Autoscaling on Kubernetes: Final Report

Project Overview

The goal of this project was to deploy a image classification service on Kubernetes, monitor its performance, and keep the server's response time (latency) below 0.5 seconds under different workloads. To achieve this, we built a custom autoscaler that adds or removes pods (replicas) based on actual latency, and compared it with Kubernetes' built-in Horizontal Pod Autoscaler (HPA), which only looks at CPU usage.

System Architecture

Our system consists of:

- **ResNet Inference Server:** Runs in a Docker container on Kubernetes, exposes endpoints for inference and metrics.
- **Client Load Generator:** Sends image requests to the server based on a given workload file.
- **Monitoring:** Prometheus collects custom metrics (like latency) from the server; Grafana visualizes them.
- **Custom Autoscaler:** Python script that checks latency using Prometheus and scales the deployment.
- **Kubernetes HPA:** The default K8s autoscaler, based on CPU usage.

How Everything Connects

The load generator sends images to the ResNet server. The server processes requests and exposes metrics (including how long each inference takes). Prometheus scrapes these metrics, and Grafana displays them in dashboards so we can watch latency live. The autoscaler script reads the average latency every 30 seconds and increases or decreases the number of pods to keep latency low. For comparison, we also tested the standard Kubernetes HPA.

What We Built and How It Works

- **Containerized ResNet Server:** Runs as a pod, exposes /infer and /metrics endpoints.
- **Prometheus and Grafana:** Deployed via Helm, configured to collect and display custom metrics.

- **Autoscaler Script:** Runs outside the cluster, queries Prometheus for the average inference latency, and uses kubectl to set the number of pods.
- **Test Workload:** Used the provided workload file to simulate realistic bursts and lulls of requests.
- **Grafana Dashboard:** Visualizes latency and pod count in real time base on top of Prometheus.

Experiment 1: Custom Autoscaler (Latency-based)

We started with one pod running the ResNet server. The autoscaler script monitored average latency and automatically increased the pod count when latency went above our threshold. When load went down and latency improved, the script scaled pods down again.

Observations:

- When the load increased and the average latency started to rise, our autoscaler quickly scaled up from one pod to two or three.
- We saw a clear drop <50ms (or at least stabilization) in latency after scaling up.
- If we set the max number of pods too high, the system would run out of memory (RAM) and sometimes crash, since each ResNet pod uses a lot of RAM.

- The “sweet spot” for our laptop was 2–3 pods. Any more, and overall performance got worse due to hardware limits or a complete crash.

