System Design Capstone (SDC)

Engineering Journal and Developer's Notes
Will Franceschini, MCSP-20, Galvanize

Day 1 (23 May 2023)

✓ Getting familiar with the SDC requirements:

- Gather metrics about the current performance of the application
- Research and experiment with methods of optimizing and scaling the application
- Take detailed notes on all your work and findings in the engineering journal
- Report the results on a written and video format

✓ Getting individual and group GitHub dashboards setup:

- Project repo — https://github.com/Chair-B-nBeyond
- Group dashboard — https://github.com/orgs/Chair-B-nBeyond/projects
- Individual dashboard — https://github.com/orgs/Chair-B-nBeyond/projects/4

✓ Complete the project submission form
✓ Complete daily journal entry

Day 2 (24 May 2023)

✓ Get own copy of the application up and running
✓ Chose optimization path:

API/server optimization

✓ Complete daily journal entry

Day 3 (24 May 2023)

✓ Research optimization tools

Grafana Labs k6

k6 supports three execution modes to run a test; local, distributed, and cloud. The local test occurs on a single machine, container, or CI server. I decided on the dockerized version of k6. Terminal output for the installation is below:

```
will.franceschini chairbnBeyond % docker pull grafana/k6
Using default tag: latest
```

```
latest: Pulling from grafana/k6
f56be85fc22e: Pull complete
211f8214916f: Pull complete
166b2807661b: Pull complete
Digest:
sha256:acad1a06a5e32b5f04f9dbfad112075c8c71b6be8d664bd9d8f79ce6b9559b3d
Status: Downloaded newer image for grafana/k6:latest
docker.io/grafana/k6:latest
```

Developed the following ‘docker run’ startup script for k6:

```
docker run -d --rm --name k6-test -p 3060:3060 -v $HOME/docker/volumes/
grafana-data:/var/lib/grafana grafana/k6
```

Explanation of options or flags:

`-d`	Detached mode runs the container in the background, allowing you to continue using your terminal or shell for other tasks.
`--rm`	Specifies that the container should be automatically removed when it stops running. It is useful when you want to clean up the container and its associated resources after it completes its execution.
`--name`	Sets the name of the container
`-p`	Maps the host’s port to the container’s port [host-port:container-port]
`-v`	Mounts the host’s volume or directory to the container’s [host-volume:container-volume]

However, received the following error and will have to troubleshoot.

```
WARNING: The requested image's platform (linux/amd64) does not match the
detected host platform (linux/arm64/v8) and no specific platform was
requested
```

The host platform is a 2023 MacbookPro Apple M2 chip computer, which may be causing the error. Further research is ongoing.

✓ Complete daily journal entry

Day 4 (25 May 2023)

Continue troubleshooting k6 container on a MBP with Apple M2 chip. Several posts on stack overflow suggest adding `--platform linux/amd64` flag to the docker run command may work because it does on Apple M1 chip computers, but it did not work. Another option is to install k6 via homebrew. In the mean time, the following results were shared within my team to answer today’s assignment.

- What is being measured?

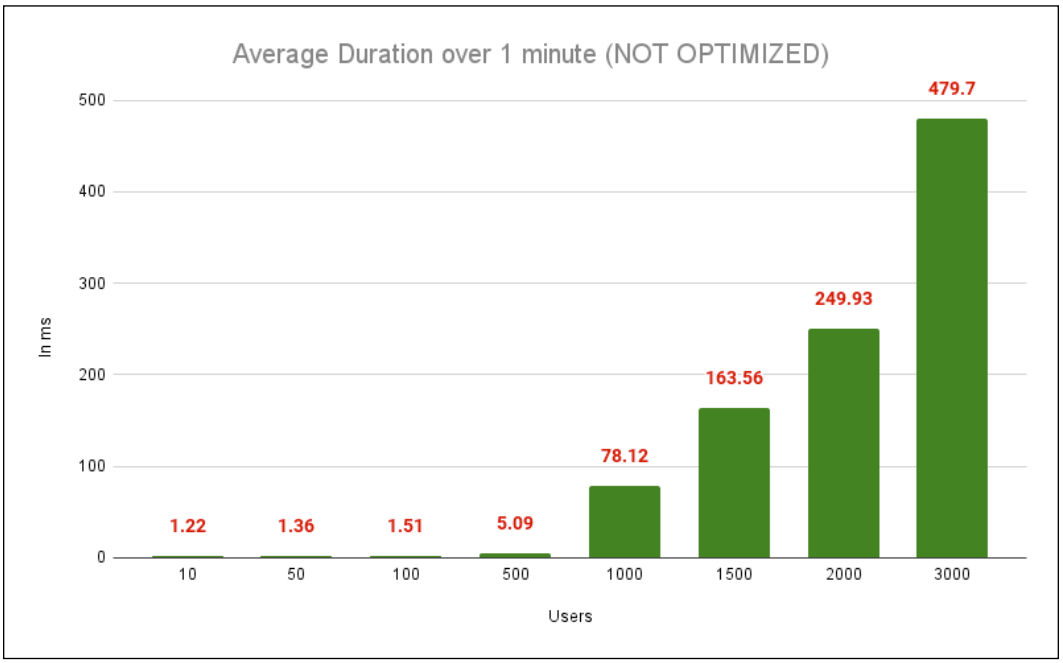
  Response time or latency in logarithms of milliseconds by the amount of users, in this case 10, 50, 100, and so on.

- How is being measured?

  According to k6 documentation, it measures the data by capturing timestamps and performance-related information; response time, throughput, request count, errors, virtual users, and custom metrics via k6 API.

- What does it means?

