategy aimed at handling a 2x spike, there is a potential for a 66% increase in costs during an hour long period from the Twitter trace (6:40 - 7:40), as illustrated in Figure 7. Such over-provisioning contributes to low resource efficiency and elevated costs [42].

2.5 Serverless to the Rescue?

While traditional microservices architectures struggle with handling sudden spikes in load, serverless computing offers a promising alternative for tackling this challenge. Within the serverless model, the application's functionality is divided into event-driven tasks, which are lightweight and stateless, referred to as functions. Being lightweight and stateless allows near-instantaneous scaling of these functions from zero to thousands of active instances [24] on the order of seconds, compared to the minutes it takes to launch VM instances. The fact that many services within a typical microservice framework are stateless by design, presents an opportunity to deploy them as serverless functions without the need to rewrite the application. Major cloud service providers, including AWS Lambda [36], GCF [12], and Azure Functions [9], offer support for serverless deployments. In addition to rapid scaling and fast instance launch, serverless computing

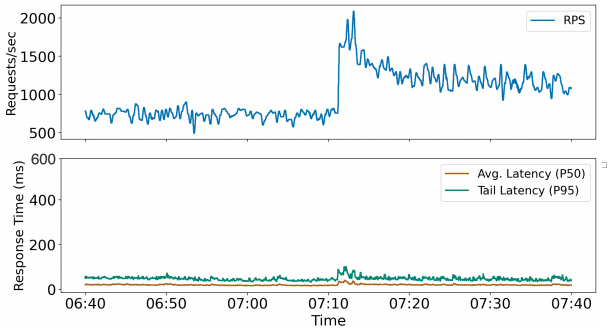

Figure 3: Impact of a sudden spike in load on a serverless-function based application.

also features a pay-as-you-go model that eliminates costs for idle resources. It's seamless scalability position serverless computing as a compelling solution for applications with highly variable load patterns.

We study the effect of the load spike highlighted in Figure 1 on the simple two-service benchmark described in 2.4 deployed using serverless functions. Results of the study are shown in Figure 3. Unlike the microservice scenario depicted in Figure 2, where the load spike resulted in a significant increase in tail and average latency (8.3x and 14x respectively), the serverless functions show only a slight uptick in both average and tail as low as 1.9x. This result highlights the resilience of serverless functions in reacting to sudden spikes in load.

Serverless functions may not be optimal for scenarios involving foreseeable and steady loads due to their higher cost compared to provisioning resources in advance[13, 23]. For instance, the estimated cost of running a single service from our toy application for an hour-long steady-load segment of the Twitter trace (from 05:10 to 06:10) is 3.6x higher on serverless as compared to running on VMs. The calculation is based on AWS Lambda and AWS EC2 pricing [7, 8], where serverless function memory is configured to match the vCPUs of the VM instance. [31]. The higher cost of running on serverless is because serverless functions are charged based on the number of requests and the time taken to execute each request. Hence, for a predictable and consistent load, it is more economical to use dedicated VMs. In contrast, serverless best accommodates sporadic workloads that can scale up and down based on demand [13].

3 DESIGN

We use two insights to address the limitations of existing scalable deployment approaches. First, we capitalize on the elasticity and ultra-fast startup time of serverless functions to efficiently manage load spikes. Second, we harness the cost-effective nature of virtual machines to handle steady workloads reliably. Leveraging both insights, we introduce Hydra, a hybrid system that strategically combines VMs and serverless functions to provide predictable performance and cost efficiency when the load is stable, with rapid scaling when a load spikes.

3.1 System in Action

Figure 4 provides a high-level depiction of Hydra in action. Incoming service requests, under typical load, are directed to existing pods in VMs by the load balancer (step 1). The Hydra controller detects load spikes by monitoring RPS and CPU metrics (step 2). Upon identifying a spike, the Hydra controller instructs the load balancer to shift a fraction of the traffic to serverless functions (step 3). Consequently, the load balancer routes the excess traffic portion to serverless functions, alleviating the load on VMs (step 4). Simultaneously, CA brings up new VMs to accommodate the increased load. Once the new VMs are operational, the Hydra controller instructs the load balancer to shift all the traffic back to the VMs (step 5).