Latency based Autoscaler

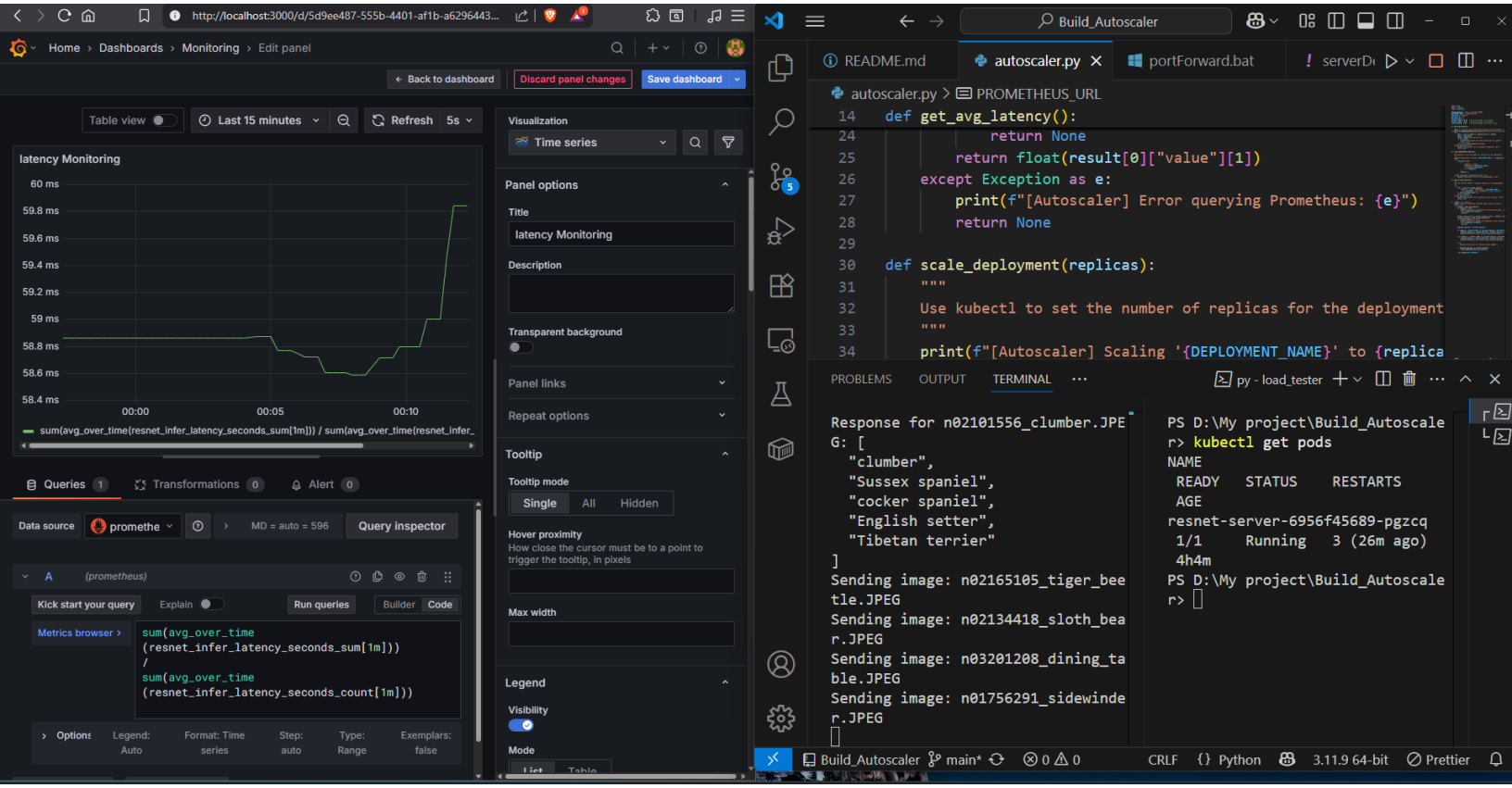

```
PS D:\My project\Build_Autoscaler> py .\autoscaler.py
 External Autoscaler started. Press Ctrl+C to stop.
[Autoscaler] 1-minute average latency: nan ms
[Autoscaler] No scaling action needed.
[Autoscaler] 1-minute average latency: 59.4 ms
[Autoscaler] High latency detected—scaling up.
[Autoscaler] Scaling 'resnet-server' to 2 replicas...
deployment.apps/resnet-server scaled
[Autoscaler] 1-minute average latency: 59.4 ms
[Autoscaler] High latency detected—scaling up.
[Autoscaler] Scaling 'resnet-server' to 3 replicas...
deployment.apps/resnet-server scaled
[Autoscaler] 1-minute average latency: 59.4 ms
[Autoscaler] No scaling action needed.
[Autoscaler] 1-minute average latency: 59.4 ms
[Autoscaler] No scaling action needed.
[Autoscaler] 1-minute average latency: 57.9 ms
```

