

**Τμήμα Μηχανικών Η/Υ & Πληροφορικής**

Πανεπιστήμιο Πατρών

Δ

[Υλοποίηση του web server σε image της Docker 1](#_heading=)

[nginx web server 1](#_heading=)

[Docker 3](#_heading=)

[Artillery 6](#_heading=)

[Υλοποίηση load tester μέσω Artillery 6](#_heading=)

[Εκτέλεση τεστ για το container 8](#_heading=h.tdoyrh588xb5)

[Σχολιασμός και ζητήματα που προέκυψαν 8](#_heading=h.gjdgxs)

[Προσπάθειες για υλοποίηση του HLS 9](#_heading=)

[Kubernetes 10](#_heading=)

[Αρχική υλοποίηση μέσω minikube. 10](#_heading=)

[ΡΥΘΜΙΣΗ ΠΕΡΙΒΑΛΛΟΝΤΟΣ 10](#_heading=h.vwuyzcee16h)

[ΥΛΟΠΟΙΗΣΗ ΤΟΥ NGINX SERVER 11](#_heading=h.77m58ypvcqm3)

[Εκτέλεση tests για το cluster 13](#_heading=h.kk5g7iaxee44)

[Για 3 replica sets, 20 χρήστες 13](#_heading=h.mogej38etwy)

[Για 10 replica sets, 20 χρήστες 14](#_heading=h.gjdw8tjlb5x6)

[Για 3 replica sets, 200 χρήστες 17](#_heading=h.igpcqv1nguj7)

[Για 10 replica sets, 200 χρήστες 17](#_heading=h.qndf2pm8f5fz)

[**Prometheus operator 17**](#_heading=h.3j916kdntz22)

Έχουμε υλοποιήσει την εργασία μέσω git, στο repository : <https://github.com/miraq3bnkc/5G-Course-Project>

# Υλοποίηση του web server σε image της Docker

## nginx web server

Υλοποιήσαμε κατευθείαν τον nginx web server σαν ένα docker image, γιατί έτσι δεν θα υπήρχε η ανάγκη μεταφοράς της εφαρμογής σε docker image και η υλοποίηση είναι πιο απλή με το nginx:alpine image, που είναι κομμάτι των linux alpine images. Υλοποιήσαμε το nginx configuration με τον τρόπο που φαίνεται στο screenshot του κώδικα, ώστε να μπορεί να αποστείλει ένα mp4 αρχείο, και να παιχτεί στο innate video player του html5:

events {}

http {

     types {

        video/mp4 mp4;

    }

    server {

        listen 80;

        listen [::]:80;

        location / {

            root /usr/share/nginx/html;

            index video.mp4;

            mp4;

            mp4\_buffer\_size 1m;

            mp4\_max\_buffer\_size 8m;

            default\_type video/mp4;

            # kill cache

            add\_header Last-Modified $date\_gmt;

            add\_header Cache-Control 'no-store, no-cache';

            if\_modified\_since off;

            expires off;

            etag off;

        }

        location /stub\_status {

            stub\_status;

            allow all;

        }

    }

}

Χρησιμοποιήσαμε το ngx\_http\_mp4\_module, το οποίο προσφέρεται σαν για μετάδωση βίντεο στην δωρεάν έκδοση της nginx <http://nginx.org/en/docs/http/ngx_http_mp4_module.html>

Το mp4 module δίνει την δυνατότητα για pseudo streaming μετάδοση ενός mp4. Το βίντεο που θα μεταδίδει ο σέρβερ είναι ένα και συγκεκριμένο, και εισάγεται στο nginx image με βάση το dockerfile, στο οποίο θα αναφερθούμε αργότερα στο τμήμα που μιλάμε για το τι κάναμε στο docker. Έχουμε δώσει σε αυτό το όνομα “video”. Έχουμε θέσει ένα σαν MIME type που μπορεί να αποστείλει ο web server σε http request το video mp4. Θέτουμε το αρχικό και πάνω όριο του μεγέθους του buffer. Επιλέγουμε να μην χρησιμοποιούμε cache, ώστε κάθε φορά να έχουμε το ίδιο αποτέλεσμα στην εξυπηρέτηση χρηστών, για τους σκοπούς της δοκιμής του δικτύου. Ακούμε στο port 80 για ipv4 και ipv6. Το ipv6 το χρειάστηκε γιατί έπρεπε να προσθέσουμε την επιλογή [stub\_status](http://nginx.org/en/docs/http/ngx_http_stub_status_module.html), για να μπορούμε να συνδεθούμε στην σελιδα που εκτίθενται οι μετρικές και με ipv4 και με ipv6 έτσι ώστε να μπορεί να βλέπει o prometheus τα στοιχεία του nginx.[[nginx-prometheus-exporter]](https://github.com/nginxinc/nginx-prometheus-exporter)

## Docker

Το dockerfile οπού κατασκευάζει το image είναι το εξής:

# Use the nginx:alpine base image

FROM nginx:alpine

# Install curl for downloading NGINX Prometheus exporter binary

RUN apk add --no-cache curl

# Download NGINX Prometheus exporter binary and extract it

RUN curl -LO https://github.com/nginxinc/nginx-prometheus-exporter/releases/download/v1.1.0/nginx-prometheus-exporter\_1.1.0\_linux\_amd64.tar.gz \