Next, we describe the Hydra Controller, a novel component that comprises a monitoring module and is responsible for executing the entire load balancing logic.

3.2 Hydra Controller

We introduce a lightweight component called the *Hydra Controller* that runs as an application within the cluster. This controller is responsible for monitoring CPU and RPS metrics and making decisions accordingly. Hydra's monitoring system operates at a fine granularity of 1-second intervals, ensuring close to instantaneous detection of any surge in demand. The controller is also in charge of distributing the traffic between VMs and serverless instances whenever serverless is engaged or disengaged. This controller is fully automated, operating in the background, continously monitoring the load and making decisions accordingly.

Hydra employs a weighted load balancing approach[17, 29] to effectively distribute traffic between VMs and serverless components. Under steady load conditions, traffic is directed solely towards VM clusters, with serverless clusters receiving no traffic. In the event of a load spike, Hydra dynamically calculates weights in real-time for both VM and serverless clusters based on observed metrics. During weight calculation, Hydra ensures that the weight assigned to the serverless cluster corresponds only to the excess traffic generated by the load spike. Once the CA has added new VMs to the cluster and they are operational, the Hydra controller gradually updates the load balancer weights to redirect traffic back to the VMs.

Key Takeaway: The Hydra controller, serving as the sole novel component, runs independently within the K8s cluster.

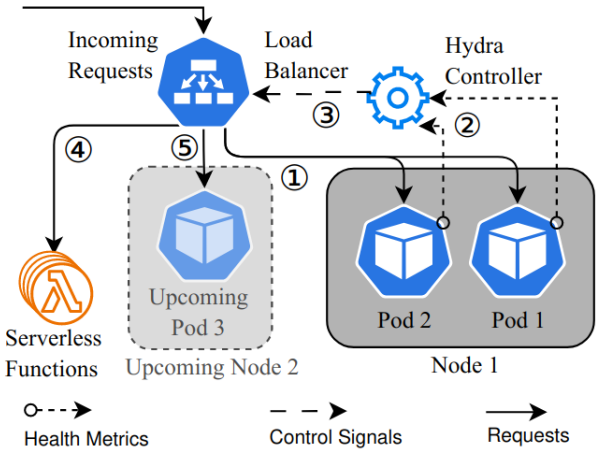

Figure 4: Hydra design overview

Hydra controller integrates into current K8s deployments without necessitating any modifications. Additionally, both data and control paths remain unaltered for both serverless and VM implementations.

4 IMPLEMENTATION

We implement Hydra using Knative enabled K8s cluster. Knative extends K8s to provide a framework for deploying and managing serverless functions. [20]

Fine-Grained Monitoring: To detect the load spikes, K8s uses CAdvisor [11] to monitor the CPU usage of each pod. To enable real-time load spike detection, we extend the CAdvisor source to collect CPU usage statistics from each pod every 1 second rather than the older minimum of 15 seconds and trigger the redirection of requests to the serverless functions. These metrics are consumed by Prometheus [33], which is a widely used monitoring system and is exposed to the controller. Prometheus also collects service-level metrics like RPS and latency and exposes them to the controller. The Hydra controller uses these metrics to calculate the weights within the load-balancing algorithm.

Dynamic Load Shifting: For dynamic redirection of requests, we use Istio [39]. Istio is a service mesh that offers a way to control how microservices interact with each other without changing the microservices themselves. Istio supports dynamic redirection by injecting a sidecar container named Envoy proxy[16], alongside each microservice and configuring those proxies to control and track network traffic between services. Outgoing requests from a microservice are intercepted by the Envoy proxy and redirected to the appropriate destination. Each service has 2 possible destinations: the containerized version of the service and the serverless version of the service. The ratio of requests sent to each destination is determined by the load-balancing logic described under Section 3.2.