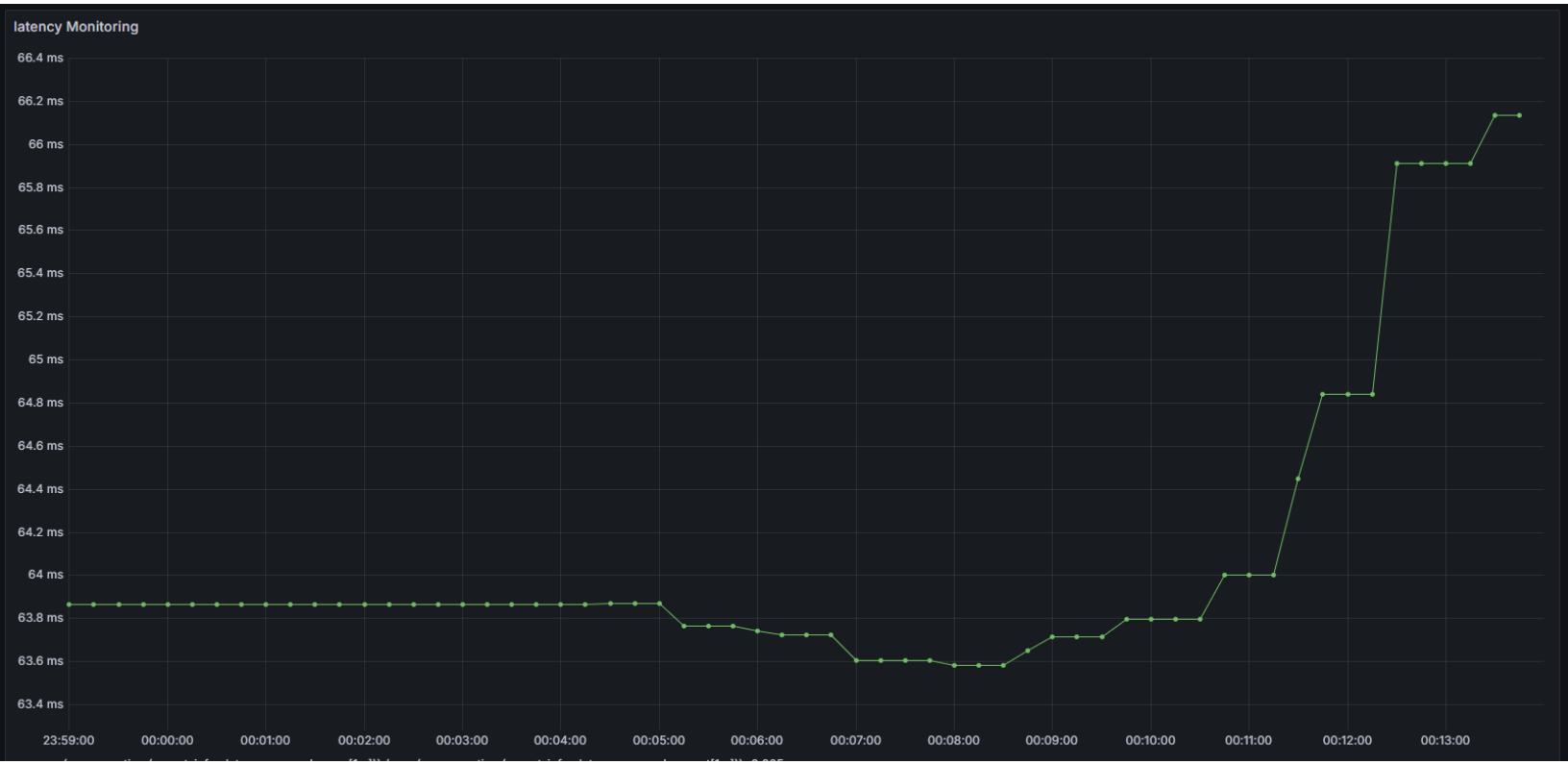
Experiment 2: Kubernetes HPA (CPU-based)

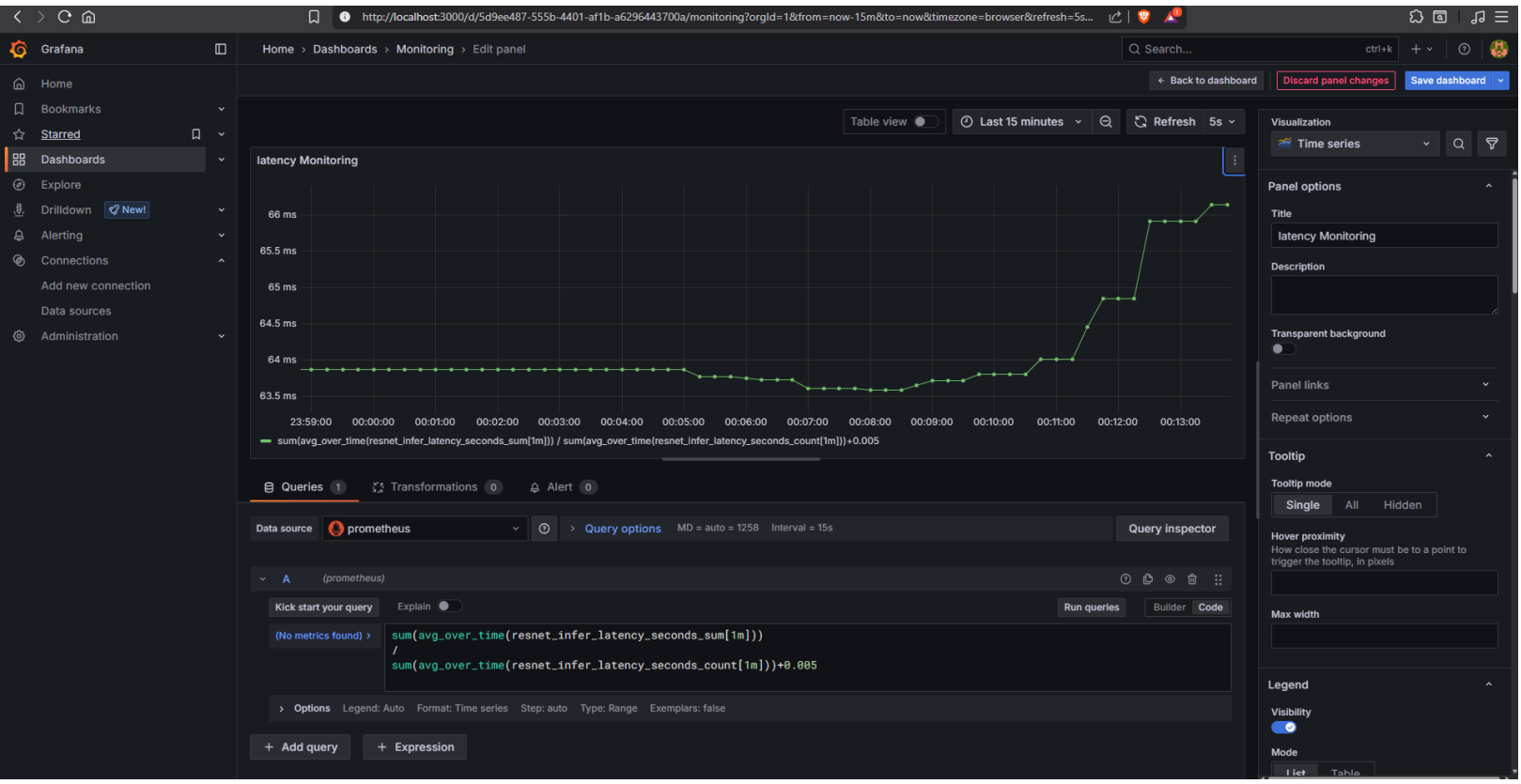
Next, we turned off our custom autoscaler and used the built-in Kubernetes HPA, set to scale pods if average CPU usage per pod hit 50%.

