Blurring the Boundary Between Serverless and Containerized Deployments For Interactive Webapps

Mondo Jiang
mondo.jiang@wisc.edu

Shivam Hire
hire@wisc.edu

Thinh Nguyen
tdnguyen25@wisc.edu

***Abstract*—This study explores the integration of containerized and serverless deployment strategies to optimize the scalability and cost-efficiency of interactive web applications. The report evaluates Kubernetes-based container deployments and Knative-powered serverless deployments using the DeathStarBench HotelReservation application. Experiments were conducted on a CloudLab cluster to assess performance under varying workloads. Results reveal that serverless deployments demonstrate higher latency compared to containers. Conversely, container deployments exhibit lower latency but lack dynamic scalability. The findings highlight the potential for a hybrid approach that combines the rapid scalability of serverless functions with the cost efficiency and performance stability of containers. Future work should address the challenges of hybrid deployment and focus on cost analyses to validate the feasibility of such solutions.**

I. Introduction

Web services have to handle unpredictable changes in demand. The deployment infrastructure have to scale out dynamically as demand fluctuates to keep latencies within acceptable limit.

Traditionally, web services were deployed using VMs [1] and as containers. To scale out we have to setup a static policy like scale up if CPU > 80% and scale down if CPU < 40%. These kind[2], [2] of policies can cause inefficient resource usage. VMs have a considerable startup time hence we have to keepthem on stand-by just in case we get sudden spike in demand. This leads to wastage of resources and money.

In recent years, web apps have also be deployed on serverless offerings since they can scale out instantly and provides fine-grained flexibility. These offerings can also handle bursty traffic more efficiently. However, when compared to VMs, these offerings are more costly leading to increase in operational cost.

We will be exploring how we can combines the two system to get best of the both worlds. We will use VMs for handling normal traffic. But as soon as we see more traffic coming in we will send it serverless functions instead of spinning up new VMs. This delay in scaling out the VM can obviate the need for scaling out (in the case of bursty traffic), or hide the delay of spinning up a VM (without requiring the provider to pre-warm the VM instance).

II. Related Work

Serverless Platform Comparision The paper by Li et al. [3] focuses on the architectural blocks that influence the performance of Kubernetes-based open-source serverless platforms, including Knative, Kubeless, Nuclio and OpenFaaS[4]. It offers a deeper understanding of design considerations and their impact on performance and auto-scaling. Kaviani et al.[5] explore the commonality and differences between several serverless computing platforms' interfaces, emphasizing the need for a unified interface to prevent vendor lock-in. They use Knative as a baseline and discuss the challenges and possible directions for serverless platforms to reach commodity status.

Containers and serverless Many previous work has focused on how to use both containers and serverless technologies together to decrease latency and cost while improving scalability. In fact many use containers to implement the serverless paradigm like SOCK[6]. SOCK proposes to use lightweight isolation primitives to aleviate functions sandboxing bootlenecks and Zygote-provisioning with 3-tier caching to improve starup latency. However, they all focus on reducing latency and cost by reducing startup time of serverless functions. As per our knowledge none of them focuses on how to intelligently distribute requests between containers and serverless to take advantage of both worlds. Approaches like SCAR[7] provides a framework which gives an impression of running containers on top of serverless platform. They tried to extend elastic scaling of these platform to other non-traditionally applications like High Throughput Computing applications. However they failed to provide a solution which can achieve scaling while minimizing cost.

Knative Tran et al.[8] states the problem with the default auto-scaler in Knative being suboptimal. Knative uses Knative Horizontal Pod Autoscaler (K-HPA) in which it monitors user defined parameters and the load of each services. However, ill-defined values leads to problems such as over-provisioning. So this work provides a solution that builds an optimal profile of each service depending on what it observes. They accomplish this without modifying the code base of Knative via extensions or custom resources. For auto-scaling, there are two schemes. Vertical scaling in which the resource allocation of existing instances are increases, and whereas Horizontal scaling increases the number of instances of a service. This paper introduces a hybrid approach that uses reinforcement learning to implement the hybrid approach. Their results show the hybrid approach achieved better scaling in terms of resources allocation and service latency compared to other methods. This relates to our work in the fact that we are using horizontal scaling when the VMs are overloaded in booting up severless function instances to handle the additional traffic.