5 EXPERIMENTAL METHODOLOGY

To evaluate cost and performance trade-offs, we compare Hydra against a baseline using the following experiment setup.

Benchmarks: We create a custom benchmark consisting of 2 microservices Caller and Callee. Upon receiving a request, the Load Balancer invokes the Caller service. The Caller service then invokes Callee service and waits for the response. Once the response from Callee is received, Caller returns too. The Callee service is a CPU-intensive service written in *C++* that performs floating-point operations. The Caller service written in *golang* just acts as a proxy to invoke Callee service. Both the services are stateless and hence can be deployed as either Knative services or K8s services.

Configurations: We compare Hydra against the K8s default baseline (*Baseline*), that utilizes an EKS cluster with default parameters for HPA and CA. The HPA is configured to trigger scaling when the average CPU utilization reaches a threshold of 50% [28]. Additionally, the average Requests Per Second (RPS) threshold is configured to correspond to the RPS at 50% CPU utilization. The HPA has a default metrics polling interval of 15 seconds (*–horizontal-pod-autoscaler-sync-period*), while the CA checks the need for new VMs at intervals of 10 seconds(*-scan-interval*).

Cluster: The K8s cluster is deployed using Amazon EKS [3] and the nodes are EC2 instances. The nodes are of type *t3.xlarge* with 4 vCPU cores and 16 GB of RAM. The cost of the cluster is 0.1670 USD per hour per VM [8]. For Knative deployments, dedicated VMs are allocated within the same K8s cluster. Since the cold start times of the production serverless offerings like AWS Lambda, Azure Functions, is 10 times lower than that of Knative [15], we keep Knative functions warm to replicate the production serverless offerings.

Load trace: We use the Twitter trace [5] to generate the load. Since, the original trace is sampled, we scale up the trace by 7x for our evaluation to generate sufficient load to trigger CA-level auto-scaling. We use locust [26] to generate the load. Locust is deployed in a VM with 4 vCPU cores and 16 GB of RAM in the same AWS region as the cluster.

Cost Estimate: For the serverless costs, we calculate the total run-time of the Knative functions and use the AWS cost calculator [7] to estimate the cost for AWS Lambda functions with the same memory and CPU configuration for a similar execution time. For example, if a Knative function runs for 1 second with 128 MB of memory and has 1000 invocations, we estimate the cost of the AWS Lambda function with 128 MB of memory for 1000 seconds.

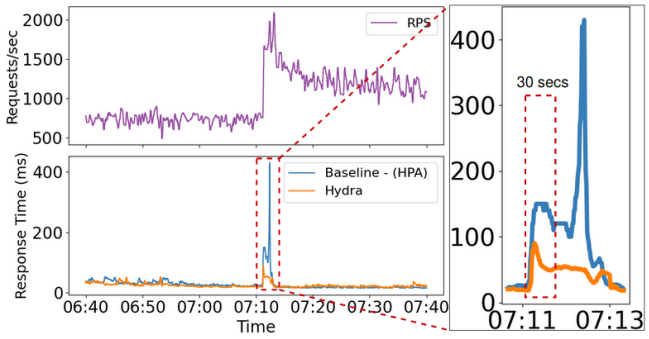

Figure 5: Average end-to-end latency comparison

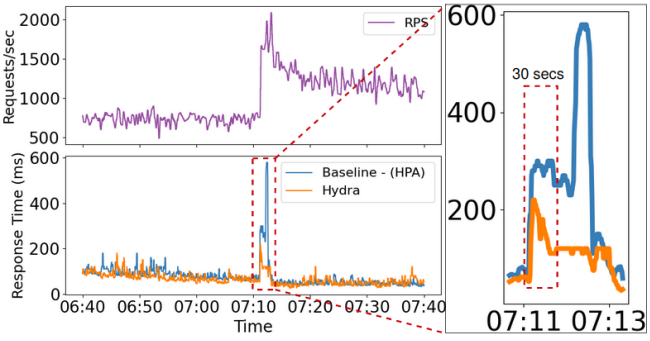

Figure 6: Tail end-to-end latency comparison (95th percentile)