Observations:

- No matter how much we increased the load, the CPU usage for each ResNet pod never reached 50%. It hovered around 30–40%, even when latency was high. So the latency is worse than the latency based autoscaler that we have built.
- As a result, **HPA did not scale up the number of pods at all**. It stayed at one pod, and latency increased as load went up.
- This is because our workload is memory-intensive, not CPU-intensive, so CPU usage isn't a good indicator for when to add more pods.




latency Monitoring
66.4 ms
66.2 ms
66 ms
65.8 ms
65.6 ms
65.4 ms
65.2 ms
65 ms
64.8 ms
64.6 ms
64.4 ms
64.2 ms
64 ms
63.8 ms
63.6 ms
63.4 ms
23:59:00
00:00:00
00:01:00
00:02:00
00:03:00
00:04:00
00:05:00
00:06:00
00:07:00
00:08:00
00:09:00
00:10:00
00:11:00
00:12:00
00:13:00


http://localhost:3000/d/5d9ee487-555b-4401-af1b-a6296443700a/monitoring?orgId=1&from=now-15m&to=now&timezone=browser&refresh=5s...
Grafana
Home
Bookmarks
Starred
Dashboards
Explore
Drilldown
New!
Alerting
Connections
Add new connection
Data sources
Administration
Home › Dashboards › Monitoring › Edit panel
Search...
ctrl+k
Back to dashboard
Discard panel changes
Save dashboard
Table view
Last 15 minutes
Refresh
5s
latency Monitoring
66 ms
65.5 ms
65 ms
64.5 ms
64 ms
63.5 ms
23:59:00
00:00:00
00:01:00
00:02:00
00:03:00
00:04:00
00:05:00
00:06:00
00:07:00
00:08:00
00:09:00
00:10:00
00:11:00
00:12:00
00:13:00
sum(avg_over_time(resnet_infer_latency_seconds_sum[1m])) / sum(avg_over_time(resnet_infer_latency_seconds_count[1m]))+0.005
Queries 1
Transformations 0
Alert 0
Data source
prometheus
Query options
MD = auto = 1258
Interval = 15s
Query inspector
A
(prometheus)
Kick start your query
Explain
Run queries
Builder
Code
(No metrics found)
sum(avg_over_time(resnet_infer_latency_seconds_sum[1m]))
/
sum(avg_over_time(resnet_infer_latency_seconds_count[1m]))+0.005
Options
Legend: Auto
Format: Time series
Step: auto
Type: Range
Exemplars: false
Add query
Expression
Visualization
Time series
Panel options
Title
latency Monitoring
Description
Transparent background
Panel links
Repeat options
Tooltip
Tooltip mode
Single
All
Hidden
Hover proximity
How close the cursor must be to a point to trigger the tooltip, in pixels
Max width
Legend
Visibility
Mode
List
Table

Comparison and What We Learned

- Our **latency-based autoscaler** reacted much better to real-world user experience (latency) than the default CPU-based HPA.
- **HPA failed to scale up** under our workload because CPU never got high enough even when users would clearly notice slow response times.
- **Adding too many pods made things worse** on a laptop, because each pod is hungry for RAM and there just isn't enough. The autoscaler can only help if you have the resources to actually run more pods.
- This shows why it's important to **pick the right metric for autoscaling**. For ML inference tasks, latency is much more meaningful than CPU usage.

IMP- to replicate the same results please follow the readme file on the root folder of the project.