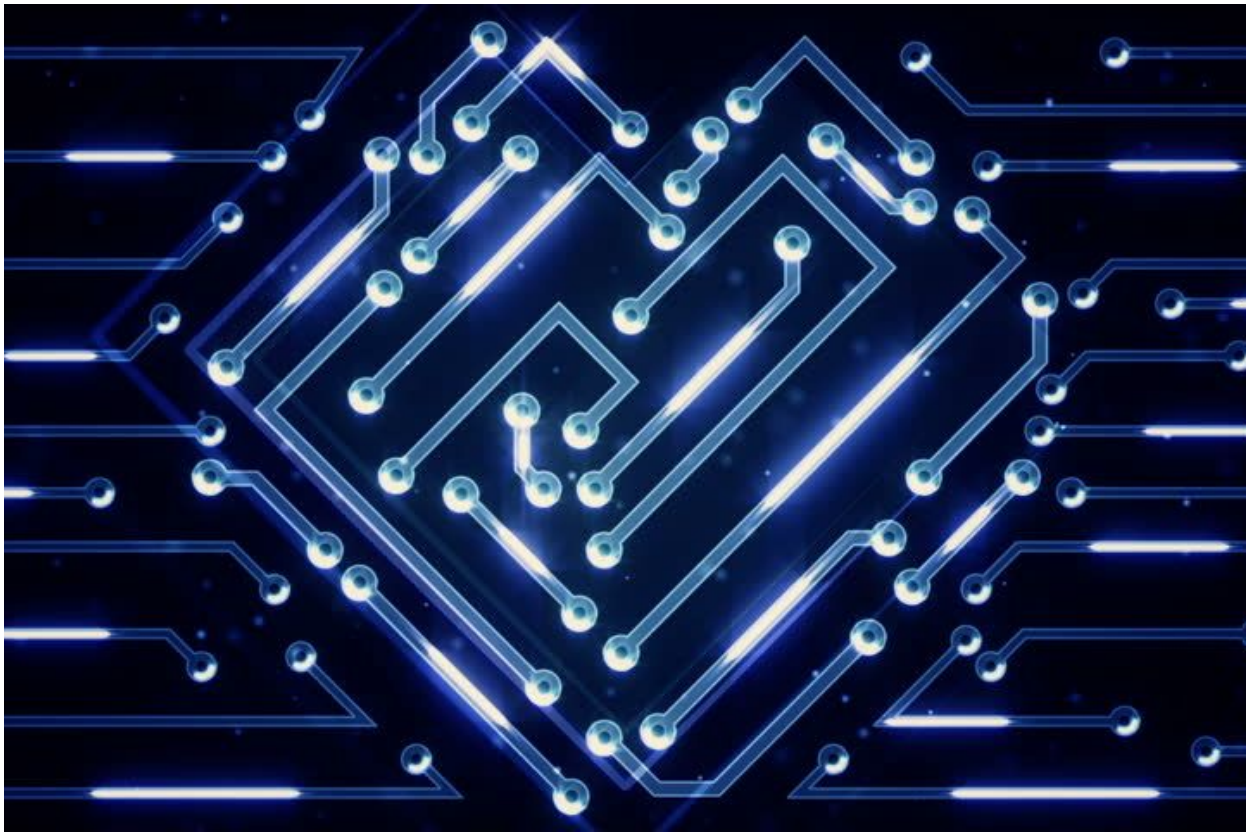


Implementing a software framework for serverless /Function-as-a-service(FaaS) computing model

CS 695: Topics in Virtualization and Cloud Computing (Spring ‘19)

Members

Surender Singh Lamba, 140050075

Mridul Sayana, 140050001

Problem Context & Description

.This project is trying to address the problem of creating a software framework to create and host functions on private hardware(laptops, pcs, servers) under Function-as-a-service(FaaS) model which is a system of hosting mostly stateless functions with a limit on memory and runtime and scale them proportional to demand and is fully handled by the provider and the client just provides the function to be hosted. Multiple functions can call each other in this setup.

The FaaS model as opposed to a central server architecture divides the functionality into components like Auth0, databases etc. and has the principal code running on the client itself which talks to these individual components over http/https. Computationally heavy or security sensitive code like for transactions is placed behind an API gateway where it in turn access various databases in an event-driven manner. A lot of these components are handled by third party BaaS providers like Google Firebase for user management, Amazon S3 for storage etc.

Some examples of existing FaaS providers include Amazon Lambda and Azure Functions, but these providers feature heavily integrated code along with their storage and database solutions and custom template code for FaaS that they result in provider lock in. There are some open-source solutions to this problem like OpenFaaS which will form our source of guidance through this project where we will code a more bare bones, possibly fast form of this architecture.