  For our initial data, the time complexity remained relatively constant until 500 virtual users, then it quickly increased to quadratic O(n^2) time complexity.



✓ Complete daily journal entry

Day 5 (25 May 2023)

Installed k6 using homebrew and tested it with the default script. Researched k6 terminology on all the available metrics and was they could be used to perform load testing on the API.

Consulted with the client about possible horizontal scaling, and started looking into Nginx as a possible load balancer solution.

Completed a few more preliminary k6 tests and realized the script needs to be modified to test the API end-points instead of the front-end.

Draft k6 script against API end-points:

```
import http from 'k6/http';
import { check, sleep } from 'k6';

export const options = {

  vus: 100, // Maximum number of virtual users (VUs)
  duration: '45s', // Total test duration

};

export default function () {
  const baseURL = 'http://192.168.1.167';
  const listingId = 1;

  const endpoints = [
    `${baseURL}:3050/api/title/${listingId}`,
    `${baseURL}:3010/api/gallery/photo_url/${listingId}`,
    `${baseURL}:3003/api/about/${listingId}`,
    `${baseURL}:3002/api/amenities/${listingId}`,
    `${baseURL}:3002/api/amenities/ten/${listingId}`,
    `${baseURL}:4001/api/users/${listingId}`,
    `${baseURL}:4001/api/hosts/about/${listingId}`,
    `${baseURL}:4001/api/hosts/photo/${listingId}`,
    `${baseURL}:3005/api/reviews/${listingId}`,
    `${baseURL}:4000/api/location/${listingId}`,
    `${baseURL}:4000/api/location/description/${listingId}`,
  ];

  const headers = {
    'Accept': 'application/json',
    'Content-Type': 'application/json',
  };

  for (const endpoint of endpoints) {
    const res = http.get(endpoint, { headers });

    check(res, {
      'Status 200': (r) => r.status === 200,
    });

    sleep(1);
  }
};
```

✓ Complete daily journal entry

Day 6 (26 May 2023)

Researched ways to self-host the Chairbnb web-app to run load tests over internet instead of on premises. Decided to setup a Cloudflare tunnel, a free service that establishes a secure connection on the hosting system with Cloudflare infrastructure, and once configured the public-facing URL routes to them first before is routed through the encrypted tunnel back to the hosting system. All without firewall modifications.

Finished setting up the first self-hosted server, hard-wired on the local network, using a relatively inexpensive single-board computer system with the following specifications:

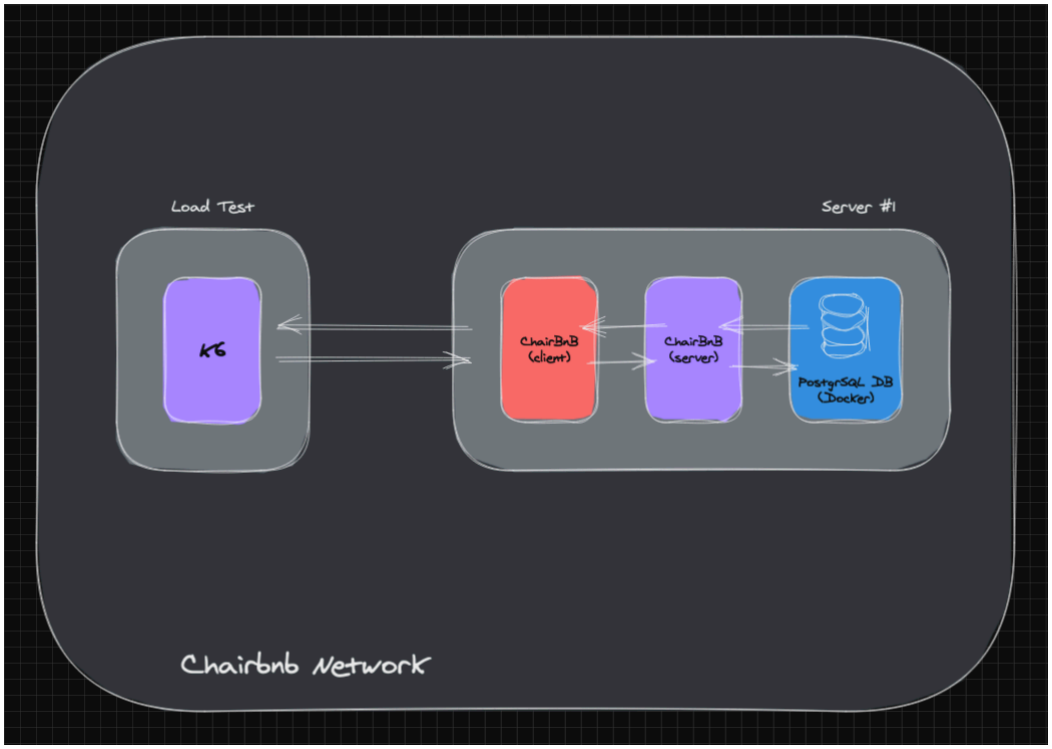
```
CPU: Quad core Cortex-A72 (ARM v8) 64-bit 1.8GHz
RAM: 8GB LPDDR4-3200 SDRAM
Storage: 64GB, NIC: Gigabit Ethernet
Power supply: 5V DC 2.5A
```