    && tar -xzf nginx-prometheus-exporter\_1.1.0\_linux\_amd64.tar.gz \

    && mv nginx-prometheus-exporter /usr/local/bin/ \

    && rm nginx-prometheus-exporter\_1.1.0\_linux\_amd64.tar.gz

# Copy custom Nginx configuration file

COPY nginx.conf /etc/nginx/nginx.conf

# Copy the video file into the container (if necessary)

COPY \*.mp4 /usr/share/nginx/html/video.mp4

# Expose port 80 for accessing the video and port 9113 for accessing nginx metrics

EXPOSE 80 9113

# Start NGINX and NGINX Prometheus exporter

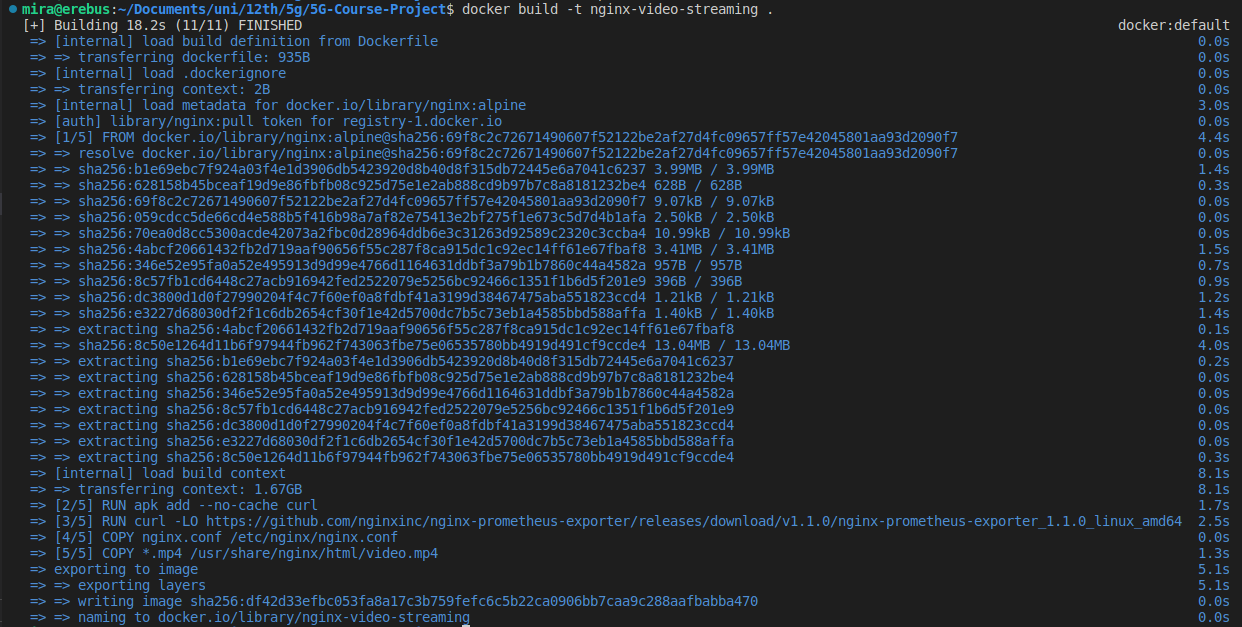
CMD ["sh", "-c", "nginx && nginx-prometheus-exporter --nginx.scrape-uri=http://localhost/stub\_status"]

**ΠΑΡΑΤΗΡΗΣΗ:**

Για την προσομοίωση της εργασίας μας είναι απαραίτητο να διαθέσετε ένα οποιοδήποτε αρχείο .mp4 μέσα στον φάκελο της εργασίας. Συγκεκριμένα, θα πρέπει το βίντεο αυτό να υπάρχει στο ίδιο directory με το Dockerfile. Δεν ανεβάσαμε το βίντεο που χρησιμοποιήσαμε ούτε στο github , ούτε στο docker hub , εμφωλευμένο μέσα στο image μας , για την προστασία προσωπικών δεδομένων και πνευματικής περιουσίας. Το βίντεο το οποίο χρησιμοποιούμε είναι περίπου 1.7GB και έχει διάρκεια περίπου

Για να φτιάξουμε το image εκτελούμε την εξής εντολή:

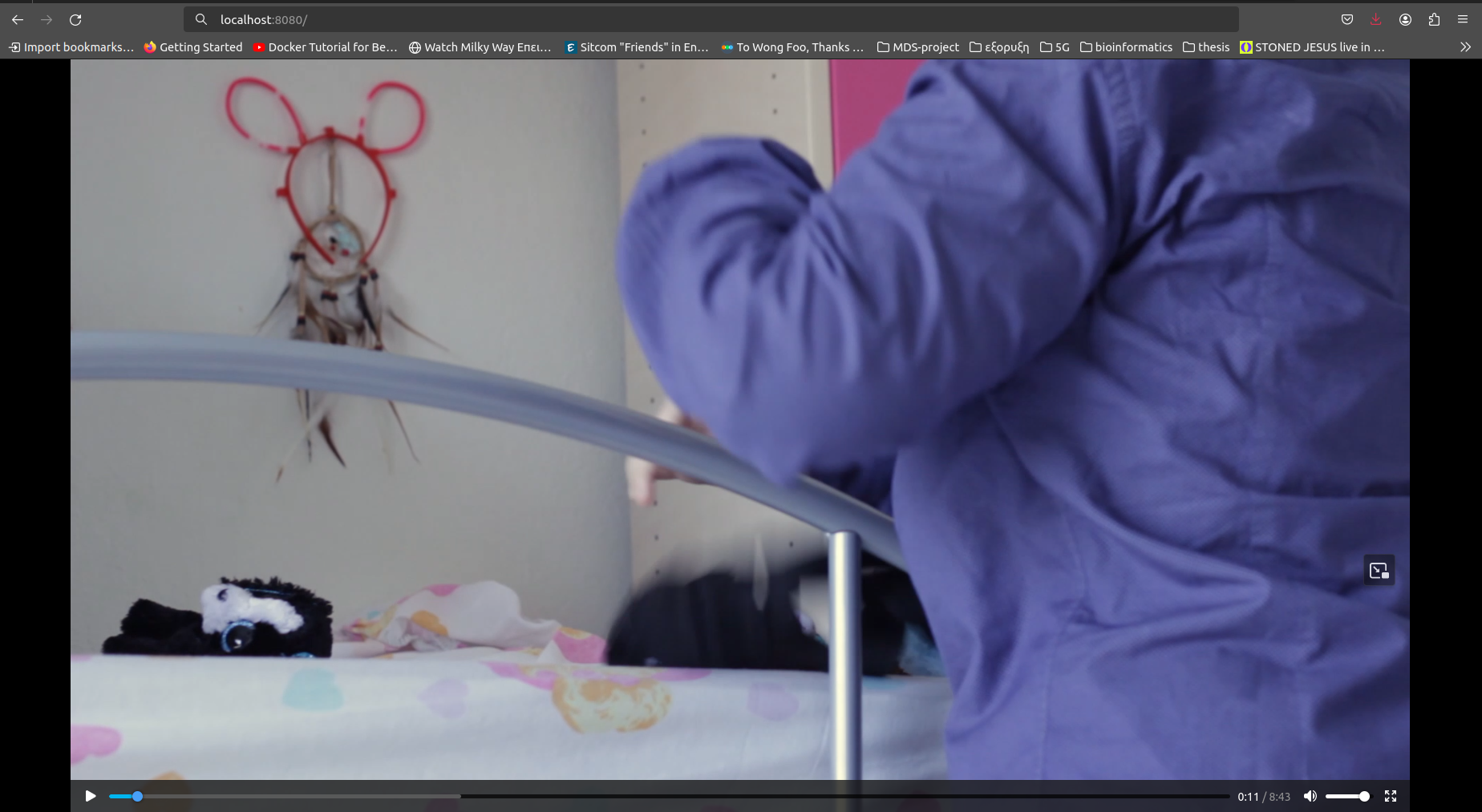
docker build -t nginx-video-streaming .