Main Components & Challenges

The main design of the project can be broken down into a few components including:

1. Proxy/Gateway to handle requests between the client and the cluster
2. Actually hosting the cluster with multiple hardware nodes
3. Hosting the functions in this cluster with input/output capability
4. Metrics collection and triggers to scale the function depending upon the demand
5. Having a network spanning the various nodes so that requests can be load balanced and there is high availability ensured, so that even if one node goes down, we recover and instantiate the the replicas on remaining nodes

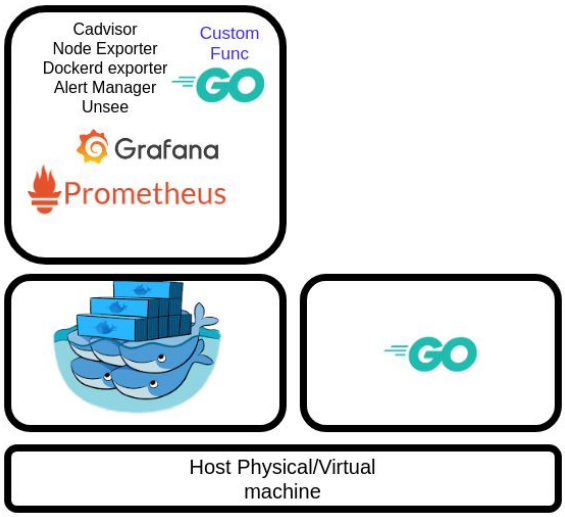
The main challenges included designing the function container itself along with the input/output capability, so that our function can execute and also keeping the size of this container limited so as to host as many functions as possible.

Another challenge was the methodology to collect and act on the metrics, there can be basically either pull method which periodically pulls metrics from all the containers and another is push metrics in which the container itself periodically pushes metrics. We went ahead with a pull model.

The remaining challenge was building a scaling system as it is not natively available in the cluster manager we chose(Docker Swarm). We also had to circumvent some kernel param limits for inotify file watches etc which were getting exhausted while running metrics collectors and we also had to manage some scripting to automatically pull the newly created function images on all the nodes(manager creates it originally) so that when we scale, all the nodes can instantiate it, we used ssh servers and sshpass tools to run the pulls and circumvented the sudo permissions for /usr/bin/docker, so that we can run these commands in an automated manner over ssh.

Components Design

We used docker swarm for the cluster manager because it was lightweight and really easy to get started compared to Kubernetes even though kubernetes has native scaling and docker swarm does not have it. Our container stack is shown in the image below.