6 EVALUATION

6.1 Performance

We first evaluate Hydra's ability to compensate for increased tail latencies due to sudden load spikes. We compare the average and tail (95%) latencies of Hydra against the baseline. Results are depicted in Figures 5 and 6. Before a spike in the trace (before 7:10), the baseline and Hydra exhibit similar latencies.

In comparison to the Kubernetes default baseline, Hydra showcases substantial latency improvements, reducing peak tail latency by 62.4% and peak average latency by 78.5%.

Additionally, it is noteworthy that Hydra quickly absorbs the load spike, with the latency spike beginning to decrease in less than 30 seconds. In contrast, the baseline takes around 2 minutes to absorb the load. This indicates that Hydra can quickly absorb load spikes with minimal performance impact.

6.2 Cost

Next, we compare the cost of Hydra against the baseline and another configuration—an over-provisioned VM-based cluster (*over-provisioned*). The over-provisioned cluster is configured to absorb the peak load in the studied trace with an average CPU utilization of 50% or less. As the observed peak load is approximately double the stable load, the over-provisioned configuration has twice the resources of the baseline cluster.

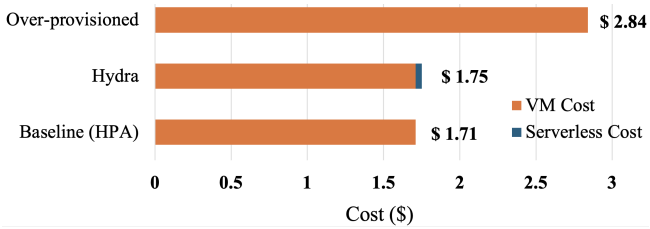

Figure 7: Cost comparison

We illustrate the cost comparison in Figure 7. Throughout an hour-long trace, the baseline incurs a cost of $1.71, the over-provisioned configuration costs $2.84, and Hydra has a cost of $1.75 in total. The cost of Hydra is 2.3% higher than the baseline and 38.3% lower than the over-provisioned configuration. This indicates that Hydra can effectively absorb load spikes with minimal cost impact.

7 CONCLUSION

Dealing with sudden spikes in load poses a considerable challenge for existing VM-based microservices due to the time required to bring up additional resources. While serverless functions with ultra-fast startup times offer a promising solution, the high cost of a serverless-only approach hinders widespread adoption. To address this issue, we propose Hydra, a hybrid architecture that seamlessly combines microservices with serverless computing to improve scalability and load resilience. Our evaluation shows that Hydra significantly reduces peak tail latency by 62.4% compared to the default Kubernetes baseline, with only a minimal 2.3% increase in cost. This highlights Hydra's effectiveness in providing a streamlined solution for achieving load resilience cost-efficiently within modern online service architectures.

REFERENCES

[1] 2022. "Fascinating facts about facades at CBS Sports". [Online; accessed 25. Jan. 2024]. (Dec. 2022). https://www.gomomento.com/blog/fascinating-facts-about-facades-at-cbs-sports.

[2] 2023. "Swarm mode overview". [Online; accessed 11. Jan. 2024]. (Dec. 2023). https://docs.docker.com/engine/swarm.

[3] 2024. Amazon EKS Customers | Managed Kubernetes Service | Amazon Web Services. [Online; accessed 25. Jan. 2024]. (Jan. 2024). https://aws.amazon.com/eks.

[4] 2024. AMI types - Amazon Elastic Compute Cloud. [Online; accessed 9. Jan. 2024]. (Jan. 2024). https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ComponentsAMIs.html#storage-for-the-root-device.