✓ Complete daily journal entry

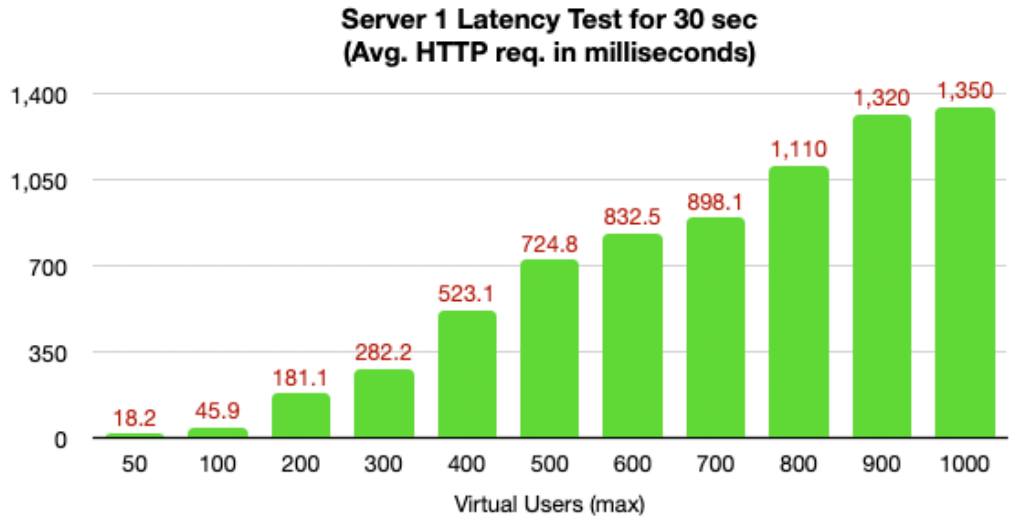
Day 7 (29 May 2023)

Finished deploying the web-app on ‘Server 1’ and drafted an architecture diagram to better understand the application and strategize ways to optimize it.

Also started looking at procuring (repurposing) a few more systems to test possible horizontal scaling of the web-app using Nginx as a load balancer.



The diagram above depicts all three elements of Chairbnb hosted on the first server. The components within the application were built as micro services, each with their own server, but accessing a single database which works fine for relatively low traffic.



Initial k6 test results for Server 1 indicated an average virtual users load of 800 to stay close to 1 sec. response time per http request for 90% of the requests. The test was conducted using the default k6 executor ‘ramping-vus’.

```
...

export const options = {

  vus: 50, // Maximum number of VUs
  duration: '30s', // Total test duration

};

...
```

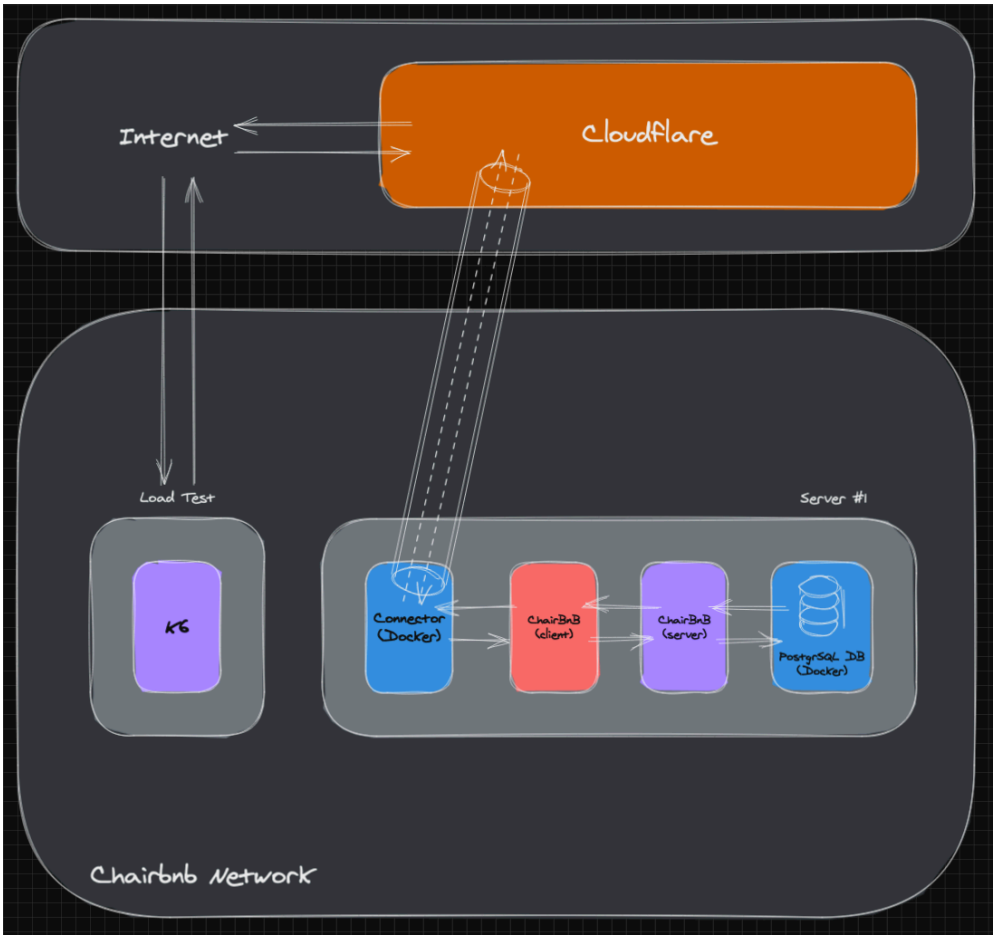
✓ Complete daily journal entry

Day 8 (30 May 2023)

Started building Server 2 and researching Nginx configuration to serve as a load balancer. Server 2 will have the same specifications as Server 1.

Both Sever 1 and Server 2 were baselined to the same operating system:

```
Debian-based v6.1.21 (Bullseye) 64-bit ’headless’ OS
```



This diagram depicts the network architecture proposed to the client leveraging the Cloudflare tunnel on a single server configuration. The architecture was tested with k6 as well, but it was challenging to achieve latency under 500ms per http request with loads higher than 500 VUs.