III. Background

A. Kubernetes

Kubernetes is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications. Born out of the need to manage complex, distributed systems efficiently, Kubernetes has emerged as the de facto standard for container orchestration due to its robustness, flexibility, and extensibility.

In the context of our project, Kubernetes serves as the foundation for deploying and managing our applications. Its auto-scaling capabilities are particularly relevant as we aim to maintain performance during unpredictable demand fluctuations while optimizing resource utilization and cost.

a) *Core Concepts:*

Kubernetes has several key concepts that define its architecture and functionality:

Pods The smallest deployable units that can be created, scheduled, and managed. A pod typically contains one or more containers that are treated as a single entity.

Nodes and Masters Nodes are worker machines in Kubernetes that run the pods. Masters, on the other hand, manage the cluster state and the scheduling of pods across nodes.

Services An abstraction that defines a logical set of Pods and a policy by which to access them, often used to expose an application running on a set of Pods as a network service.

Deployments Manage the declaration and update of application instances. They provide a way to describe the desired state of an application, including the number of replicas, and automatically handle the deployment and scaling of application instances.

Namespaces Provide a way to divide cluster resources between multiple users (where each user may represent a team or project).

These core concepts will be integral to our methodology and will be referenced in later sections of our report as we delve into the specifics of our experiments and findings.

b) *Auto-scaling in Kubernetes:*

Kubernetes support 3 types of autoscaling as given below:

- The Horizontal Pod Autoscaler (HPA) automatically adjusts the number of pod replicas in a deployment, replica set, or stateful set based on observed resource metrics such as CPU, memory, or custom metrics. It ensures the application can handle varying workloads by scaling out (adding pods) during high demand and scaling in (removing pods) during low demand. HPA is ideal for applications with fluctuating traffic, such as web apps with periodic spikes or background jobs with varying loads.
- The Vertical Pod Autoscaler (VPA) automatically adjusts CPU and memory resource requests and limits for pods to ensure efficient resource utilization. By increasing or decreasing resource allocations, VPA helps avoid over-provisioning (wasting resources) or under-provisioning (causing instability). It's particularly useful for applications with predictable resource needs or workloads where the exact resource requirements are unknown, and can complement HPA by ensuring individual pods are right-sized.
- The Cluster Autoscaler dynamically adjusts the number of nodes in a Kubernetes cluster based on pod scheduling needs. It scales up the cluster by adding nodes when pods cannot be scheduled due to insufficient resources, and scales down by removing underutilized nodes to save costs. This is ideal for cloud-based clusters where workloads require burst capacity or temporary scaling to handle large jobs, ensuring the cluster can grow or shrink with workload demands.

We primarily study HPA, but we don't actively implement it in the final experiments.

B. Serverless Platforms

a) *Knative:*

Knative is a platform for Kubernetes that streamlines the configuring process for serverless applications. It couples serverless computing with automatic scaling and event-driven architecture. For example, a core component is the Knative Serving in which it supports auto-scaling, traffic splitting, scaling to zero, and revision management. Auto-scaling adjusts in the number of pods depending on the traffic. This is supported with scaling to zero in which when the application is not used, it releases all the resources. Another core component is the Knative Eventing in which it provides a mechanism for a loosely decoupled process of the consumption and production of events. As such Knative is popular for web applications, micro-services, and event-driven applications given these core components.

Knative is composed of three main components:

Knative Serving This component manages the deployment and scaling of serverless containers. It allows developers to deploy applications without worrying about the underlying infrastructure, as Knative automatically handles the scaling from zero to hundreds or thousands of instances based on demand.

Knative Eventing This component deals with event consumption and production. It provides a loosely decoupled system for event-driven applications, enabling the asynchronous communication between services.

Knative Networking This component, which includes Kourier, manages the networking layer for Knative, providing features like routing and access control.

We mainly use Knative Serving for our experiments.

b) *OpenFaaS[4]:*

When we first began, we were looking into OpenFaaS[4] for deploying serverless functions of the hotel application. First as an experimental phase, we only used the standalone runtime of OpenFaaS[4], called faasd. This allows us to mess around with OpenFaaS[4] without having to integrate it into our running Kubernetes cluster, which takes time for configuration. With the faasd engine running, which hosts the serverless functions that can be invoked via a curl request, we start to build some functions. Building the functions entails converting each micro-service into a container. However, we stumbled upon some difficulties in this process. For example, turning each micro-service into a container required a script that intercepts the incoming url request and redirects it to a function call. This script lives inside the container and would be replicated for each container of each micro-service. This is as far as we got because we stumbled upon the fact that Hotel Reservation already had a Knative deployment. As previously mentioned, Knative is another platform for running serverless micro-services. This meant the developers of Hotel Reservation already ported these micro-services into serverless forms, so we did not have to rebuild the wheel.