Για να φτιάξουμε το image εκτελούμε την εξής εντολή:

docker run -p 8080:80 -p 9113:9113 nginx-video-streaming

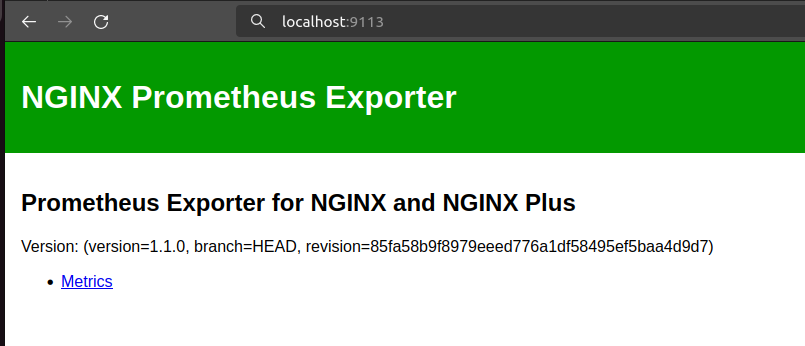
Τρέχοντας [http://localhost:8080](http://localhost:80) μπορούμε να δούμε το εξής αποτέλεσμα στην ιστοσελίδα:



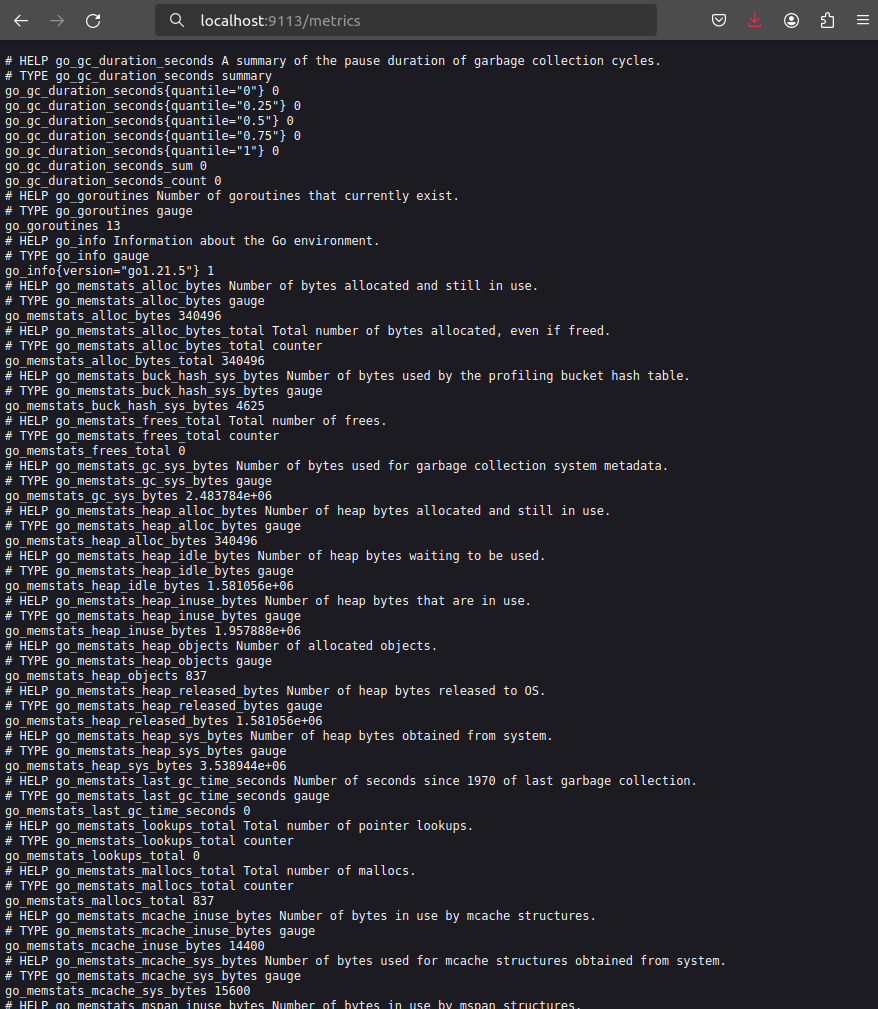
τυχαίο στιγμιότυπο του video που χρησιμοποιήσαμε , καθώς συνδεθήκαμε στον server μας.

Την θύρα 9113 που κάναμε expose βλέπουμε τα εξής:

<http://localhost:9113/> και <http://localhost:9113/metrics>



όπου συγκεκριμένα στο path “\metrics” βλέπουμε αναλυτικά όλες τις μετρικές που εκθέτει το nginx prometheus exporter



# Artillery

## Υλοποίηση load tester μέσω Artillery

Χρησιμοποιούμε το [Artillery](https://www.artillery.io/docs/get-started/get-artillery), πλατφόρμα για load testing βασισμένο στη node js, για να αναπαράγουμε ψηφιακούς χρήστες και να δοκιμάσουμε την δυνατότητα του σέρβερ να ανταποκρίνεται σε πολλά http request.

Εγκαταστήσαμε το Artillery στους υπολογιστές μας μέσω npm με την εντολή:

* npm install -g artillery@latest

VERSION INFO:

Artillery: 2.0.8-f0fd405

Node.js: v18.17.0

OS: linux

Δεν τρέχουμε το artillery σε container, το τρέχουμε από τον υπολογιστή και συνδεόμαστε μέσω του localhost IP. Γράφουμε σε ένα yaml αρχείο με όνομα nginx artillery testing το test script που το εκτελούμε γράφοντας στο terminal, στο φάκελο στο οποίο βρίσκεται το test script, την παρακάτω εντολή:

* artillery run -o nginx-pseudo-streaming-report.json nginx-artillery-testing.yaml

Το οποίο κρατάει τα αποτελέσματα στο json αρχείο nginx-pseudo-streaming-report. Για να αναπαραστήσουμε τα αποτελέσματα καλύτερα σε html σελίδα,χρησιμοποιούμε την εξής εντολή:

* artillery report nginx-pseudo-streaming-report.json

Πραγματοποιήσαμε πολλά load tests, με διαφορετικές, αλλά υπήρχαν ζητήματα που θα τα αναλύσουμε στην παρακάτω υπό-ενότητα. Το τελικό load tester που δοκιμάσαμε, που συμπυκνώνει διάφορα πράγματα που κάναμε, ήταν το παρακάτω:

config:

target: http://10.111.204.122/

http:

# Responses have to be sent within 30 seconds, or an `ETIMEDOUT` error gets raised.

timeout: 30

phases:

- duration: 60

arrivalCount: 200

name: Minimum

plugins:

ensure: {}

apdex: 2500

metrics-by-endpoint: {}

expect:

reportFailuresAsErrors: true

outputFormat: prettyError

scenarios:

- flow:

- loop:

- get:

url: '/'

headers:

Range: 'bytes=0-7500000' # Specify the byte range you want to request

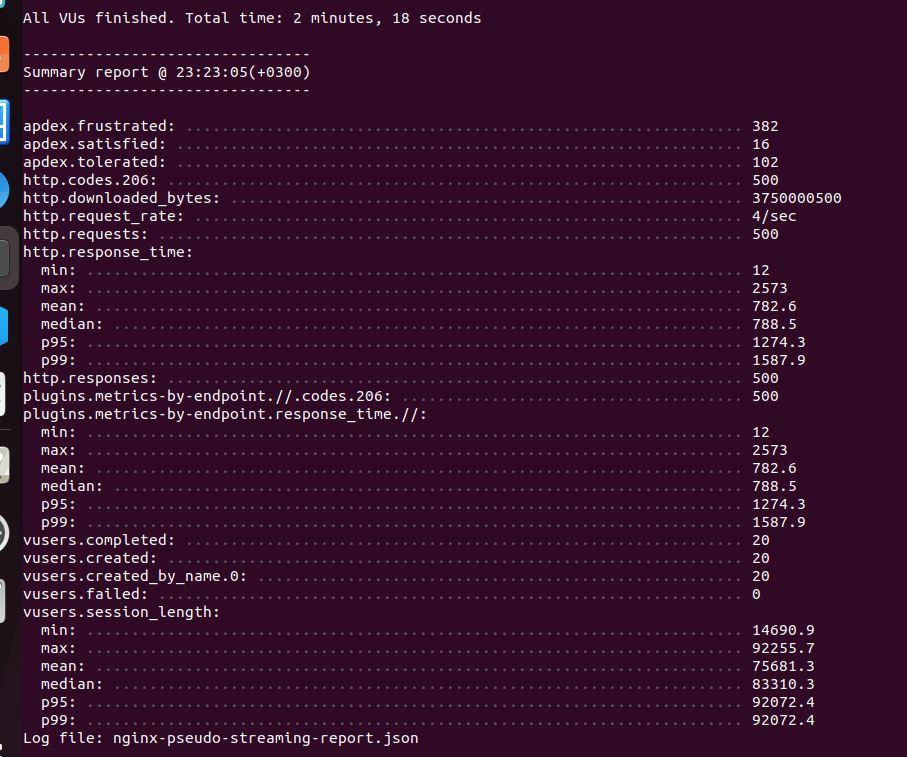
count: 25

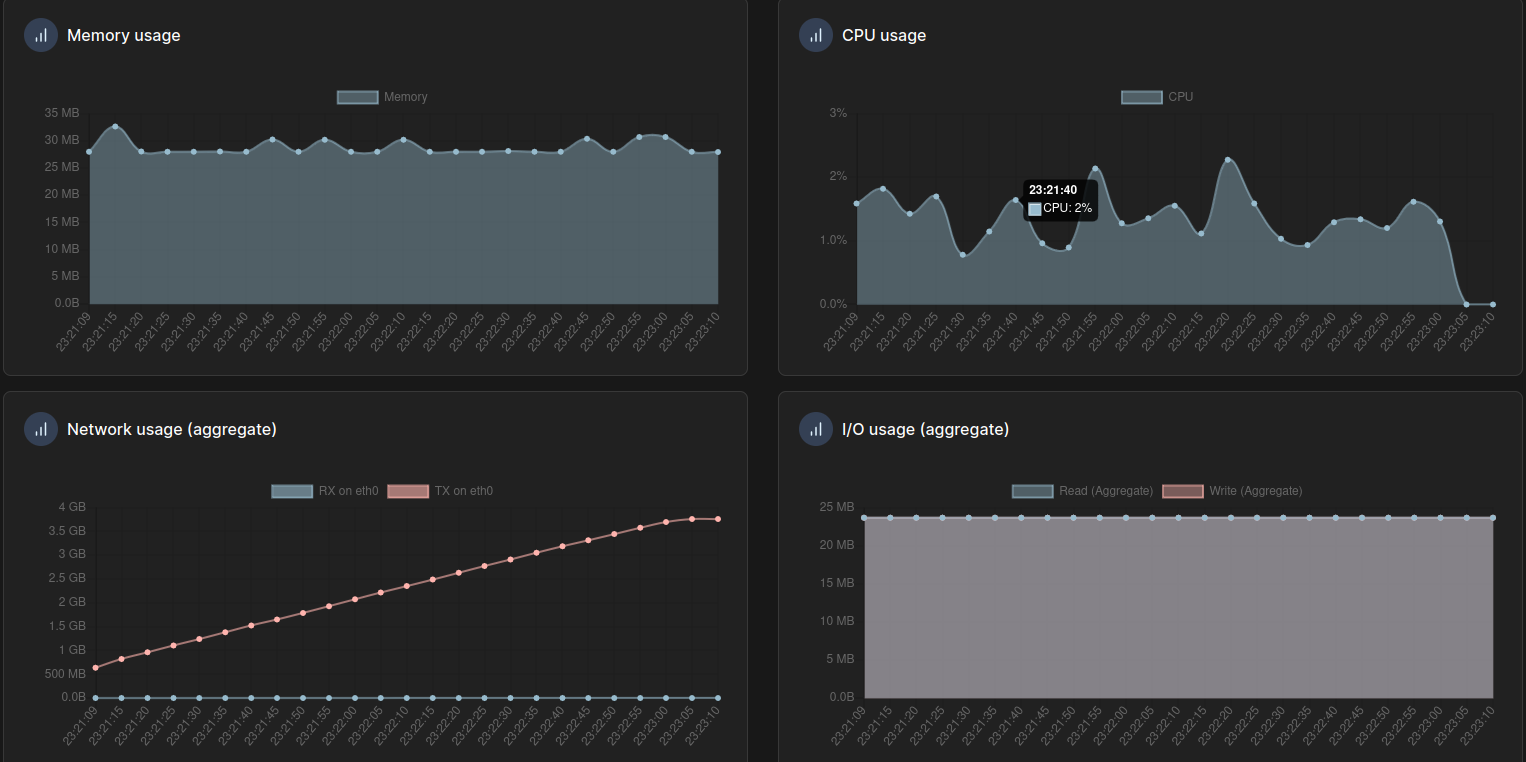
* Στο target έχουμε το Ip που θέλουμε να ελέγξουμε. Στην παραπάνω έκδοση του load tester, είναι το ip που παίρνουμε από το minikube tunnel εντολή, που τρέξαμε για να δημιουργήσει load balancer το minikube για το service που χρησιμοποιούμε για την υλοποίηση της εφαρμογής στο kubernetes. Σε απλό container, χρησιμοποιούσαμε το <http://localhost:80> ή το <http://127.0.0.1:80>. Στο minikube, παίρνουμε το external ip που δημιουργείται από το minikube tunnel, συγκεκριμένα το http://10.111.204.122/



