Evaluating Load Balancing Performance in a Containerized Cloud Environment

Albert Ye, Joe Wang, Veronica Cheung

I. Introduction

In the past few years, advances in containerization have made it easier to run multiple isolated instances on a single device. This has reduced the costs to simulate a distributed network to the cost of a single VM. However, there are some disadvantages such as the competition for resources. In this project, we explored how software-defined load balancing behaves when services actively compete for a shared, limited resource pool. We then compared this with theoretical knowledge about the subject.

II. Configuration

We ran our SDN interface on a single VM running Debian 12 on KVM/QEMU, with the bare-metal computer powered by an AMD Ryzen 9 7950X. As can be viewed in our Docker Compose file, we created five server containers. For load balancing, we did an Nginx instance configured to perform the least-connections algorithm.

Our servers would perform matrix multiplication between two **N**×**N** matrices. In doing this, our load balancer would send requests to one of the servers according to our load-balancing algorithm. The least connections balancer, for example, would assign connections according to which server is running the fewest current tasks.

```
from fastapi import FastAPI
import numpy as np
import time
from fastapi.responses import PlainTextResponse

app = FastAPI()

def compute_heavy_operation(size=3500):
	A = np.random.rand(size, size)
	B = np.random.rand(size, size)
	C = np.dot(A, B)
```

```
        return C

@app.get("/", response_class=PlainTextResponse)
async def index():
        compute_heavy_operation()
        return "Operation concluded!"

if __name__ == '__main__':
        import uvicorn
        uvicorn.run(app, host="0.0.0.0", port=5000)
```

Our clients are simulated concurrent requests. It sends a user-defined number of requests to the load balancer with a given size **N**. The load balancer will then, according to the load-balancing algorithm defined in the Nginx config, send the request to one of the servers.

```
import requests
import time
import argparse
from concurrent.futures import ThreadPoolExecutor

def send_request(url):
        try:
                response = requests.get(url)
                return response.status_code,
                        response.elapsed.total_seconds()
        except Exception:
                return 0, None  # Consistent format for error

def load_test(url, num_requests=10, num_threads=5):
        with ThreadPoolExecutor(max_workers=num_threads) as executor:
        futures = [executor.submit(send_request, url) for _ in
                range(num_requests)]
        results = [future.result() for future in futures]
        return results
```

"Number of threads" refers to the number of concurrent clients, and an increased number of threads generates increased load for a given load size.

We used FastAPI to handle the client-server relationship, which is simple because FastAPI automatically manages HTTP routing, request handling, and asynchronous processing with minimal configuration. It allows us to easily define server behavior using Python functions while automatically generating API documentation and handling concurrent requests efficiently—making it an ideal framework for simulating services in a load-balanced, containerized environment.

III. Methodology

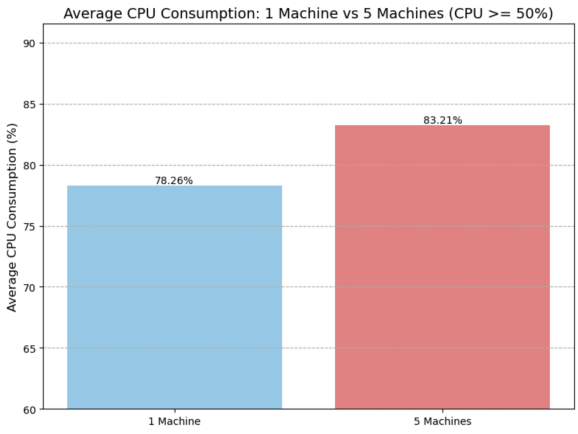
For each combination of parameters (number of threads, computational load, number of servers):

1. Run a batch of 10 runs, collect their CPU and response time.
2. Restart the server to minimize confounding factors like CPU caching.
3. Run a batch of 10 runs again, collect their CPU and response time.
4. Compute averages across all 20 runs, 20 data points per combination.

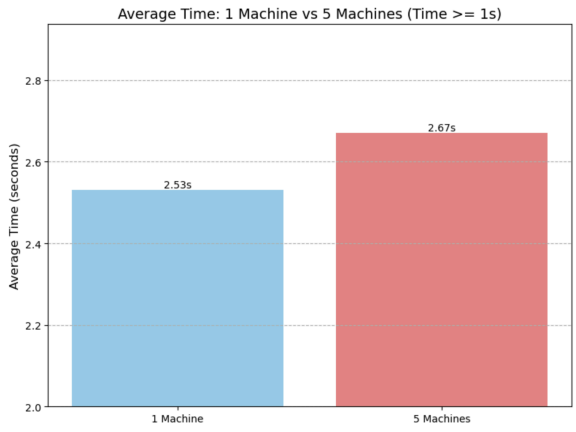
For our plots, we used best-fit lines for latency vs. threads, and best-fit cubics for CPU and latency vs. load as matrix multiplication is roughly $O(N^3)$.