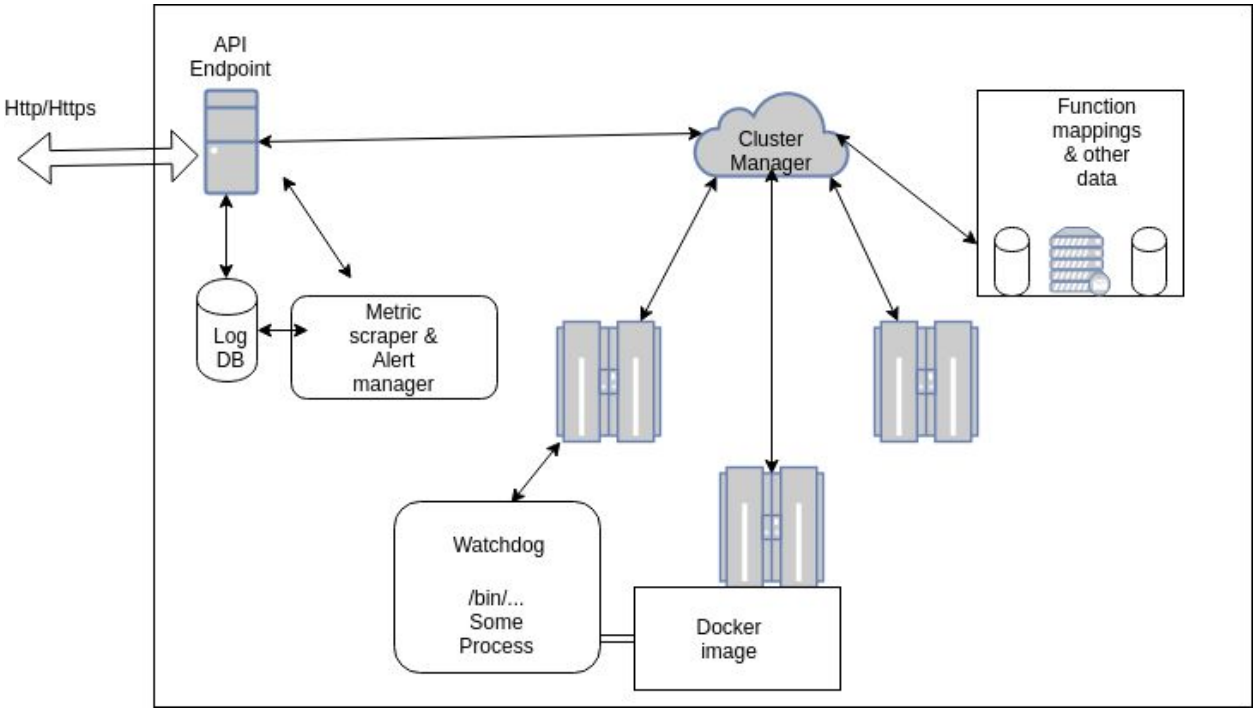


We basically host our functions in a container with the statically compiled version of the function and a statically compiled GO server. This GO server handles the requests made and returns the output and also exposes metrics of the server like number of requests and response time which can be pulled by Prometheus every 5 secs. Similarly there are various metrics exporters like cadvisor(containers metrics), dockerd-exporter (Docker daemon

metrics), node-exporter (host metrics). We can visualize them in grafana. When a predefined alert query satisfies, we ring an alert to the Golang server running on the host which scales the service using a bash script.

Components Implementation Details

The working diagram of the system is shown below:



API Endpoints/Proxy Gateway

This component serves as the proxy to receive http/https requests from outside and maps the function requested and forwards it to the cluster manager to run the function and returns the result. We use Haproxy for this and run a round robin method on all the available nodes to external load balance the system (There is existing internal load balancing in Docker Swarm)

Metrics and Alert Manager

All the requests received at API endpoints are logged in containers and a manager like prometheus is used which can query data from a live time series stream and this is used to detect rise and fall in demand. This rise and fall in demand triggers alert manager which fires a request to the Go server running on cluster manager to scale the containers to a higher/lower value of replicas.

Cluster Manager

We use a cluster manager Docker Swarm and initialize our cluster of machines/servers and our admin hosts the api endpoints too. This talks with the watchdogs for appropriate functions so it can scale that particular function. Cluster manager also helps in load balancing containers across the cluster and also providing availability if one node crashes.

Watchdogs

A watchdog becomes a init process in the container with an embedded Golang server and monitors the function and relays back result. It also collects metrics and exposes them to prometheus to scrape. Go also implements concurrency so that it can handle a lot of requests by itself, we scale to let each container respond to requests between 50 requests/second and 150 requests/second.

Container Images

These are created after we write the function required to host and are used to create multiple instances of the function during autoscaling. We push these images to Docker hub which forms our registry to let other nodes download it to host replicas.

OpenSSH server

We have ssh servers running all nodes so that we can automate a lot of tasks like initializing the swarm and make all the nodes join it by sharing the token over ssh and also downloading newly created function images from registry, so that all the nodes can host the functions.

Experimental Evaluation

Our experimental evaluation mostly revolves around seeing if the system works and what are the general metrics, so we ran two main experiments.

Setup

We used two laptops as two nodes of the cluster with one acting as a manager and a third system(PC) ran the proxy and also ran the benchmarking requests.

Manager:

Intel® Core™ i3-2365M CPU @ 1.40GHz × 4

Intel® Sandybridge Mobile

5.7 GiB RAM, Ubuntu 16.04 LTS 64-bit, 232.3 GB Hard Disk

Worker:

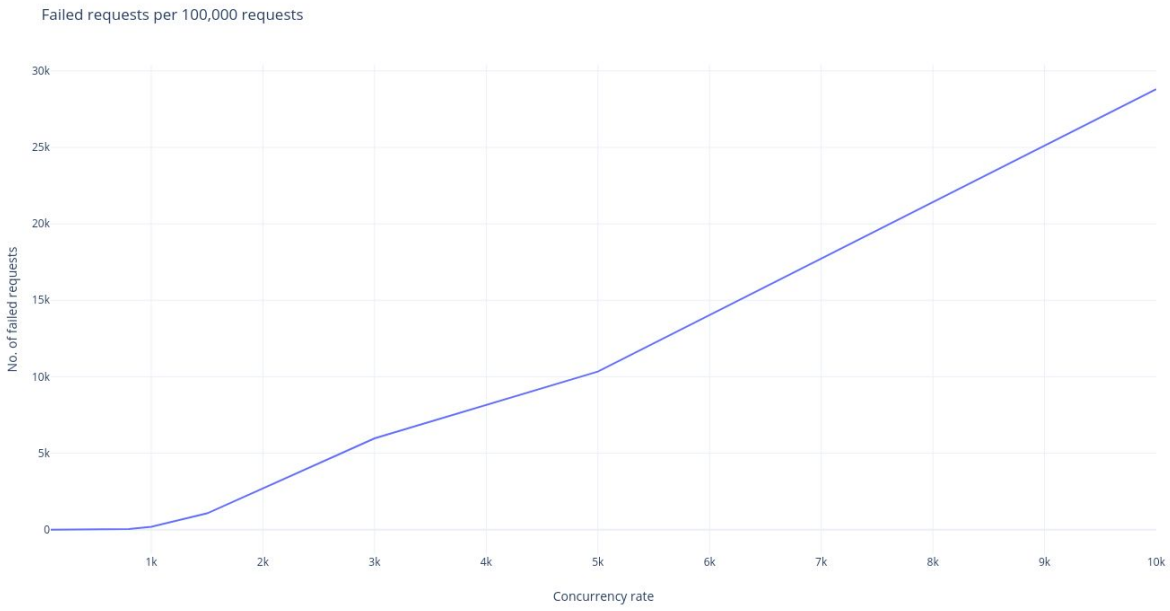
Intel® Core™ i5-7200U CPU @ 2.50GHz × 4

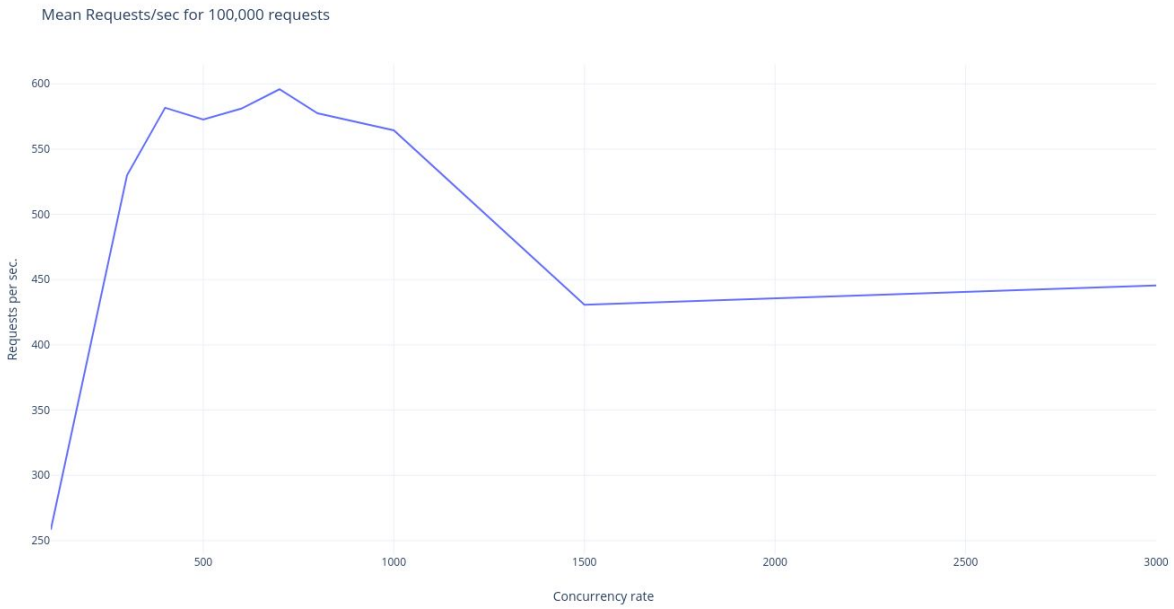
Intel® HD Graphics 620 (Kaby Lake GT2)

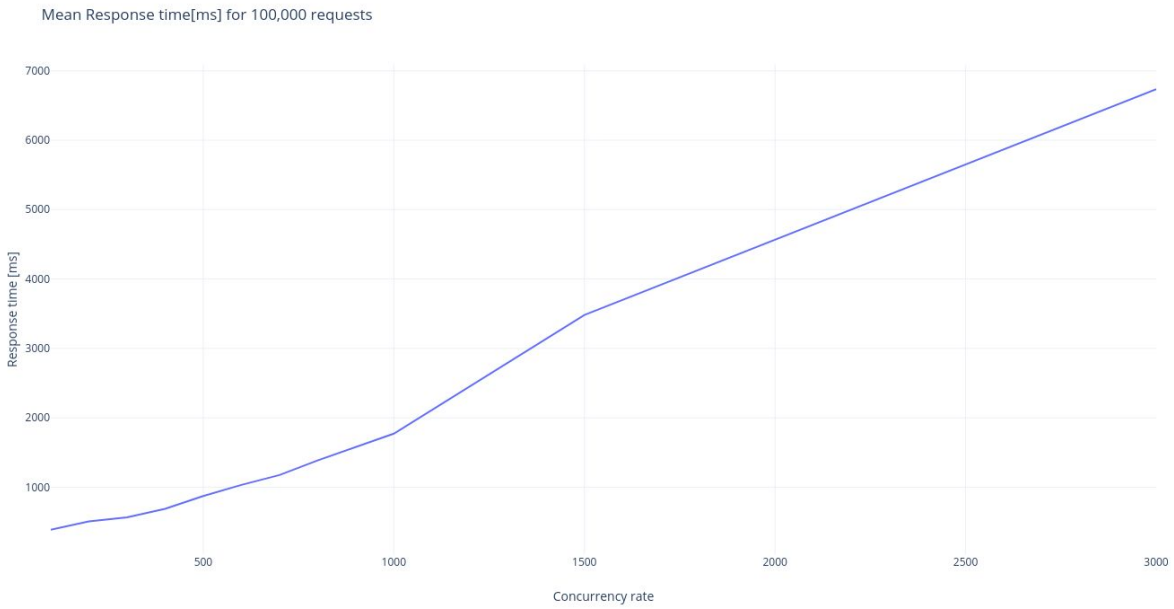
7.7GiB RAM, Ubuntu 16.04 LTS 64-bit, 209.1 GB Hard Disk