IV. Design and Methodology

A. Cluster Setup

In this section, we aim to provide a comprehensive overview of the setup and configuration of our Kubernetes clusters, which serve as the foundation for our experiments. We start with local cluster and a private cluster running in Mondo's personal pc. then switch to cloudlab for a universal testing environment.

a) *Local Cluster:*

To gain hands-on experience with Kubernetes, we initiated our project by setting up local clusters on our development machines. We utilized kind[2], which facilitates the creation of local Kubernetes clusters using Docker container nodes. This approach allowed each team member to have a personal, safe environment to experiment with Kubernetes configurations and applications without the need for a shared, remote setup.

b) *Private Cluster:*

For a more integrated development experience, we established a private cluster on a Windows 11 machine with WSL (Windows Subsystem for Linux) running Ubuntu 24.04. This private setup provided us with a shared working environment, leveraging an 8-core AMD 5800X processor and a 16GB memory limit. We employed kind[2] to create the base cluster[1] and installed essential tools such as ingress-nginx for managing external access to the services within our cluster and k8s-dashboard for visual cluster management.

To ensure seamless accessibility of our cluster services, we set up a dynamic DNS (DDNS) container. This component updates DNS records to point to the kind[2] cluster container and the ingress entrypoint, allowing us to access our services via a domain name owned by us rather than IP addresses.

To foster collaboration, we configured a container optimized for SSH access, enabling our team members to connect to the cluster securely. This setup leveraged Docker's internal network to call the Kubernetes API, make it safer to not export cluster directly to the public network.

c) *CloudLab:*

To ensure a stable and standardized testing environment, we opted for CloudLab, which provided us with three 32-core c220g1 machines running Ubuntu 22. This choice was pivotal as it offered a more reliable platform compared to our previous workstations, which suffered from performance instabilities due to multiple layers of virtualization.[2]

We employed the latest version of Kubespray (v2.26.0) to initialize our Kubernetes cluster on CloudLab. However, we encountered several challenges that required us to deviate from the standard setup process:

Removing EphemeralNode Flag We had to remove the EphemeralNode flag from the Kubernetes Feature Gate List, as it was not compatible with the newer version of kubespray and kubernetes.

Manual Script Adjustments After logging into node-0, we manually corrected the CloudLab Kubespray script located at /local/repository/setup-kubespray.sh. Specifically, we removed the redundant line layer3: for metallb, which was causing Kubespray to panic.

Rerunning Ansible Playbook After the necessary adjustments, we reran the Kubespray Ansible playbook and the setup-kubespray.sh script, which ultimately led to a successfully functioning cluster.

Enabling MetalLB for Load Balancer Support Given that Knative's default network layer, Kourier, requires collaboration with a load balancer service type, we chose to enable MetalLB. MetalLB is the only solution that provides load balancer support between nodes in a bare-metal setup like ours, ensuring that our cluster could handle external traffic effectively.

d) *Setup Procudure:*

Our codebase, which is crucial for reproducing our experiments, is hosted on GitHub at **uwm-cs736/project-public.**

For reproducible experiments, we set up our cluster using Terraform and Helm configurations, along with automated testing scripts. The process to create the initial state for the cluster is detailed below:

Preparation: Kubeconfig File Generation:

To connect to the Kubernetes cluster from our local machines, we first need to generate a kubeconfig file. This file contains the necessary credentials and configurations to interact with the cluster.

We have a script named export-kube-config.sh located in the cluster/scripts directory that facilitates this process. This script should be executed on the CloudLab node, which already has access to the Kubernetes API thanks to Kubespray.

Once the kubeconfig file is generated, we need to download this file to local machine, then set up our local KUBECONFIG environment variable to point to this new file, allowing us to interact with the cluster from our local development environment.

Initial State with Terraform:

For establishing the initial state of our cluster, we use Terraform by running terraform apply in the cluster folder (you can also run ./ctf a in the project root folder as a shortcut).

This command installs the Knative operator, Knative serving for HTTP/web workloads, the Knative network layer (Kourier), along with monitoring tools like Prometheus and Grafana, and the Kubernetes dashboard, including their respective service configurations for access.