✓ Complete daily journal entry

Day 9 (31 May 2023)

Finished building Server 2 and the Nginx server. Researched and troubleshooted load balancing configurations. The Nginx server specifications are the same as the other two servers but with 4GB instead of 8.

/etc/nginx/nginx.conf

```
user www-data;
worker_processes auto;
pid /run/nginx.pid;
include /etc/nginx/modules-enabled/*.conf;

events {
        worker_connections 5000;
```

```
        # worker_connections 768;
        # multi_accept on;
}

http {
        upstream chairbnb {
                least_conn;
                # server demo1.tech-n-code.com;
                server 192.168.1.167:5173;
                # server demo2.tech-n-code.com;
                server 192.168.1.111:5173;
        }

        upstream chairbnb-title {
                least_conn;
                server 192.168.1.167:3050;
                server 192.168.1.111:3050;
        }

        upstream chairbnb-gallery {
                least_conn;
                server 192.168.1.167:3010;
                server 192.168.1.111:3010;
        }

         upstream chairbnb-about {
                least_conn;
                server 192.168.1.167:3003;
                server 192.168.1.111:3003;
        }

        upstream chairbnb-amenities {
                least_conn;
                server 192.168.1.167:3002;
                server 192.168.1.111:3002;
        }

        upstream chairbnb-users-hosts {
                least_conn;
                server 192.168.1.167:4001;
                server 192.168.1.111:4001;
        }

        upstream chairbnb-reviews {
                least_conn;
                server 192.168.1.167:3005;
                server 192.168.1.111:3005;
        }

        upstream chairbnb-location {
                least_conn;
                server 192.168.1.167:4000;
                server 192.168.1.111:4000;
        }

         server {
                listen 80;
                server_name demo3.tech-n-code.com;
```

```
        location / {
                proxy_set_header Host $host;
                proxy_pass http://chairbnb;
        }
}

server {
        listen 80;
        server_name 192.168.1.238;

        location /nginx_status {
                stub_status on;
                access_log off;
                allow 192.168.1.132;
                deny all;
        }

        location / {
                proxy_set_header Host $host;
                proxy_pass http://chairbnb;
        }
}

 server {
        listen 8080;
        server_name 192.168.1.238;

        location /api/title/ {
                proxy_set_header Host $host;
                proxy_pass http://chairbnb-title;
        }

        location /api/gallery/photo_url/ {
                proxy_set_header Host $host;
                proxy_pass http://chairbnb-gallery;
        }

        location /api/about/ {
                proxy_set_header Host $host;
                proxy_pass http://chairbnb-about;
        }

        location /api/amenities/ {
                proxy_set_header Host $host;
                proxy_pass http://chairbnb-amenities;
        }

        location /api/amenities/ten/ {
                proxy_set_header Host $host;
                proxy_pass http://chairbnb-amenities;
        }

        location /api/users/ {
                proxy_set_header Host $host;
                proxy_pass http://chairbnb-users-hosts;
        }

          location /api/hosts/about/ {
```

```
                proxy_set_header Host $host;
                proxy_pass http://chairbnb-users-hosts;
        }

        location /api/hosts/photo/ {
                proxy_set_header Host $host;
                proxy_pass http://chairbnb-users-hosts;
        }

        location /api/reviews/ {
                proxy_set_header Host $host;
                proxy_pass http://chairbnb-reviews;
        }

        location /api/location/ {
                proxy_set_header Host $host;
                proxy_pass http://chairbnb-location;
        }

        location /api/location/description/ {
                proxy_set_header Host $host;
                proxy_pass http://chairbnb-location;
        }

        location / {
                proxy_set_header Host $host;
                proxy_pass http://chairbnb;
        }

}

 server {
        listen 80;
        server_name 192.168.1.238;

        location / {
                proxy_set_header Host $host;
                proxy_pass http://chairbnb;
        }

}

##
# Basic Settings
##

sendfile on;
tcp_nopush on;
types_hash_max_size 2048;
# server_tokens off;

# server_names_hash_bucket_size 64;
# server_name_in_redirect off;

include /etc/nginx/mime.types;
default_type application/octet-stream;

##
```