* Στο phases έχουμε τον αριθμό των χρηστών που θα αναπαραχθούν και μέσα σε πόσο χρόνο θα δημιουργηθούν. Καθώς είδη είναι μεγάλο το βάρος που χρειάζεται να στείλει το request, ορίζουμε συγκεκριμένο αριθμό. Το ελέγχουμε με 20 χρήστες, και 200 χρήστες
* Θέτουμε το timeout του http request σε 30 seconds ωστέ να μην βγαίνουν ETIMEDOUT οι ψηφιακοί χρήστες
* Στο apdex, ορίζουμε σαν threshold 2500 milliseconds, που ορίζει αν ο χρήστης θα θεωρηθεί satisfied ή όχι.
* Το scenarios έχει τις ενέργειες που θα κάνουν οι ψηφιακοί χρήστες, που στην περίπτωσή μας ήταν απλά μία get request. Το έχουμε βάλει μέσα σε μία λούπα που εκτελείται 25 φορές, και κάθε get ζητάει 10 megabytes. Οπότε προσομοιώνουμε ο χρήστης να κατεβάσει 187,5 megabytes από το βίντεο.

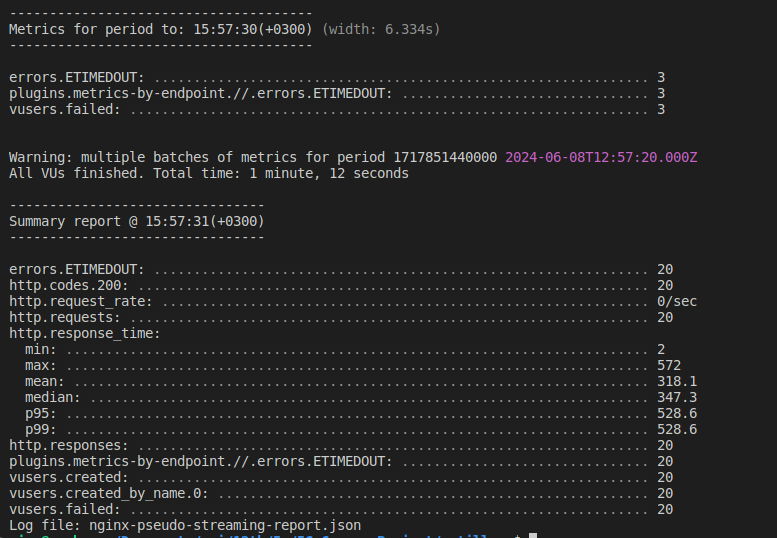
## Εκτέλεση τεστ για το container





## Σχολιασμός και ζητήματα που προέκυψαν

Το μεγάλο ζήτημα που προέκυψε ήταν ότι οι ψηφιακές χρήστες σε κάθε load test “πέταγαν” το ETIMEDOUT error , όπως βλέπουμε και στο παρακάτω summary report. Αυτό σημαίνει ότι το request δεν ικανοποιήθηκε σε ένα δοσμένο χρονικό διάστημα, οπότε το artillery σταματάει το request και πετάει το συγκεκριμένο error. Το default στο artillery είναι 10 δευτερόλεπτα και στο http είναι 30 δευτερόλεπτα.



Συγκεκριμένα , όταν δεν είχαμε Range Request στο artillery , ακόμα και σε load test με ένα χρήστη, μας έβγαζε αυτό το σφάλμα. Αυτο που παρατηρήσαμε και απο τα logs ήταν ότι ζητούνταν ολοκληρο το βίντεο να κατεβει όλο μαζί με ένα request. Παράλληλα για να προσομοιώσουμε μία μέση χρήση ενός web server που αποστέλλει βίντεο σε χρήστες, χρησιμοποιούσαμε βίντεο πάνω των 100-200 megabytes, το οποίο είναι ένα μέγεθος που μπορεί να έχει μία ταινία hd ή ένα μεγάλο βίντεο στο youtube.

Το θέμα είναι ότι επειδή το ελέγξαμε από το προσωπικό ασύρματο δίκτυο μας, έχουμε περιορισμό στο upload speed, και μαζί με άλλους λόγους, είχαν πολύ περιορισμένο download speed μεταξύ των χρηστών. Αυτό δεν είναι ζήτημα σε κανονική χρήση, επειδή το pseudo streaming λειτουργεί με progressive download που μπορείς να προχωρήσεις σε κομμάτι του βίντεο που δεν έχει κατέβει, με νέο http request. Αλλά από όσο φαίνεται, οι ψηφιακοί χρήστες του artillery κατεβάζουν όλο το αρχείο και θεωρούν σαν fail state αν δεν κατέβει σε συγκεκριμένο χρονικό διάστημα. Το ρωτήσαμε στο discussion. <https://github.com/artilleryio/artillery/issues/2701>

Ένας έλεγχος που έγινε είναι ότι συνδεθήκαμε στην διεύθυνση που “σερβίραμε” την υπηρεσία του nginx video streaming απο διαφορές συσκευές. Είδαμε πως και με 4 συσκευές παράλληλα δεν υπήρχε κανένα πρόβλημα όπως αυτό που εμφανιστηκε στο artillery load test ακόμα και για ένα χρήστη.

Σε αυτό το κομμάτι, είδαμε τι επιλογές δίνει το artillery, όπου εκεί βρήκαμε το plugin που δίνει την δυνατότητα να ελέγξει HTTP live streaming (HLS) <https://www.artillery.io/docs/reference/extensions/hls>. Αυτό θα έλυνε το πρόβλημα που είχαμε παραπάνω. Κάναμε μία αναζήτηση για μία υλοποίηση του hls, που δεν πέτυχε και περιγράφεται παρακάτω. Θεωρηθηκε μάλιστα πως ίσως έφταιγε το image μας, και πως έπρεπε να γίνουν αλλαγές στο nginx configuration, πριν συνειδητοποιήσουμε ότι δεν είχε κανένα πρόβλημα η εικόνα μας.

Για μία προσομοίωση ενός video streaming , χρησιμοποιήσαμε το header για το range, που περιορίζει τα δεδομένα που αποστέλλονται ανά χρήστη, σε 7.5 megabytes. Στο nginx configuration έχουμε max buffer size 8MB. Προσομοιώνουμε έτσι την αποστολή ενός segment του βίντεο κάθε φορά. Όμως, δεν είναι συνεχές, και αποστέλλει ένα μικρό μέγεθος από το βίντεο οπότε το βαλαμε να τρέχει σε επανάληψη στο σενάριο.