[1]kind cluster config: https://github.com/uwm-cs736/project-public/blob/main/cluster/cluster-config.yml

[2]see our CloudLab experiment parameters here.

Long-Living Containers for Debugging and Testing:

To assist with debugging and running workload tests, we have installed long-living containers such as wrk2 and nettool (for network diagnostics).

These containers are instrumental in ensuring that our applications perform as expected under various conditions and help us identify and resolve any network-related issues within our cluster. wrk2 also serves as our main entrypoint for generating workloads.

B. *Metric Collection*

In our project, we focused on collecting detailed metrics to evaluate the Kubernetes-based deployments. This section outlines our approach to metric collection, which was conducted exclusively on CloudLab.

a) *Tools:*

We utilized the combination of Prometheus and Grafana for our metric collection and visualization needs. Prometheus is a powerful open-source monitoring system that can scrape and store metrics directly from monitored targets or instruments, while Grafana provides a rich set of visualization tools to analyze and display these metrics.

Combined together, they can save all the metric histories from our experiments, allowing us to analyze them even after the experiments are finished.

b) *Data Collection Methodology:*

Direct Collection from Kubernetes Core Components To ensure accuracy and timeliness, we configured Prometheus to collect data directly from the core components of our Kubernetes cluster. This approach allowed us to monitor key performance indicators such as CPU usage, memory consumption, network I/O, and more.

Port-Forwarding for Web Panel Access To access the web panels of Prometheus and Grafana, we employed port-forwarding. This technique allowed us to securely expose the web interfaces of these tools on our local machines, enabling us to interact with the dashboards without exposing them publicly.

It's important to note that while OpenFaaS[4] is more popular in the community and offer more user friendly features, its integration with Prometheus is not free. This was a significant factor in our decision to abandon OpenFaaS[4].

C. *Application Deployment*

a) *Container Form:*

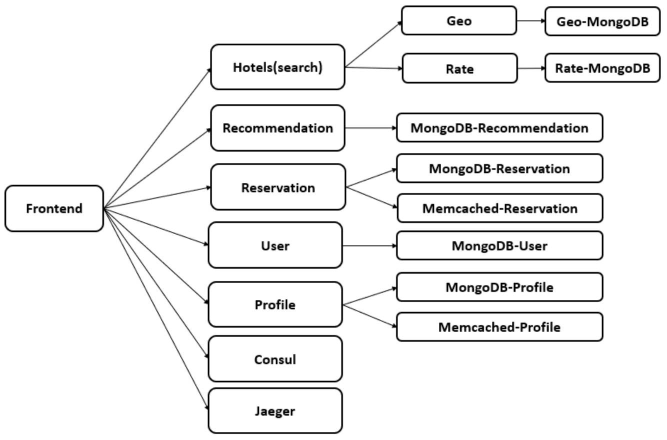

Fig. 1: The architecture for HotelReservation application. The frontend is the only HTTP entry point to all the gRPC services. The arrows indicate dependencies.

We adopted the Kubernetes configurations from the DeathStarBench[9] HotelReservation application and made slight changes to them. Since the original configuration did not work correctly, we had to modify the command for each service container to ensure the application runs properly.

The architecture for HotelReservation is shown in Fig. 1.

b) *Serverless Form:*

For running workloads in serverless form, we use the Knative deployment provided in Hotel Application. The DeathStarBench[9] repository provided the set of yamls file to deploy. First we started up a Knative cluster and loaded in the yaml files that were provided. These files deploys the microservices of Hotel Application as a set of serverless forms. After deploying these forms, we deploy the wrk (which is a benchmark provided by DeathStarBench[9] repository) container as a Pod in Kubernetes. Afterwards, we enter the container via exec and run the wrk executable while providing it with the endpoint of the frontend that was deployed earlier along with the microservices. Here, the frontend is what users interact with when using the Hotel Application. With the wrk container generating the traffic into frontend, the frontend routes them to the requested micro-services.