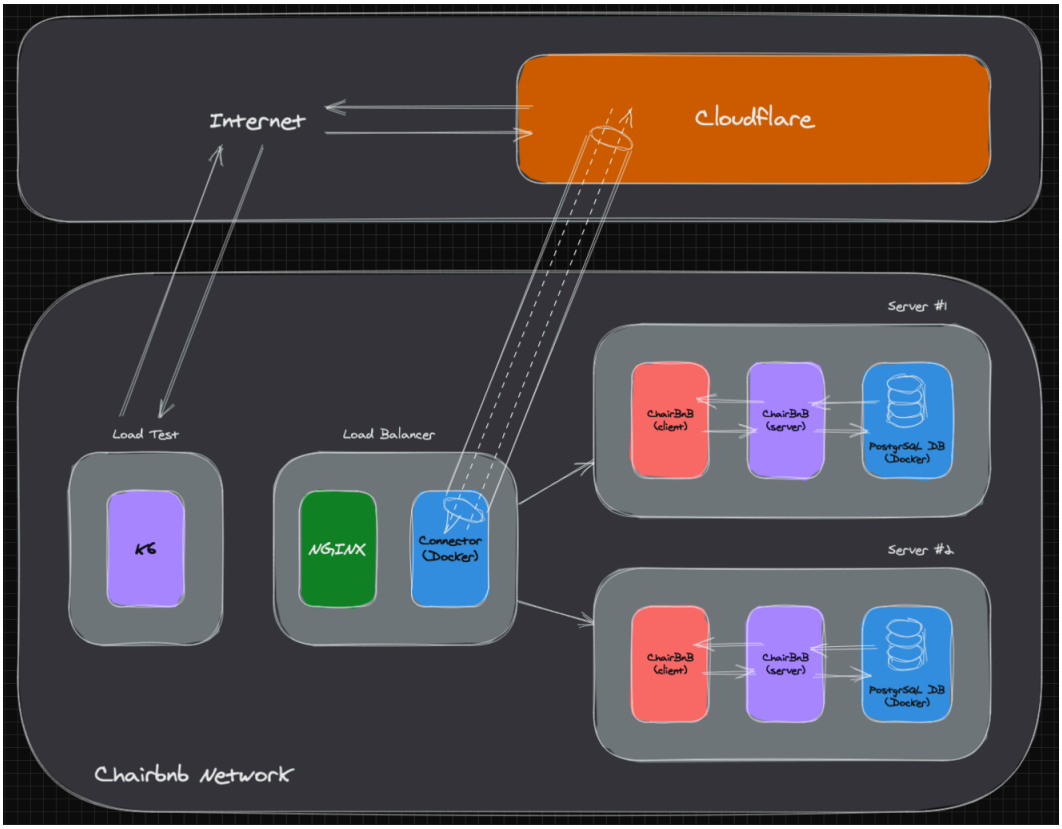
```
        # SSL Settings
        ##

        ssl_protocols TLSv1 TLSv1.1 TLSv1.2 TLSv1.3; # Dropping SSLv3, ref:
POODLE
        ssl_prefer_server_ciphers on;

         ##
        # Logging Settings
        ##

        access_log /var/log/nginx/access.log;
        error_log /var/log/nginx/error.log;

        ##
        # Gzip Settings
        ##

        gzip on;

        # gzip_vary on;
        # gzip_proxied any;
        # gzip_comp_level 6;
        # gzip_buffers 16 8k;
        # gzip_http_version 1.1;
        # gzip_types text/plain text/css application/json application/
javascript text/xml application/xml application/xml+rss text/>

        ##
        # Virtual Host Configs
        ##

        include /etc/nginx/conf.d/*.conf;
        include /etc/nginx/sites-enabled/*;
}
```

✓ Complete daily journal entry

Day 10 (1 Jun 2023)

As a proof of concept, implemented and tested Nginx load balancing between Server 1 and Server 2 with traffic routed through the Cloudflare tunnel to Chairbnb's font-end.

However, because tunnel routes are configured against one IP and one port, to route traffic to the each of the eleven (11) API endpoints would have taken a significant amount of time re-configuring and troubleshooting between tunnels, the load balancer, and the k6 script. For the sake of time, decide to finalize the architecture and focus on load testing before and after the introduction of an additional server and the Nginx load balancer.



Additional setbacks were encountered implementing Prometheus metrics monitoring and setting up Grafana for its comprehensive dashboard capabilities. Prometheus container was installed on the Nginx server, and corresponding Node-extension containers were installed on Server 1 & 2. Grafana container was installed on the testing computer hosting k6. Troubleshooted reports and graphs most of the afternoon.

✓ Complete daily journal entry

Day 12 (2 Jun 2023)

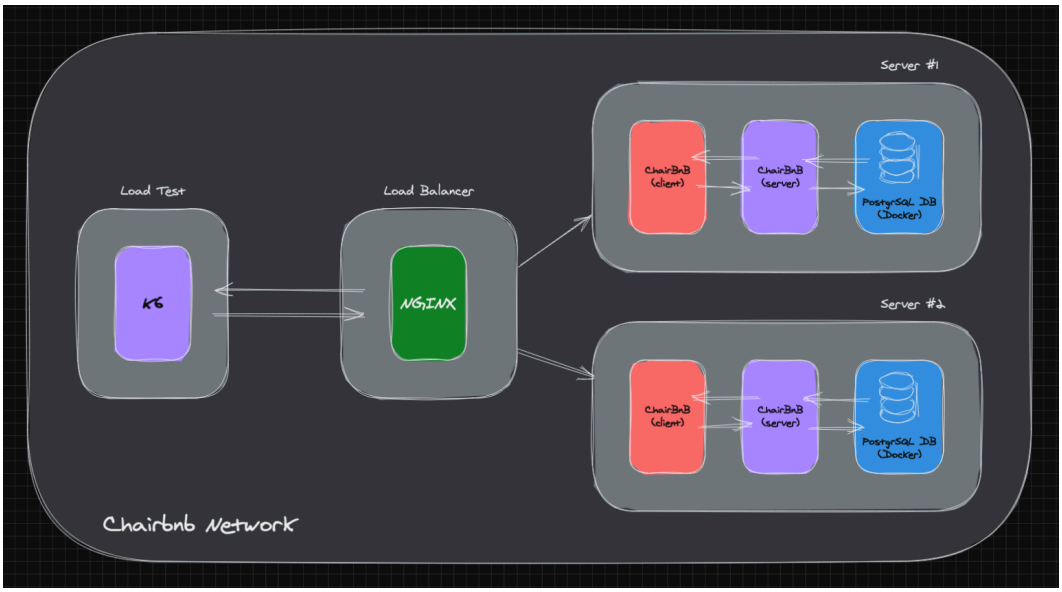
Finished troubleshooting Prometheus with its basic configuration. Realized the need to also implement Prometheus client for Node.js (prom-client) in order to get meaningful metrics from the API endpoints. The prom-client library would have to be installed in the project, modules imported, along with coding logic within each endpoint in order to generate metrics such as event loop lag and http req. stats and counters at the component level. Amazing capability, but out of scope of this project due to time constrains.