[5] 2024. Archive Team: The Twitter Stream Grab. [Online; accessed 9. Jan. 2024]. (Jan. 2024). https://archive.org/details/twitterstream.

[6] 2024. AWS EKS Horizontal Pod Autoscaler Sync Interval. [Online; accessed 9. Jan. 2024]. (Jan. 2024). https://github.com/aws/containers-roadmap/issues/1809.

[7] 2024. AWS Lambda pricing calculator. [Online; accessed 27. Jan. 2024]. (Jan. 2024). https://calculator.aws/#/createCalculator/Lambda.

[8] 2024. AWS VM Instances cost. [Online; accessed 27. Jan. 2024]. (Jan. 2024). https://aws.amazon.com/ec2/instance-types/t3/.

[9] 2024. Azure Functions – Serverless Functions in Computing | Microsoft Azure. [Online; accessed 11. Jan. 2024]. (Jan. 2024). https://azure.microsoft.com/en-gb/products/functions#overview.

[10] 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* (SoCC '19). Association for Computing Machinery, Santa Cruz, CA, USA, 126–138. ISBN: 9781450369732. DOI: 10.1145/3357223.3362706.

[11] 2024. cadvisor. [Online; accessed 24. Jan. 2024]. (Jan. 2024). https://github.com/google/cadvisor.

[12] 2024. Cloud Functions | Google Cloud. [Online; accessed 11. Jan. 2024]. (Jan. 2024). https://cloud.google.com/functions.

[13] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. 2021. Sebs: a serverless benchmark suite for function-as-a-service computing. (2021). arXiv: 2012.14132 [cs.DC].

[14] TC Currie. [n. d.] Airbnb's 10 Takeaways from Moving to Microservices — thenewstack.io. https://thenewstack.io/airbnbs-10-takeaways-moving-microservices/. [Accessed 08-01-2024]. ().

[15] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. 2020. Xanadu: mitigating cascading cold starts in serverless function chain deployments. In *Proceedings of the 21st International Middleware Conference* (Middleware '20). Association for Computing Machinery, Delft, Netherlands, 356–370. ISBN: 9781450381536. DOI: 10.1145/3423211.3425690.

[16] 2024. Envoy proxy - home. [Online; accessed 2. Feb. 2024]. (Feb. 2024). https://www.envoyproxy.io.

[17] 2024. Envoy Proxy | Load Balancing. [Online; accessed 27. Jan. 2024]. (Jan. 2024). https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/upstream/load_balancing/load_balancers.

[18] Yu Gan et al. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '19). Association for Computing Machinery, Providence, RI, USA, 3–18. ISBN: 9781450362405. DOI: 10.1145/3297858.3304013.

[19] 2024. Google Kubernetes Engine Horizontal Pod Autoscaler Sync Interval. [Online; accessed 10. Jan. 2024]. (Jan. 2024). https://cloud.google.com/kubernetes-engine/docs/concepts/horizontalpodautoscaler.

[20] 2024. Home - Knative. [Online; accessed 24. Jan. 2024]. (Jan. 2024). https://knative.dev/docs.

[21] Jenny Qiu Hylbert and Steve Cosenza. 12 August 2020. Rebuilding twitter's public api. (12 August 2020). https://blog.twitter.com/engineering/en_us/topics/infrastructure/2020/rebuild_twitter_public_api_2020.

[22] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '21). Association for Computing Machinery, Virtual, USA, 152–166. ISBN: 9781450383172. DOI: 10.1145/3445814.3446701.

[23] Eric Jonas et al. 2019. Cloud programming simplified: a berkeley view on serverless computing. (2019). arXiv: 1902.03383 [cs.OS].

[24] 2024. Lambda function scaling - AWS Lambda. [Online; accessed 11. Jan. 2024]. (Jan. 2024). https://docs.aws.amazon.com/lambda/latest/dg/lambda-concurrency.html.