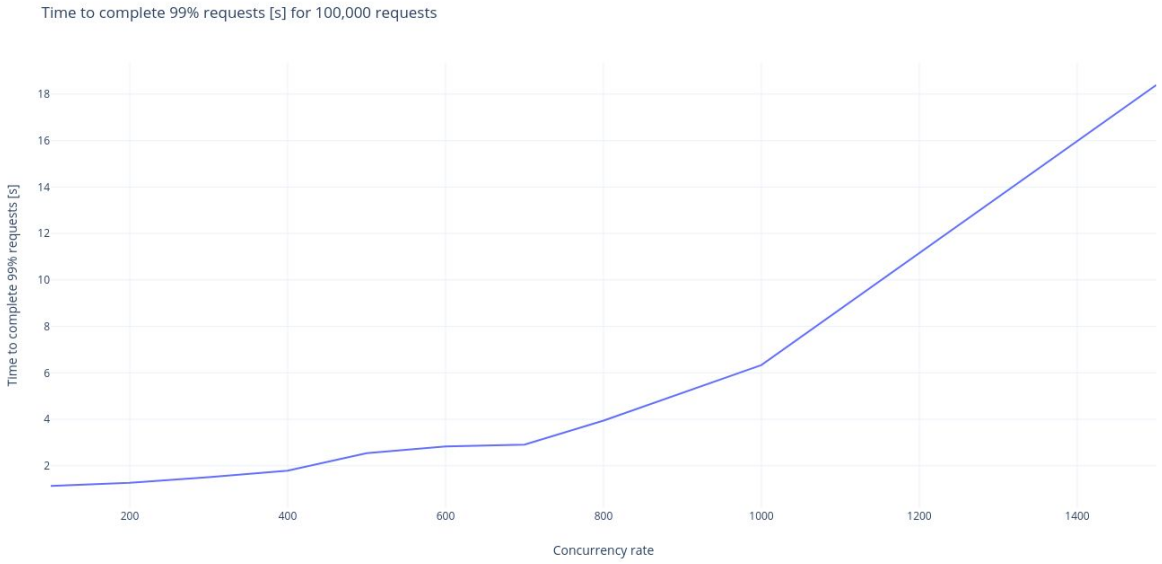
Experiments

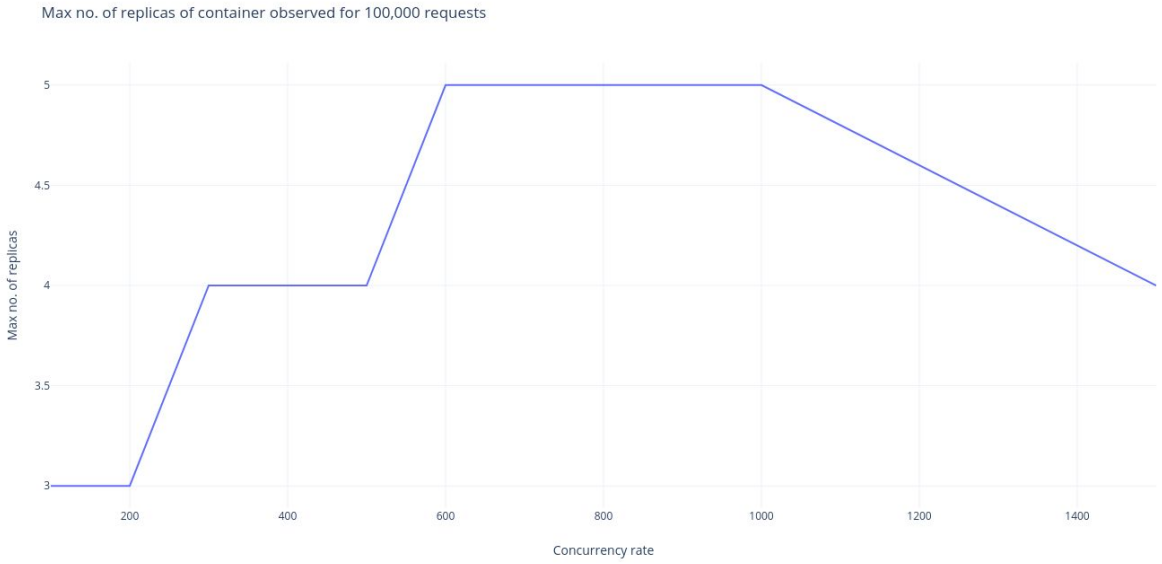
1. We used ab(Apache HTTP server benchmarking tool) to run 100,000 requests at various concurrency rates and obtained various metrics