Prometheus basic configuration would not track http requests handled by each server, which was needed to evaluate how many http requests each API server was handling

and how effective was Nginx at load balancing utilizing the ‘least connections’ algorithm.

Decided against continuing to pursue Prometheus and shifted to implementing Influxdb for k6 in order to display k6 results metrics in Grafana and at least gain some experience with it. Then encountered another setback after implementing Influxdb v2.0 requiring the xk6-output-influxdb extension, forcing me to custom build a k6 binary with the required extension that never really worked with Influxdb v2.0. It was not until downgrading to Influxdb v1.8 that I was able reconfigure k6 commands to send results to the database. Even then, after configuring Influxdb as a data source on Grafana, the metrics were no more telling than the standard k6 report on the terminal.

Narrowed down the technologies that will be kept for the final round of testing and report writing; namely Nginx, and vanilla k6 over the local network, as depicted.



✓ **Complete daily journal entry**

Analysis of Results

I set out to answer the following questions for the client:

1. What are the API endpoints' performance under the current configuration?
2. What are some ways key metrics such as latency, throughput, and availability, can be improved?

The methodology for the tests was relatively simple; gather baseline metrics on Server 1 and Server 2 independent from each other, then deploy the Nginx load balancer and test again, analyze the results, and provide recommendations.

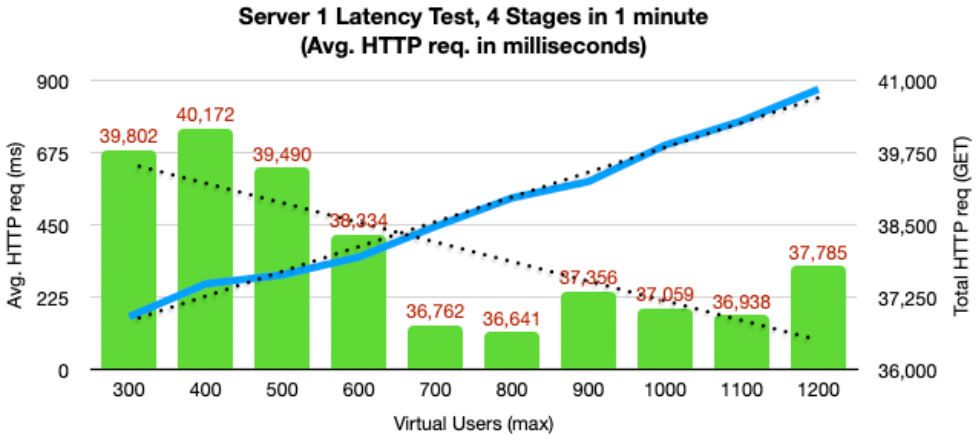
After extensive experimentation with k6, the following options were finalized and used on all three phases of the test (Server 1, Server 2, and load balanced configuration with Nginx). The aim is to 'stress test' the system(s) to help analyze how the each performs when load exceeds the expected average (est. at 300 VUs). Ramping up in stages allows for a more realistic settings, where the system(s) warms up before the max VUs load, sustains the increase load for 30 seconds, and then ramps down for 10 seconds.

```
...

export const options = {

  stages: [
    { duration: '10s', target: 100 }, // Stage 1: 100 VUs for 10s
    { duration: '30s', target: 300 }, // Stage 2: Ramp-up to x VUs over 30s
    { duration: '10s', target: 100 }, // Stage 3: 100 VUs for 10s
    { duration: '10s', target: 0 }, // Stage 4: Ramp-down to 0 VUs over 10s
  ],

  thresholds: {
    http_req_duration: ['p(90) < 500'], // Latency -- Can 90% of requests
take less than 500ms?
    'http_reqs{method: "GET"}': ['rate >= 600'], // Throughput -- Can it
handle 600 RPS or more?
    http_req_failed: ['rate < 0.5'], // Availability (error rate) -- 99.5% of
time available (error rate less than 0.5%)
  },

};

...
```

Answer to question #1 (current performance):

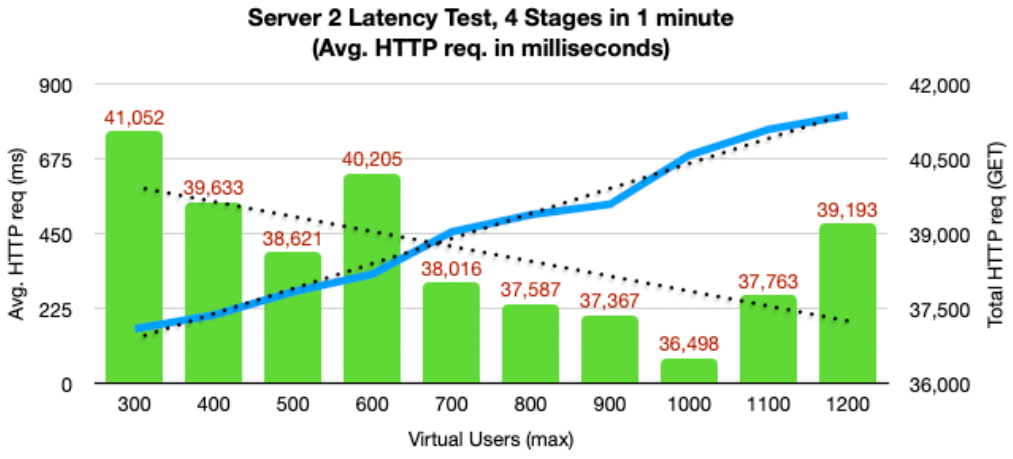
Both Server 1 and Server 1 results were relatively close to each other. Server 1 averaged 633.9 http requests per second (RPS) while Server 2 averaged 643.2 RPS. Latency under 500ms per http request was also close on both servers, able to achieve this in average for 700-800 VUs. However, even though they had the same hardware and software configuration, Server 1 ran hotter than Server 2 with a difference in

temperature of 3°F. It is possible that Server 2 had better airflow within the rack enclosure than Server 1, but as stated before, their performance was similar. In summary, they were able to easily handle the estimated normal load of 300 VUs with latency under 200ms per http request, but struggled with more than 800 VUs, maxing their throughput around 650 RPS.