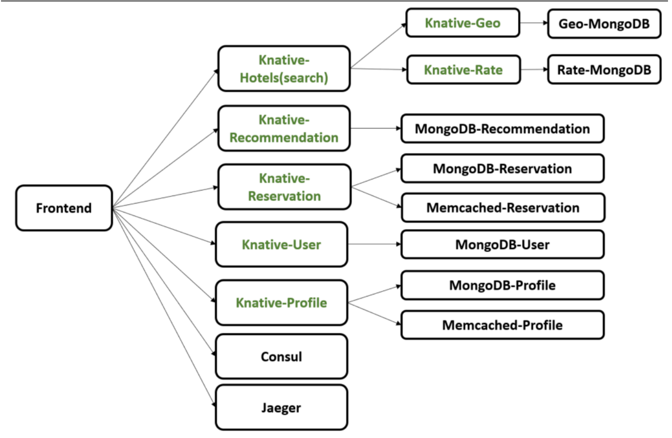

Fig. 2: The architecture for HotelReservation application in Knative format. The frontend remain normal k8s container form application. Services marked Green are refactored from k8s services to Knative services.

As a comparision to contianer form, the serverless form of HotelReservation is shown in Fig. 2

D. Workload Generation

We want to analyze how the 99th percentile latency changes as number of request/sec increases.

We provision only one pod and see how latency is distributed using CDF graph. Additionally, we also see how 99th percentile latency changes as we increase the request rate. This will give us an idea of how many request a node can handle. To generate workload we use wrk2 utility. We use following commands:

```
wrk -D exp -t 1 -c 1 -d 30 -L -s /mixed-
workload_type_1.lua -R [request-per-
second] <frontend-service-host>
```

The lua script is hosted in our repository as well.[3]

V. Experiments

We conduct our experiments on the CloudLab cluster as described in Section IV.A.c. We use wrk2 to generate the workload. Since the frontend service remains the same deployment in both forms of the application, we can use the same testing script for both applications.

For each form, we first restore the cluster to its original state by removing the previous test's configuration entities or by recreating the cluster using terraform destroy and terraform create. Then, we run our automation test preparation script[4] and wait for the containers to be ready.

After all application components are ready (for the Knative form, this means Knative serving successfully creates the deployment set for each service, even if scaled to zero meaning no pods are running for the service), we run the workload generation script, using different requests per second (rps) settings between 10 to 1000 (specifically, rps = 10, 100, 500, 1000). Wrk2 will give us the latency distribution, and by logging timestamps, we can accurately retrieve metrics from Prometheus for the timeframe during which experiments are running.

For all the service containers, we requested for 100m (0.1 Core) and set a maximum limit of 1000m (1 Core). We enabled hard currency auto scaling rule (containerConcurrency=20[5]) for Knative form deployments. We disabled HPA for container form.

A. Container Form Result

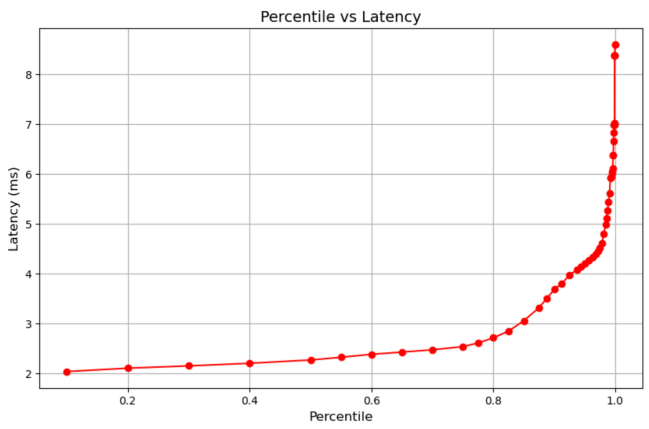

Fig. 3: Tail Latency Distributiuon (rps=100)

In Fig. 3, it is showing the distribution of the latency and it appears that the 99.99% is a great outlier from the rest. However, the latency is miliseconds is rather fast when compared to the proceeding charts in which we scale up the rps.

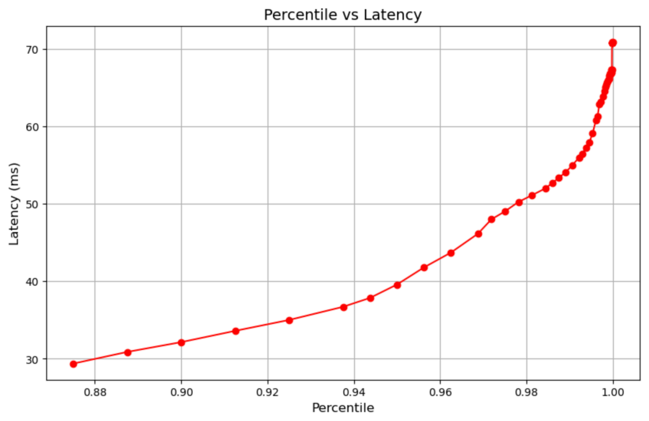

Fig. 4: Tail Latency Distributiuon (rps=500)