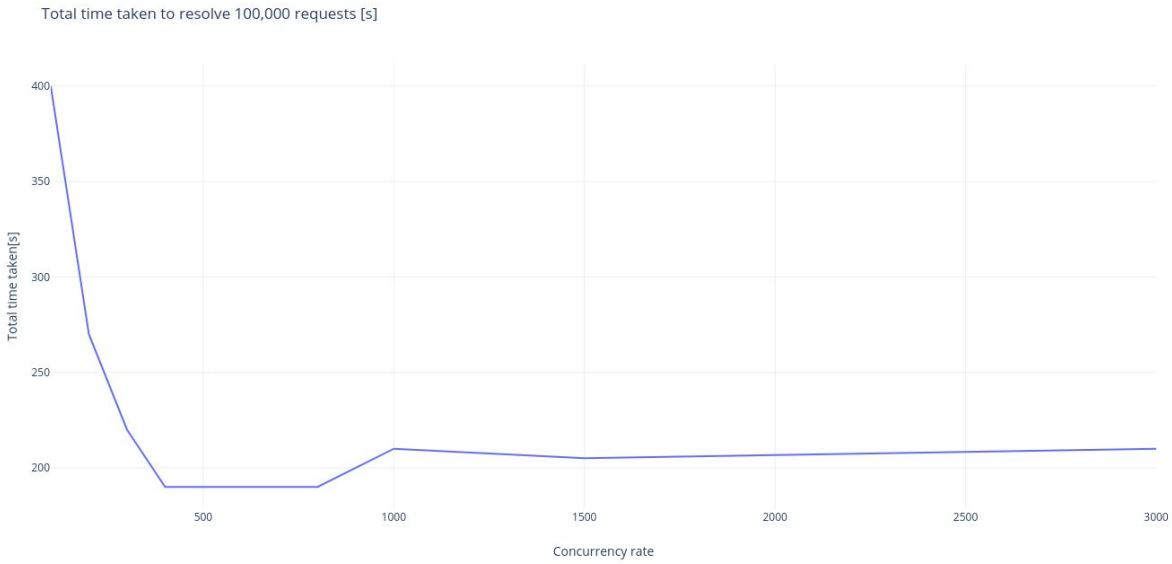



Mean Requests/sec for 100,000 requests
600
550
500
450
400
350
300
250
Requests per sec.
500
1000
1500
2000
2500
3000
Concurrency rate

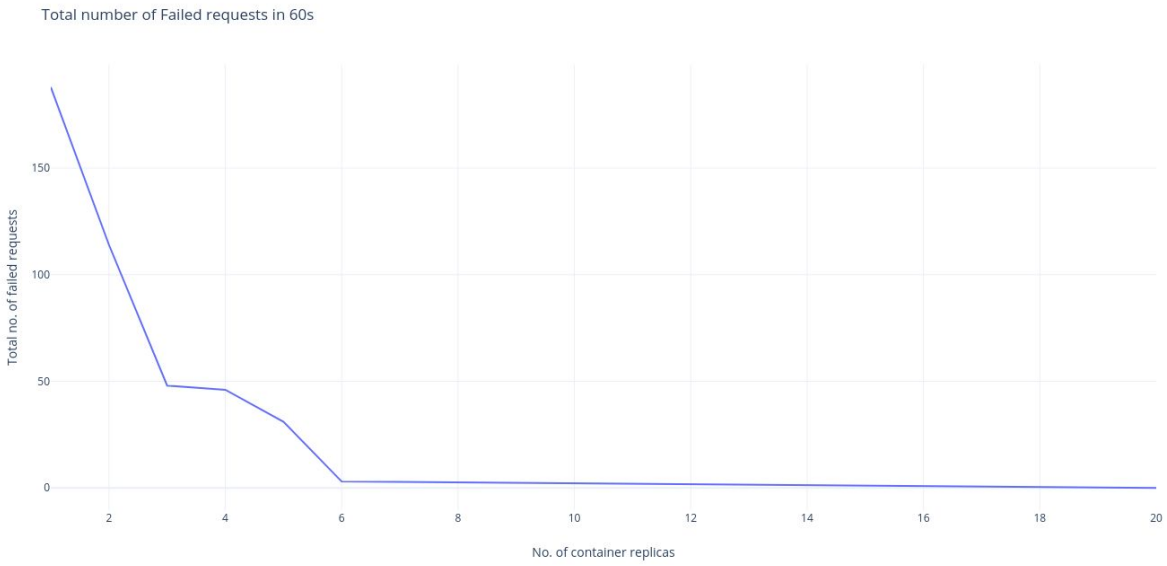

Mean Response time[ms] for 100,000 requests
7000
6000
5000
4000
3000
2000
1000
Response time [ms]
500
1000
1500
2000
2500
3000
Concurrency rate