Server 1 Latency Test, 4 Stages in 1 minute
(Avg. HTTP req. in milliseconds)

vus	vus_max	http_req_duration (avg.)	http_reqs (total GET reqs)	http_req /sec (RPS)	http_req_failed	Avg. CPU usage (htop)	Peak CPU Temp °F (htop)
5	300	163.6	39,802	663.4	0	72	104
6	400	266.21	40,172	669.5	0	68	104
7	500	294.2	39,490	658.2	0	71	103
6	600	350.0	38,334	638.9	0	67	104
6	700	445.2	36,762	612.7	0	71	105
6	800	534.3	36,641	610.7	0	70	104
7	900	585.3	37,356	622.6	0	70	105
6	1000	698.9	37,059	617.7	0	69	104
9	1100	774.8	36,938	615.6	0	70	104
7	1200	873.2	37,785	629.8	0	68	105
		Averages =>	38,033.9	633.9		69.6	104.2

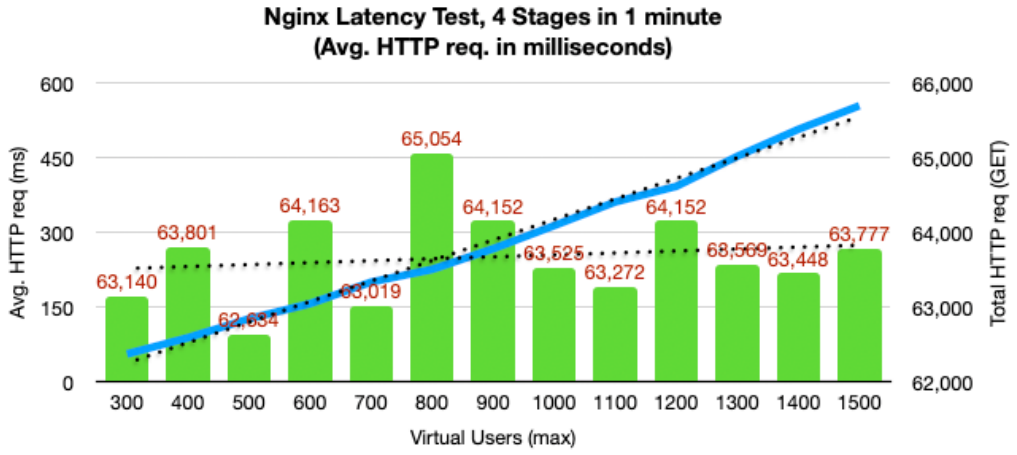


Server 2 Latency Test, 4 Stages in 1 minute
(Avg. HTTP req. in milliseconds)

vus	vus_max	http_req_duration (avg.)	http_reqs (total GET reqs)	http_req /sec (RPS)	http_req_failed	Avg. CPU usage (htop)	Peak CPU Temp °F (htop)
5	300	162.9	41,052	684.2	0	71	100
6	400	204.9	39,633	660.6	0	70	101
7	500	274.8	38,621	643.7	0	70	101
6	600	327.9	40,205	670.1	0	69	102
7	700	455.2	38,016	633.6	0	71	101
6	800	506.9	37,587	626.5	0	69	101
7	900	538.4	37,367	622.8	0	70	101
8	1000	686.2	36,498	608.3	0	69	100
7	1100	763.7	37,763	629.4	0	71	101
9	1200	806.4	39,193	653.2	0	69	102
		Averages =>	38,593.5	643.2		69.9	101

Answer to question #2 (ways to improve latency, throughput, and availability):

The last phase of testing against the Nginx load balancer revealed promising results considering horizontal scaling was limited to the addition of only one server. Load balancing increased the average throughput of the system to 1,062 RPS from 650 RPS utilizing one server (63% improvement), and increased the maximum amount of VUs from 700-800 to 1300-1400 (86% improvement) while maintaining average latency of http requests under 500ms, all without compromising availability as error rate remained zero across the board.



Nginx Latency Test, 4 Stages in 1 minute
(Avg. HTTP req. in milliseconds)

vus	vus_max	http_req_duration (avg.)	http_reqs (total GET reqs)	http_req /sec (RPS)	http_req_failed	Avg. CPU usage (htop)	Peak CPU Temp °F (htop)
5	300	55.1	63,140	1052.3	0	3.9	90
6	400	87.8	63,801	1063.4	0	3.4	91
5	500	125.9	62,634	1043.9	0	4.2	91
6	600	156.3	64,163	1069.4	0	3.8	90
7	700	199.6	63,019	1050.3	0	3.4	91
6	800	225.2	65,054	1084.2	0	4.1	92
7	900	266.7	64,152	1069.2	0	4.1	91
9	1000	313.2	63,525	1058.8	0	4.3	90
9	1100	360.7	63,272	1054.5	0	4.2	92
9	1200	392.1	64,152	1069.2	0	4.5	92
8	1300	452.6	63,569	1059.5	0	4.5	91
8	1400	506.8	63,448	1057.5	0	4.1	90
7	1500	554.2	63,777	1063.0	0	5.3	91
		Averages =>	63,669.7	1,061.2		4.1	90.9

Based on the results above, I encouraged the client to consider scaling horizontally to at least one more server. The ‘new normal’ with one load balancer between two low-power servers increased to 700 VUs (133% improvement), with a throughput of 1050 RPS (63% improvement), at the same baseline latency of 200ms and zero req. errors.