Fig. 4 Shows the latency of container Hotel Reservation application. There appears to be a rather steady increase in latency until the range between 60ms to 70ms in which it is explosive; however, it is less explosive than in Fig. 3. Most likely because the higher rps saturates the average latency so the effect of an explosion is less. For example, here the latency starts at around 30ms, whereas the latency starts around 2ms

[3]https://github.com/uwm-cs736/DeathStarBench/blob/master/hotelReservation/wrk2/scripts/hotel-reservation/mixed-workload_type_1.lua

[4]https://github.com/uwm-cs736/project-public/blob/main/expr/1-container/, start.sh is for preparation, stop.sh is for tearing down the test, and test.sh is the actual workload.

[5]https://knative.dev/docs/serving/autoscaling/concurrency/#hard-limit

in Fig. 3. The far range stems from when the containers start to get overloaded on the work.

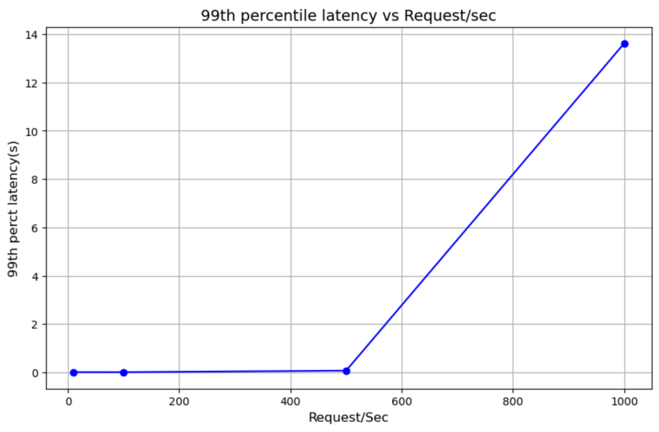

Fig. 5: Latency explosion for container form of the web application.

Here shows the sudden jump in latency because the auto-scaler is not implemented so there is a clear inflection point in which the workload starts to overwhelm the container (approximately 500 rps). Information such as this could be used to implement the decision making process of when to scale out, i.e. finding the inflection points. However, with more data points instead of 500rps and 1000rps, it might not be a straight line as shown here due to a finer-grained detail.

B. *Serverless Form Results*

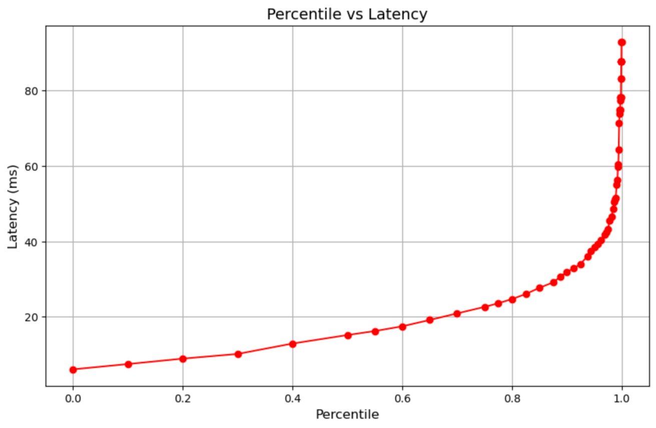

Fig. 6: Tail Latency Distribution (rps=100)

Compare both forms when rps is 100, for Knative shown in Fig. 6, the latency is 10 times slower (from 8ms to 80ms) it is compared to the container variant as shown in Fig. 3.