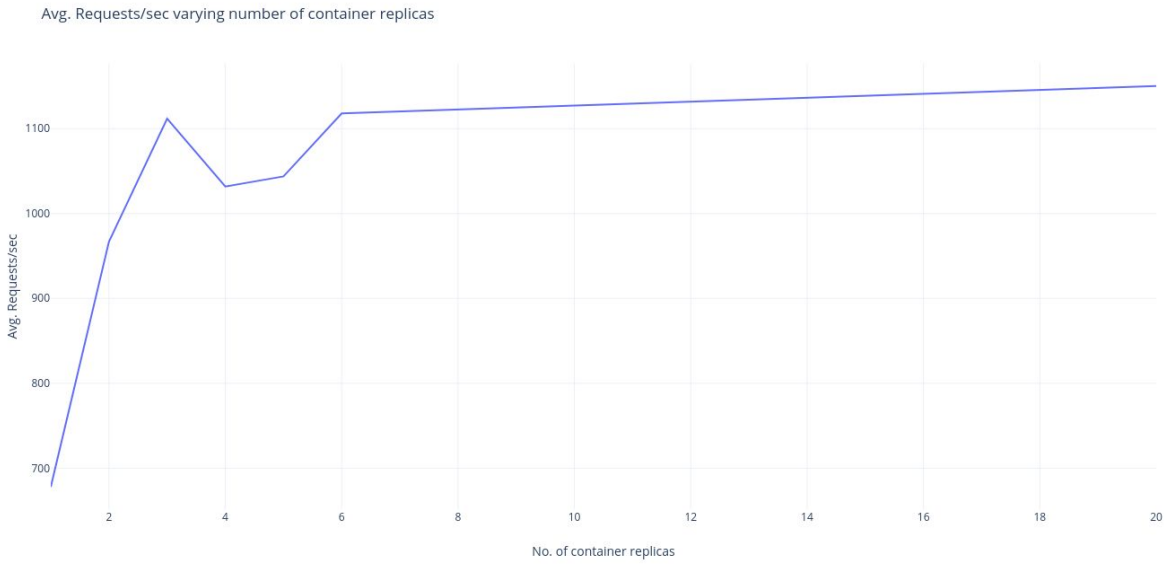

Time to complete 99% requests [s] for 100,000 requests
Time to complete 99% requests [s]
18
16
14
12
10
8
6
4
2
200
400
600
800
1000
1200
1400
Concurrency rate