Αλλάξαμε το http timeout σε 30 seconds, ώστε να μην βγαίνουν οι χρήστες εκτός λόγω καθυστέρησης απάντησης του http request. Επίσης, αλλάξαμε το apdex να έχει μέγεθος 2500 milliseconds, καθώς το 300 milliseconds που έχει by default είναι αχρείαστα μικρό. Θέλουμε να έχει συνεχή εμπειρία παρακολούθησης χωρίς buffer, αλλά δεν μας νοιάζει να έρθει κατευθειαν.

## Προσπάθειες για υλοποίηση του HLS

Το nginx έχει δυνατότητα αποστολής βίντεο μέσω HLS και live streaming και on-demand. Το πρόβλημα είναι ότι αυτό είναι στην επι πληρωμή εκδοχή, που δεν είχαμε πρόσβαση. Για να υλοποιήσει κάποιος streaming στην nginx, μπορεί να κατεβάσει το [nginx rtmp module](https://github.com/arut/nginx-rtmp-module), το οποίο υλοποιεί το rtmp protocol, το οποίο μία από τις δυνατότητές του είναι να χρησιμοποιηθεί για hls on demand. Προσπαθήσαμε να το υλοποιήσουμε με αυτό τρόπο, αλλά δεν καταφέραμε να κατέβει σε image μορφή και να δουλέψει. Δοκιμάσαμε να κατεβάσουμε κάποια github project που είτε me ειδική εκδοχή για docker <https://github.com/tiangolo/nginx-rtmp-docker> είτε για video on demand <https://github.com/kaltura/nginx-vod-module> .

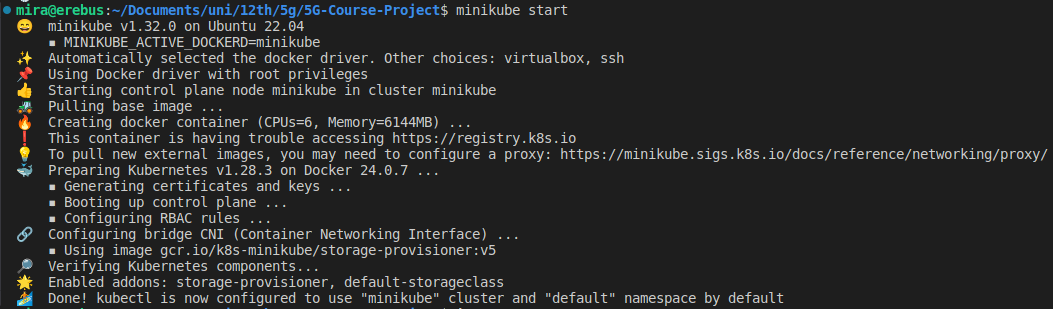
Αλλά σε κάθε περίπτωση, δεν κατάφερε να δουλέψει έτσι όπως το θέλαμε. Επειδή ο κώδικας σε κάθε βιβλιοθήκη ήταν πολύ περίπλοκος, δεν προσπαθήσαμε να υλοποιήσουμε κάποια αλλαγή. Μαλιστα για τα ζητούμενα της εργασίας μας το mp4 module ήταν αρκετό , καθώς δεν ζητούνταν να υλοποιηθεί live streaming service. Μπορείτε να δείτε μια μη επιτυχημένη προσπάθεια στο “test-branch1” στο github. Έτσι κρατήσαμε το vanilla artillery και το προσομοιώσουμε μέσω ενός συγκεκριμένου byte range για να αναπαραστήσει τα μέγεθος ενός segment και ένα αριθμό επαναλήψεων για να δείξουμε ότι ο κάθε χρήστης παρακολουθεί ένα βίντεο για ένα συγκεκριμένο χρονικό διάστημα. Έχουμε ορίσει το μέγεθος του κάθε segment να είναι 7.5 megabytes, που αντιστοιχεί σε 5 δευτερόλεπτα, ένα segment, και ευκρίνεια 1080p,10 mbps.

# Kubernetes

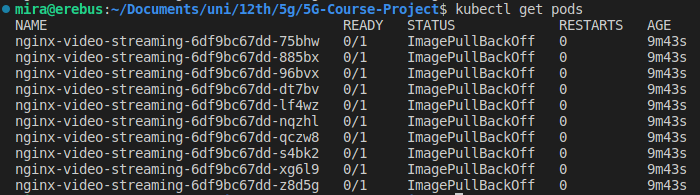
## Αρχική υλοποίηση μέσω minikube.

### ΡΥΘΜΙΣΗ ΠΕΡΙΒΑΛΛΟΝΤΟΣ

Με το minikube start , είχαμε θέσει αρχικά να αξιοποιεί 6GB από την RAM και 6 CPUs , οποτε πλέον με κάθε “minikube start” γίνεται χρήση αυτών των πόρων.



Επιπλέον κρίθηκε απαραίτητη η εκτέλεση της εντολής eval $(minikube -p minikube docker-env). Η συγκεκριμένη εντολή ρυθμίζει το περιβάλλον του κελύφους (shell) ώστε να χρησιμοποιεί τον Docker daemon μέσα στην εικονική μηχανή (VM) του Minikube. Χωρίς αυτή την ρύθμιση κατά το deployment των manifest για εφαρμογή του nginx μας είχαμε διαφορα προβλήματα στην εύρεση της εικόνας που είχαμε δημιουργήσει.



Για να λυθεί του σφαλμα “ImagePullBackOff” , χρειαζόταν να χτίσουμε την εικόνα και μέσα στην εικονική μηχανή του minikube (με τον ακριβώς ίδιο τρόπο που προείπαμε όταν εξηγήθηκε η δημιουργία του image) καθώς και να συμπεριλάβουμε αυτή την εντολή “imagePullPolicy: IfNotPresent” στο deployment αρχείο μας.

### ΥΛΟΠΟΙΗΣΗ ΤΟΥ NGINX SERVER

Υλοποιήσαμε το deployment στο να δημιουργεί 3 replica set και το service ωστέ να κάνει load balancer, το οποίο στο minikube γίνεται με minikube tunnel. Έχουμε ορίσει σαν σταθερό port το 80

Το deployment manifest (αρχείο deployment.yaml) που φτιάξαμε είναι το εξής:

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-video-streaming

labels:

app: nginx-video-streaming

spec:

replicas: 3 # Specify the number of replicas you want

selector:

matchLabels:

app: nginx-video-streaming

template:

metadata:

labels:

app: nginx-video-streaming

spec:

containers:

- name: nginx

imagePullPolicy: IfNotPresent

image: nginx-video-streaming # Docker image name

ports:

- containerPort: 80

Και το service manifest (αρχείο service.yaml):

apiVersion: v1

kind: Service

metadata:

name: nginx-video-streaming

spec:

selector:

app: nginx-video-streaming

ports:

- protocol: TCP

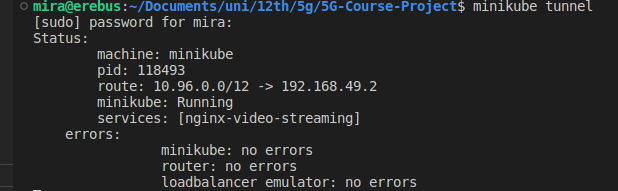
port: 80

targetPort: 80

type: LoadBalancer

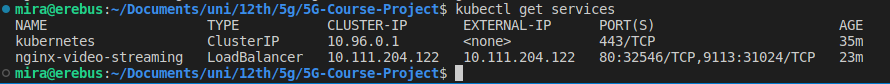
Για την εφαρμογή μας χρησιμοποιούμε υπηρεσία τύπου LoadBalancer , καθώς κατανέμει μία λειτουργίας απρόσκοπτα μεταξύ ενός αριθμού διακομιστών με σκοπό τη μείωση του φόρτου εργασίας.