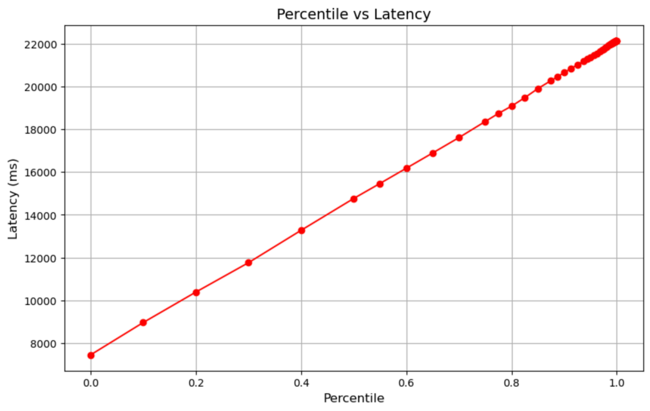

Fig. 7: Tail Latency Distribution (rps=500)

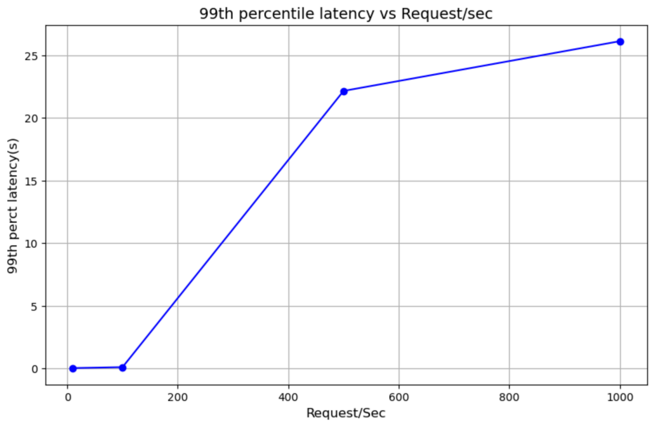

Fig. 8: Latency compared to requests per second

After we increase the rps to 500, the application saturates as shown in Fig. 7 and Fig. 8, there is a significant increase in latency, similar to what we observed with the RPS 1000 setting in the container form. This also indicates that our auto-scaling configuration for Knative should be adjusted to suit our future tests.

C. *Cluster Metrics*

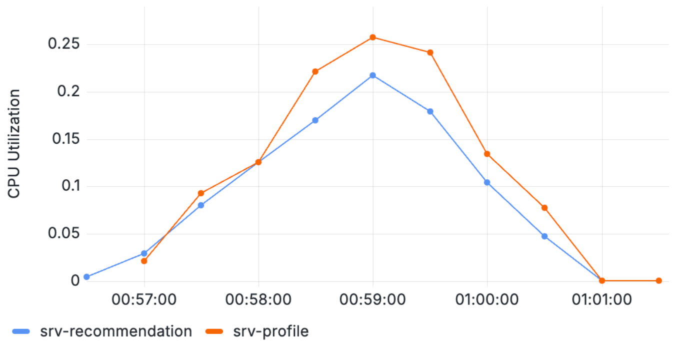

Fig. 9: Shows CPU utilization while the front-end is being tested with a mix workload for serverless recommendation and profile micro-services.

In Fig. 9, the time range begins when we start the workload and ends shortly after that. The workload consists of 1 thread, 1 connection, for 30 seconds with an increasing (10, 100, 500, 1000) request/sec. Here, the CPU gradually increases as the workload progresses and then diminishes as the workload ends. The workload ran for approximately two minutes. The form factor of the web application is serverless within this time range in the x-axis.

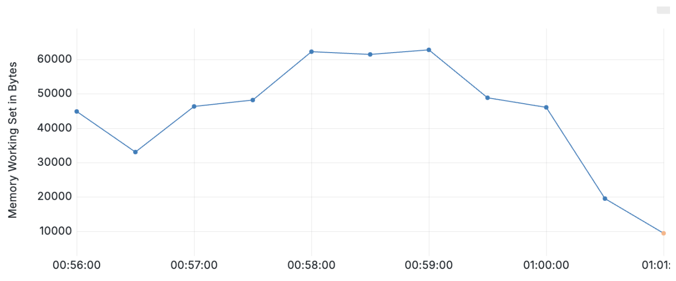

Fig. 10: Shows memory working set of wrk pod (the workload appliance to the serverless form services.

Fig. 10 Shows the bytes in terms of the working set on wrk, which is the program we used to apply a workload to the web application. During this time range, the kubernetes cluster was measuring the performance of serverless workload.