[25] Jianshu Liu, Qingyang Wang, Shungeng Zhang, Liting Hu, and Dilma Da Silva. 2023. Sora: a latency sensitive approach for microservice soft resource adaptation. In *Proceedings of the 24th International Middleware Conference* (Middleware '23). Association for Computing Machinery, <conf-loc>, <city>Bologna</city>, <country>Italy</country>, </conf-loc>, 43–56. ISBN: 9798400701771. DOI: 10.1145/3590140.3592851.

[26] 2024. Locust.io. [Online; accessed 27. Jan. 2024]. (Jan. 2024). https://locust.io.

[27] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. 2021. Characterizing microservice dependency and performance: alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing* (SoCC '21). Association for Computing Machinery, Seattle, WA, USA, 412–426. ISBN: 9781450386388. DOI: 10.1145/3472883.3487003.

[28] 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* (SoCC '22). Association for Computing Machinery, San Francisco, California, 355–369. ISBN: 9781450394147. DOI: 10.1145/3542929.3563477.

[29] 2024. NGINX Docs | NGINX Load Balancing. [Online; accessed 27. Jan. 2024]. (Jan. 2024). https://nginx.org/en/docs/http/load_balancing.html#nginx_weighted_load_balancing.

[30] 2024. Nomad | HashiCorp Developer. [Online; accessed 11. Jan. 2024]. (Jan. 2024). https://developer.hashicorp.com/nomad.

[31] 2024. Optimizing Lambda Cost with Multi-Threading. [Online; accessed 27. Jan. 2024]. (Jan. 2024). https://web.archive.org/web/20220629183438/https://www.sentiatechblog.com/aws-re-invent-2020-day-3-optimizing-lambda-cost-with-multi-threading?utm_source=reddit&utm_medium=social&utm_campaign=day3_lambda.

[32] 2024. Production-Grade Container Orchestration. [Online; accessed 9. Jan. 2024]. (Jan. 2024). https://kubernetes.io.

[33] Prometheus. 2024. Prometheus - Monitoring system & time series database. [Online; accessed 25. Jan. 2024]. (Jan. 2024). https://prometheus.io.

[34] [n. d.] Q&A with Jim Brikman: Splitting Up a Codebase into Microservices and Artifacts. en. (). Retrieved Jan. 8, 2024 from https://engineering.linkedin.com/blog/2016/02/q-a-with-jim-brikman--splitting-up-a-codebase-into-microservices.

[35] 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, (Nov. 2020), 805–825. ISBN: 978-1-939133-19-9. https://www.usenix.org/conference/osdi20/presentation/qiu.

[36] 2024. Serverless Function, FaaS Serverless - AWS Lambda - AWS. [Online; accessed 11. Jan. 2024]. (Jan. 2024). https://aws.amazon.com/lambda.

[37] 2024. Store Server Overloaded Resulting in Missed Flash Sales Purchases. [Online; accessed 9. Jan. 2024]. (Jan. 2024). https://forum.unity.com/threads/store-server-overloaded-resulting-in-missed-flash-sales-purchases.1265966.

[38] Web Team. 2023. Microservices at netflix: lessons for architectural design. (Jan. 2023). https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/.

[39] 2024. The Istio service mesh. [Online; accessed 24. Jan. 2024]. (Jan. 2024). https://istio.io/latest/about/service-mesh.

[40] 2016. The Opportunities Microservices Provide at Uber Engineering. [Online; accessed 8. Jan. 2024]. (Apr. 2016). https://www.uber.com/en-GB/blog/building-tincup-microservice-implementation.

[41] 2024. Unity | Asset Store. [Online; accessed 30. Jan. 2024]. (Jan. 2024). https://assetstore.unity.com/.

[42] Chris Zaloumis. 2022. Are Your Data Centers Keeping You From Sustainability? - IBM Blog. *IBM Blog*, (June 2022). https://www.ibm.com/blog/are-your-data-centers-keeping-you-from-sustainability.