Εκτελώντας “minikube tunnel” σε ξεχωριστό παράθυρο terminal εκθέτουμε μία IP διεύθυνση με την οποία μπορούμε να συνδεθούμε στην nginx video streaming υπηρεσία μας.



Εκτελώντας kubectl μπορούμε να δούμε ποια είναι η διεύθυνση αυτή , η οποία βρίσκεται κάτω από την στήλη external IP.

Άρα συνδεόμαστε στην διευθυνση: http://[ EXTERNAL IP]/ για να δούμε το βίντεο και http://[ EXTERNAL IP]:9113/metrics για το scraping των nginx μετρικών.



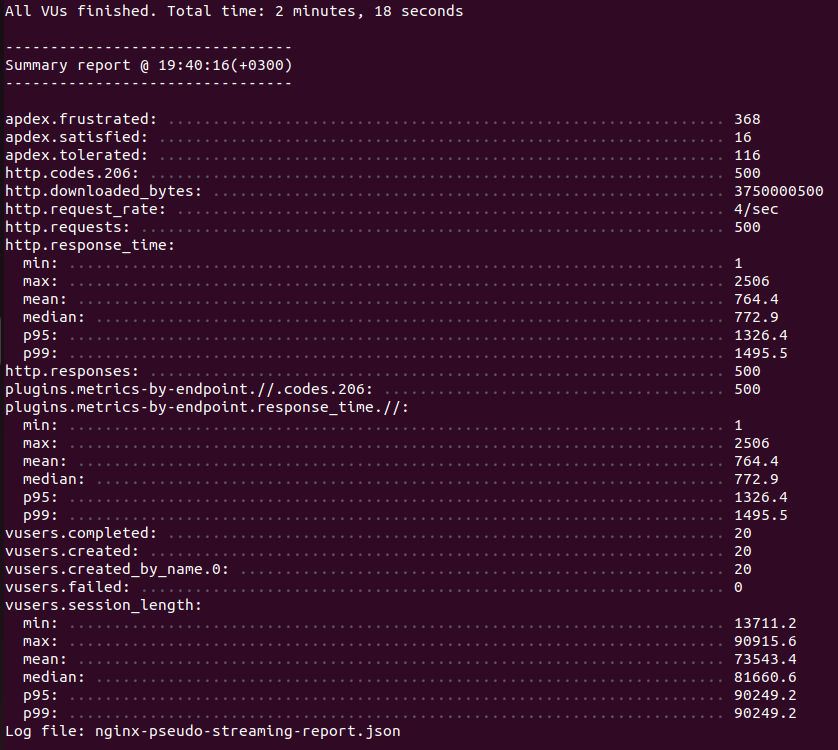
Χρησιμοποιούμε το minikube dashboard ui για να δούμε μετρικές εσωτερικές στον pod. Συγκεκριμένα, μέσω του metrics server μπορούμε να δούμε την χρήση του cpu και τις ram σε κάθε ξεχωριστό pod.