D. *Knative Scale to Zero*

One interesting effect to highlight is the scale-to-zero feature of Knative. During our multiple tests, you can clearly see from Fig. 11 that after our test ends, the pods are automatically eliminated by Knative. This is not possible with traditional Kubernetes applications as shown in Fig. 12

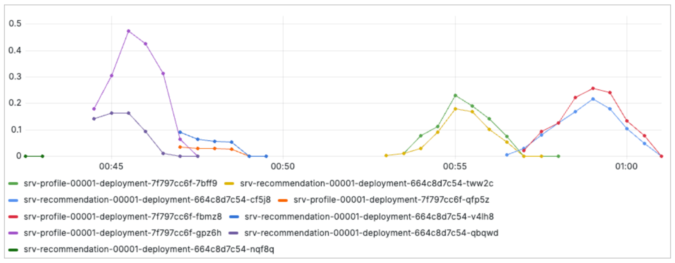

Fig. 11: CPU usage of containers across multiple Knative experiments runs. There's no change in application deployment configuration. All src-* containers are automatically scaled up and down by Knative Serving. Our workload generation script only hits the recommendation and profile services.

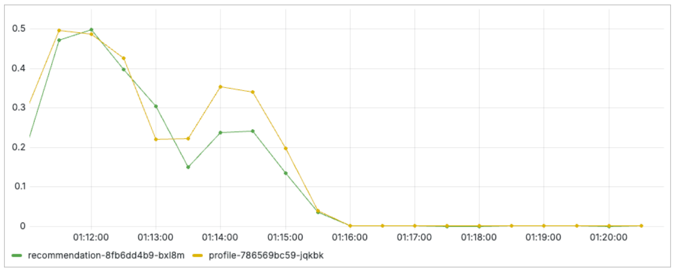

Fig. 12: CPU usage of container form applications. After the experiments, the CPU usage almost reaches zero, but they still exist in the cluster if we don't destroy them.

VI. Conclusion & Disccusion

We tested both forms of the Hotel Reservation application from DeathStarBench[9] on a CloudLab cluster. The results demonstrate that the serverless form of the application is ten times slower than the container form. Our results indicate the need for future research on the behavior of the autoscaler.

Our original goal for this study was to explore the potential of combining serverless and containerized deployments for web applications. But due to the complexity of kubernetes, it took us a huge amount of time to figure out how to use kubernetes and how to govern a kubernetes cluster.The potential future work should try to combines these two forms together and do a cost analysis, and the researcher should be prepared to face a tough nut to crack.

VII. Contribution

Mondo Cluster setup, guidance, automating tests.
Shivam Container form application test design, wrk2 workload generation.
Thinh Serverless form application test design.

References

[1] M. Armbrust *et al.*, "Above the Clouds: A Berkeley View of Cloud Computing," Feb. 2009. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html

[2] "kind is a tool for running local Kubernetes clusters using Docker container "nodes".." [Online]. Available: https://kind.sigs.k8s.io/

[3] J. Li, S. G. Kulkarni, K. K. Ramakrishnan, and D. Li, "Understanding Open Source Serverless Platforms: Design Considerations and Performance," in *Proceedings of the 5th International Workshop on Serverless Computing*, in WOSC '19. Davis, CA, USA: Association for Computing Machinery, 2019, pp. 37–42. doi: 10.1145/3366623.3368139.

[4] "OpenFaaS makes it simple to deploy both functions and existing code to Kubernetes." [Online]. Available: https://www.openfaas.com/

[5] N. Kaviani, D. Kalinin, and M. Maximilien, "Towards Serverless as Commodity: a case of Knative," in *Proceedings of the 5th International Workshop on Serverless Computing*, in WOSC '19. Davis, CA, USA: Association for Computing Machinery, 2019, pp. 13–18. doi: 10.1145/3366623.3368135.

[6] E. Oakes *et al.*, "SOCK: Rapid Task Provisioning with Serverless-Optimized Containers," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, Boston, MA: USENIX Association, Jul. 2018, pp. 57–70. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/oakes

[7] A. Pérez, G. Moltó, M. Caballer, and A. Calatrava, "Serverless computing for container-based architectures," *Future Generation Computer Systems*, vol. 83, pp. 50–59, 2018, doi: https://doi.org/10.1016/j.future.2018.01.022.

[8] M.-N. Tran and Y. Kim, "Optimized resource usage with hybrid autoscaling system for knative serverless edge computing," *Future Generation Computer Systems*, vol. 152, pp. 304–316, 2024, doi: https://doi.org/10.1016/j.future.2023.11.010.

[9] Y. Gan *et al.*, "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*, in ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 3–18. doi: 10.1145/3297858.3304013.