Depending on budget, scaling can be done in-house, repurposing systems already on hand, or on the cloud with one of many infrastructure as a service (IaaS) providers. For the particular architecture used throughout this project, self-hosted on low power single-board computers, the estimated 24/7 monthly cost is just over $2. This is based on an average of 4 watts per server and load balancer, and 14 watts for a dedicated network switch (non-POE). The estimate does not account for the procurement of infrastructure, vertical scaling (as needed), nor the technical expertise to maintain it, upgrade it over the long term, or perhaps plan an execute disaster recovery.

A rough cost estimate to run just one quad core 64-bit 1.8 GHz system with 8GB of RAM and 64GB for storage on IaaS is close to $30 a month (24/7) based on $0.0416 per hour for 730 hours (month est.), using 2021 pricing. On a two server and one load balancer configuration the monthly 27/4 cost just to keep the system running can be close to $100.

However, depending on the client’s optimization goals and budget, the IaaS benefits of quick scalability, geographic flexibility with multiple data centers, high availability/ uptime, and disaster recovery among others things can quickly outweigh in-house solutions.

In many cases, by offloading infrastructure management to IaaS providers, businesses can focus on their core competencies of running their business.

Lessons Learned

Connecting to a remote host using SSH-key

Entering passwords troubleshooting remote systems gets old really fast. The following workflow not only saves time but its also more secure than passwords alone.

- ☐ Generate an SSH-key running `ssh-keygen -t ed25519` or your favorite cipher. This will generate a private (no extension) and public key (<file-name>.pub). You can create as many as you want and even give them a passphrase to secure them even more.

- ☐ Add your private key to your machine's SSH agent by running `ssh-add ~/.ssh/<key-name>` on the terminal. Notice this is the private key (no extension). This allows your SSH session to use the key to authenticate with a remote hosts that has your public key. To list all keys loaded on your SSH agent run `ssh-add -l`

- ☐ Copy your public key to a remote host by running `ssh-copy-id -i ~/.ssh/<key-name>.pub <user>@<hostname or ip>` This will allow you to SSH to a host without asking for a password, unless you added a passphrase in which case you'll have to provide that each time.

The first time you SSH to a remote host your `known_hosts` file (usually in the `/.ssh` directory) will add fingerprint data for tracking purposes. If the IP or other identifying features such as MAC, etc. on the remote host change you will receive a warning and the connection will be refused. To reset this behavior, navigate to the `known_hosts` file and delete the entry for the remote host then try again.

VScode Terminal vs. regular system Terminal/CLI in the context of SSH

The lesson here is that they may have the same look and feel, but they are not the same, especially when it comes down to running commands in `sudo` mode installing packages, etc. Each shell (bash, zsh ,etc.) have its own configuration file with its own aliases and environment variables. It may be better to do installs, creating or moving files on the system's terminal/CLI, and use VScode terminal strictly for starting or stopping services. Just be cognizant of that if experiencing permissions issues.

Running services over SSH

If you close the SSH widow, the connection will close and also terminate any processes traced back to your session. This is generally well understood, but the following commands were not (at least to me):

☐ `nohup npm start &` //this will run the process in detached mode (`&`) and it won't terminate when closing the SSH connection (`nohup`)

☐ `nohup npm run dev &` //another example

Docker commands over SSH

Before SDC I might have setup a dockerized database at the start of a project, mostly via straight `docker run` command and its options, and a few times through dockerfile or docker-compose. But it was not until now that I was exposed to daily troubleshooting of containers over SSH. I find the workflows and commands below essential for junior developers exposed to containerized applications.

☐ Removing containers and their corresponding image over SSH:

- `docker ps -a` //lists all containers regardless if they are running
- `docker stop <container-id>` //stops the container
- `docker rm <container-id>` //removes the container
- `docker image ls` or `docker images` //lists all images
- `docker rmi <image-id>` //removed the image

☐ Troubleshooting Docker daemon over SSH (headless Debian):

- `sudo systemctl is-active docker` //returns 'active' or not to console
- `sudo systemctl start docker` //starts Docker daemon
- `sudo systemctl stop docker` //stops Docker daemon
- `sudo systemctl enable docker` //enable Docker daemon to auto start after rebooting the host system

Troubleshooting ports over SSH (Linux based)

On more than one occasion, because of human error, poor workflow or both, I found myself running services on top of already running ones. Some servers failed to run, but running `npm start` would restart them all. Since I was running commands detached mode I couldn't see if they were running as I normally would in console.

Normally restarting the system and trying again would take care of it. But the following commands saved me a lot of time and I think should be part of a full-stack developer's repertoire.

☐ `sudo ss -tuln` //lists all ports and the protocol they are listening to (UDP/TCP)

☐ `sudo fuser -n tcp <port>/tcp` //once you get the port, you can get the process id (PID) using using a particular port

- `sudo kill <PID>` //kill the process, or in my case, the random server still up causing port conflicts

- `sudo fuser -k tcp <port>/tcp` //you can also kill the process directly with the port number

- `sudo pkill -f <command>` //you can also kill the process using the command that starts it