IV. Metrics

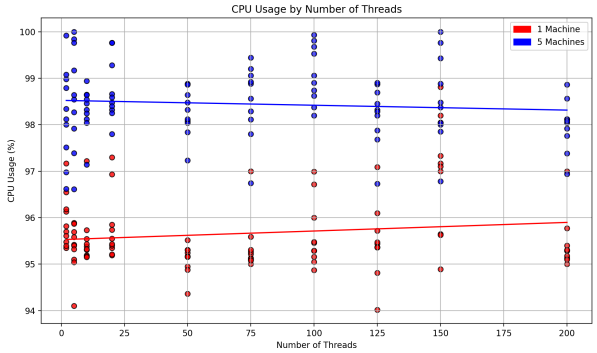
Average CPU Consumption



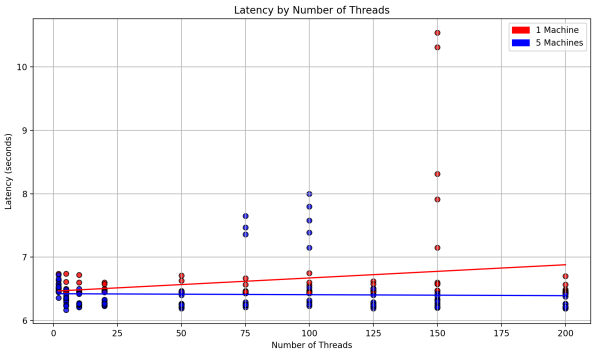
Average Latency



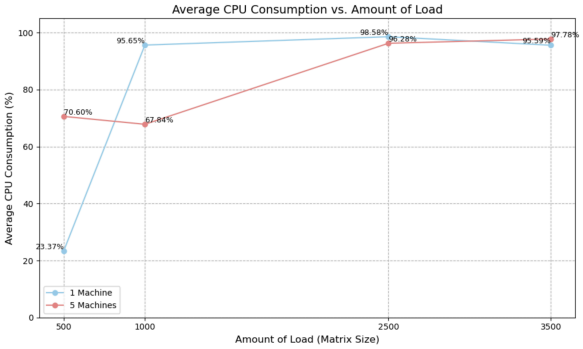
CPU Usage vs. Threads (n = 3500)



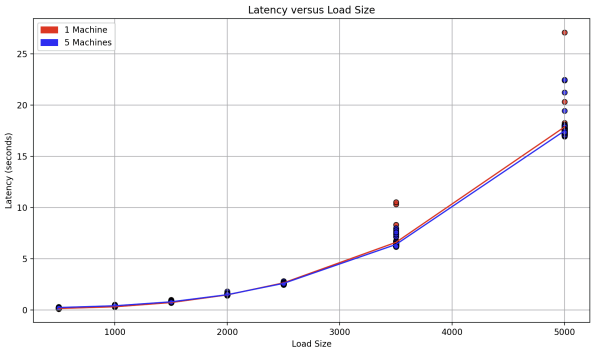
Latency vs. Threads (n = 3500)



CPU vs. Load

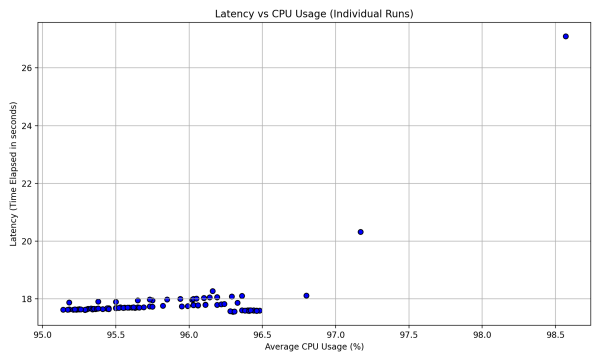


Latency vs. Load

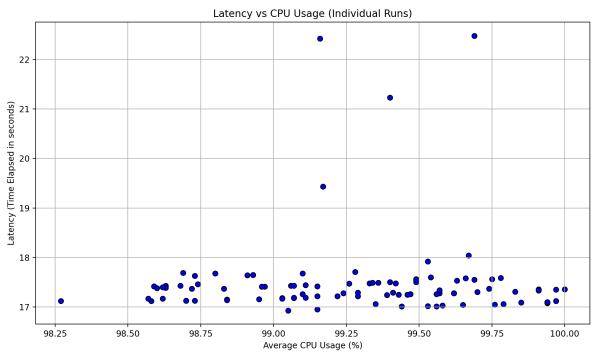


Latency vs. CPU Usage

(1 machine, n=5000)



(5 machines)