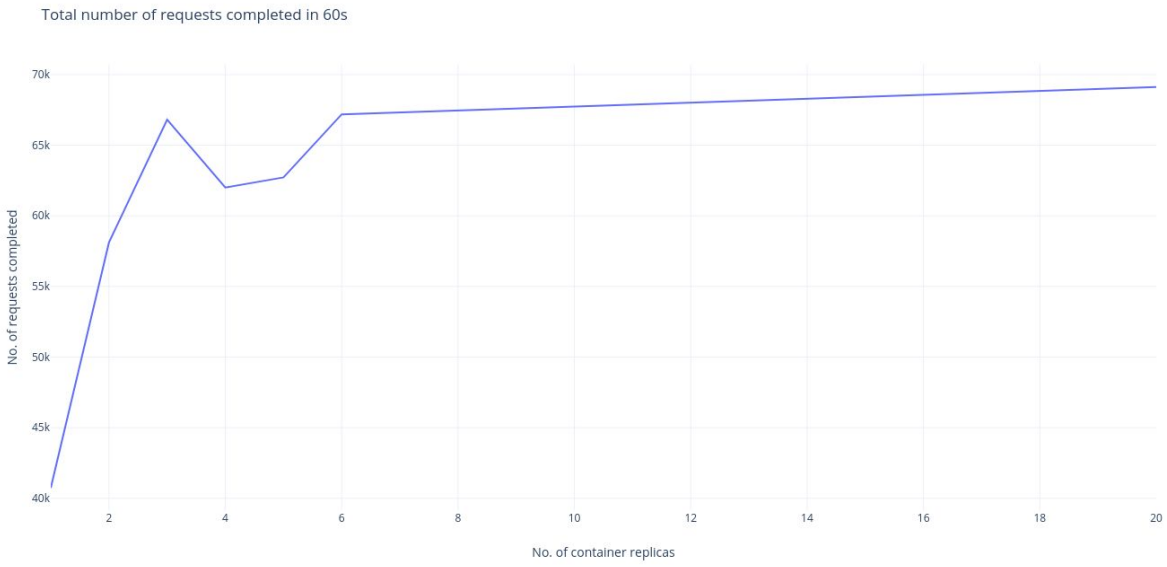


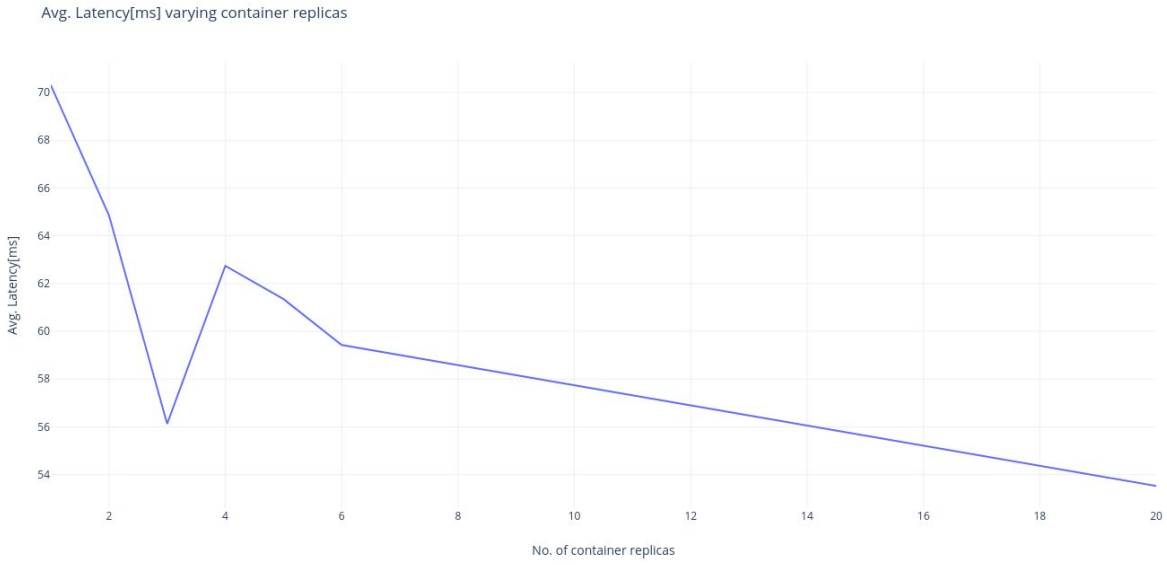


2. We used WRK tool to stress test the system with various number of replicas of the functions active and obtained the following graphs


Total number of Failed requests in 60s
Total no. of failed requests
150
100
50
0
2
4
6
8
10
12
14
16
18
20
No. of container replicas


Avg. Requests/sec varying number of container replicas
Avg. Requests/sec
1100
1000
900
800
700
2
4
6
8
10
12
14
16
18
20
No. of container replicas





These graphs show various metrics for a simple hosted function which is written in c++ and outputs the params sent to it.

We also measured that on average, each container is 9 MB in memory and thus provides high density of hosted functions.

The startup time of the container varied and was on average below 5 seconds.

The main bottleneck observed was CPU which was getting used up by 1000 concurrent requests and just 5 or 6 containers due to inbuilt concurrency in GO servers.

We used the perf tool to observe which kernel syscalls was the most limiting one and observed that many of the concurrent threads were waiting on the futex syscall which suggests that file operations are also proving expensive.

Scaling in action:

Interesting Aspects & Future Work

The best thing about the project was to experience how easy it is to export pre-built environments in docker and how lightweight and easy it is to develop a system over a swarm and the ingress network for internal load balancing is amazing.

The most interesting thing was to see the power of statically compiled versions of Go servers and functions in containers, which brought down the size of the container from 300 MB to 10 MB because a non static version would require that all the dependencies are present in the container, this dramatically increased the possible function density on the given hardware and also startup time.

We observed that prometheus takes a lot of memory to create and store data(around 300MB) and uses significant cpu, grafana is also not the most ideal tool to visualize. We found a new tool NetData at the end of the project which is really efficient in this matter and runs on 1% cpu and has a cycle time as low as 1 second for calculating every metric, We think netdata will provide a more agile system to scale the containers which can respond to smaller spikes in time. It also natively has visualization tool and alert system which can run scripts itself, so our setup of a GO server on host is also removed and I think it will be a more tidy approach to go about in the extension.

Further future work can be explored to also add cloud infrastructure rather than just private hardware and segment the stack so that computationally heavy tasks are run on the cloud and more time sensitive tasks can be run on private specialized hardware for better performance.