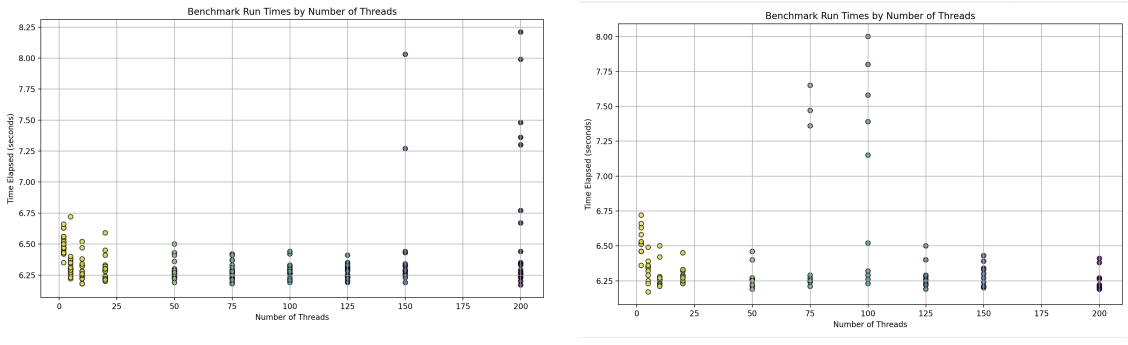


V. Analysis

One interesting result is the comparison of latency against the number of concurrent requests. When we have a very small number of concurrent requests, increasing the number of containers increases the amount of time for all requests to go through. However, when we increase the number of requests, load-balancing 5 containers works better than just having 1. In theory, we can model the k-machine example as an M/M/k queue with equal arrival rates for each queue. However, for a small number of concurrent requests, we find that the arrival rates are not equal as the queueing time is significantly larger than the processing time. The least-connections algorithm does not help in this case, as there is nothing inflight whenever new tasks are scheduled. The added overhead of using the Nginx load-balancer would increase the total time. In this case, testing an algorithm like round-robin which evenly distributes packets regardless may prove beneficial. This would back up the theoretical claim that load balancing for smaller loads is not beneficial because the queueing delay is so small that the overhead of queue insertions and deletions becomes nontrivial.

However, for larger numbers of concurrent requests we see that expected latency increases slightly as a function of thread count for 1 machine while it decreases slightly as a function of thread count for 5 machines.

Despite this, we notice a couple of outliers for each load size. These outliers have been observed only with high numbers of concurrent requests, and have appeared when running the tests in all orders to eliminate the chance of CPU caching.



Left: Running benchmarks for N=3500 from high to low thread counts

Right: Running benchmarks for N=3500 from low to high thread counts

In the 1-machine example, such outliers can easily be attributed to spikes in CPU usage, where all outliers had significantly higher CPU usage than the non-outlier cases. However, in the 5-machine example, the outliers are not necessarily related to total CPU usage. We theorize that the reason for such outliers are certain spikes in CPU usage during a run, as opposed to consistent high usage. The server may be running other background processes that influence the amount of resources provisioned to each Docker container.

Unfortunately, due to the limitations of time and compute, we were unable to test the differences for larger values of N, or larger numbers of load balancers. However, given the divergence between best-fit curves at N=3500, and in line with the idea that adding more queues to a network decreases the queueing delay for each queue, we hypothesize that there will be wider gaps in latency between 1 and 5 machines for larger N. However, the benefits will be expected to be diminished compared to 5 standalone machines because Docker must ration limited memory across all containers.[3]

Overall, this project provides empirical evidence to reinforce the theoretical notions of when to use load balancing. For smaller loads and fewer requests, load balancing has been shown to either have no or adverse effect on the latency of the requests. However, as load sizes and request sizes scale, the need for a load balancer begins to show itself.

This project also demonstrates the limitations of inexpensive NFV solutions. By virtualizing the orchestrator as well as using the Docker network, we have created situations where there is competition for resources between supposedly separate members of the same network. Thus, for production-grade load balancing, it is crucial to understand the models of performance versus cost. This model of software defined load balancer running on the same host as the servers is widely used in industry production systems. For example, in Kubernetes - a popular tool for container orchestration for production systems -, the Ingress component is modeled after Nginx[4]. It runs as a deployment – essentially a container – that forwards requests to other deployments (ie. application servers), to achieve redundancy and fault tolerance.

VI. References

[1] NGINX Inc. 2025. NGINX Ingress Controller. (2025). Retrieved April 14, 2025 from https://github.com/nginxinc/kubernetes-ingress

[2] Gogineni, N., & Sivalingam, S. M. (2024). A systematic review on recent methods of scheduling and load balancing for containers in distributed environments. International Journal of Advanced Technology and Engineering Exploration, 11(116), 1030-1048. doi:https://doi.org/10.19101/IJATEE.2023.10102431

[3] Docker Inc. 2025. Resource constraints. (2025). Retrieved May 9, 2025 from https://docs.docker.com/engine/containers/resource_constraints/

[4] The Linux Foundation. 2025. Ingress. Retrieved May 9, 2025 from https://kubernetes.io/docs/concepts/services-networking/ingress/

Our project source code and data are in this github repository: https://github.com/Storce/EE122-Project-SP2025