minikube addons enable metrics-server

## Εκτέλεση tests για το cluster

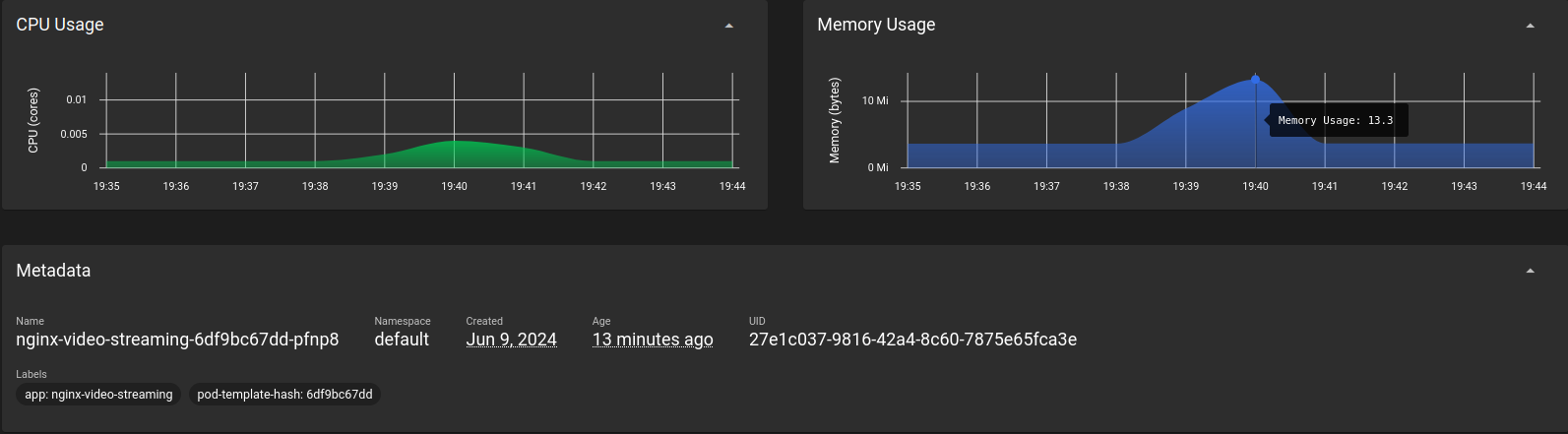
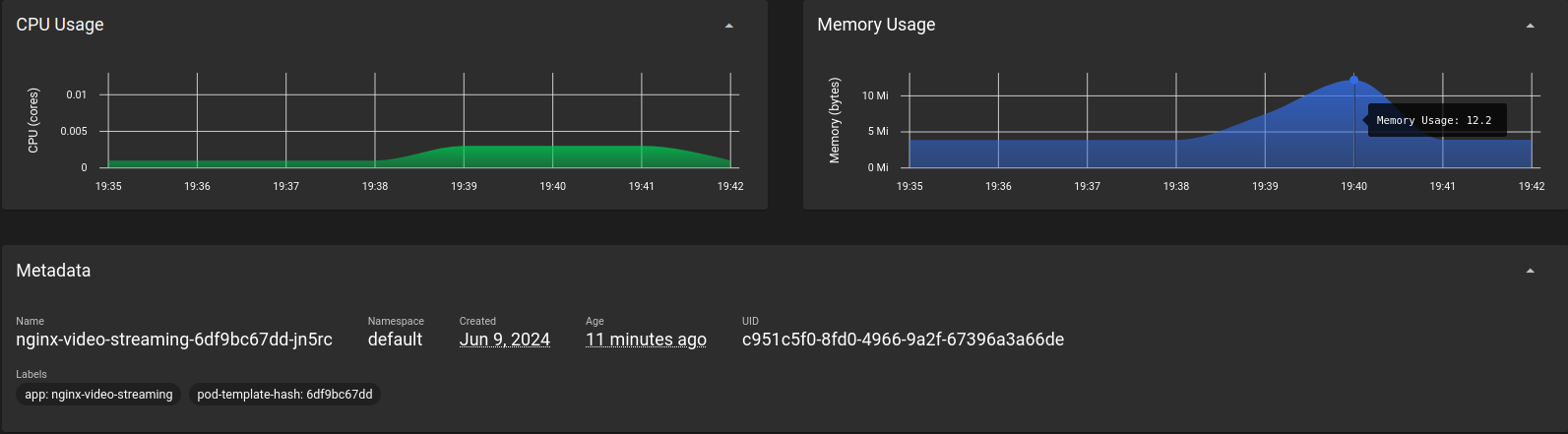
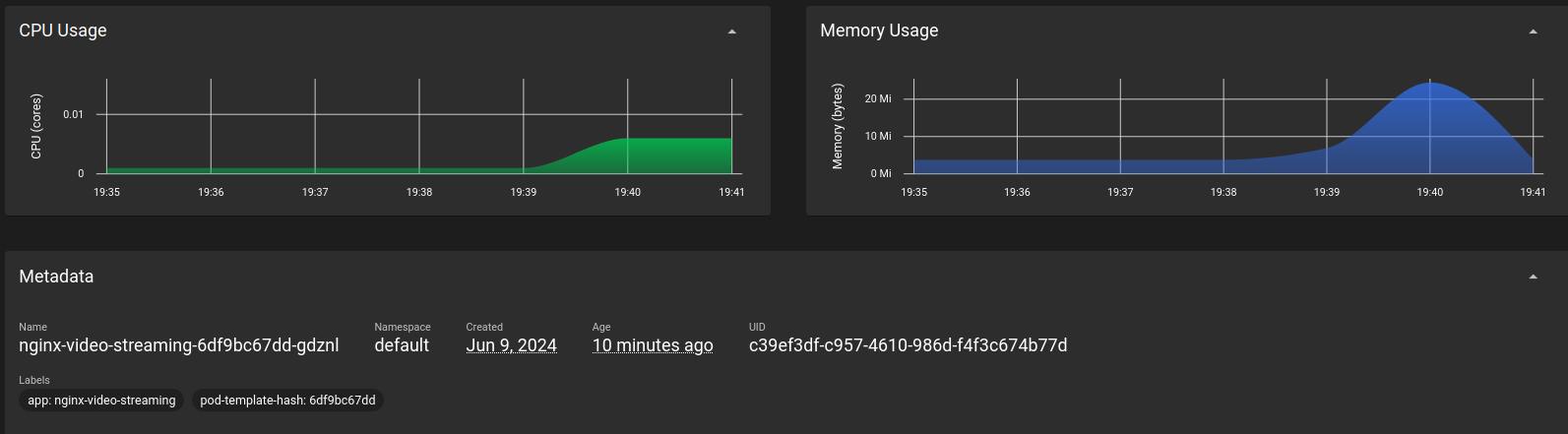
### Για 3 replica sets, 20 χρήστες

Έπειτα από εκτέλεση για το παραπάνω artillery script, έχουμε την παρακάτω αναφορά σε artillery:



Μπορούμε να δούμε ότι η χρήσης της cpu και της ram για όλα τα pod είναι η εξής, παρατηρώντας από το σημείο που ξεκινάει να χρησιμοποιείται μνήμη και cpu:

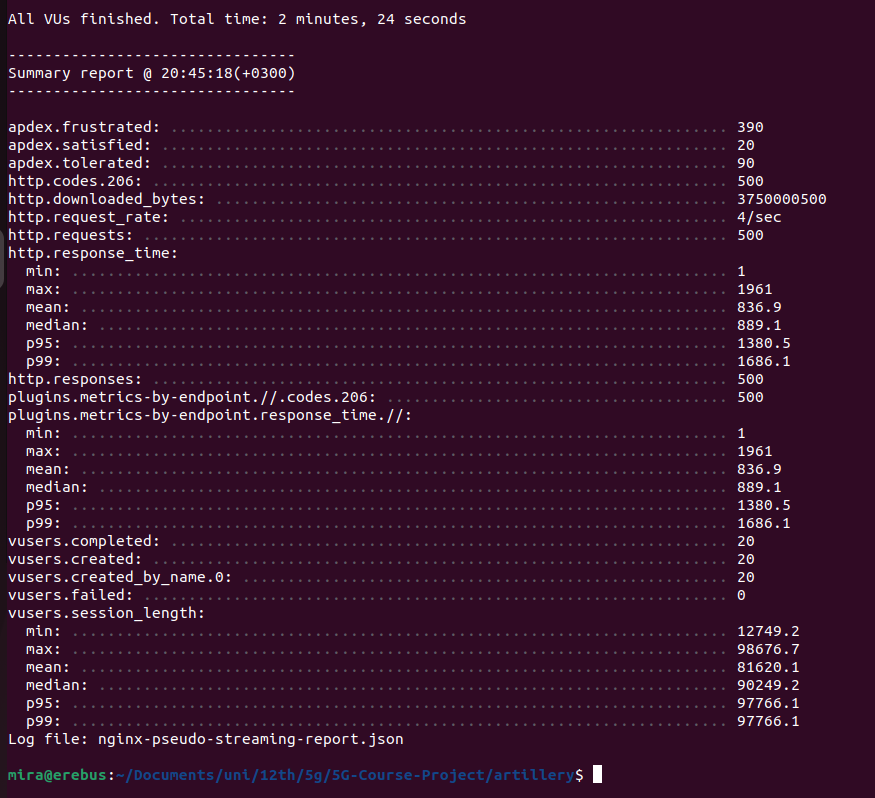


Και για κάθε pod ξεχωριστά έχουμε τα εξής παραπάνω αποτελέσματα  


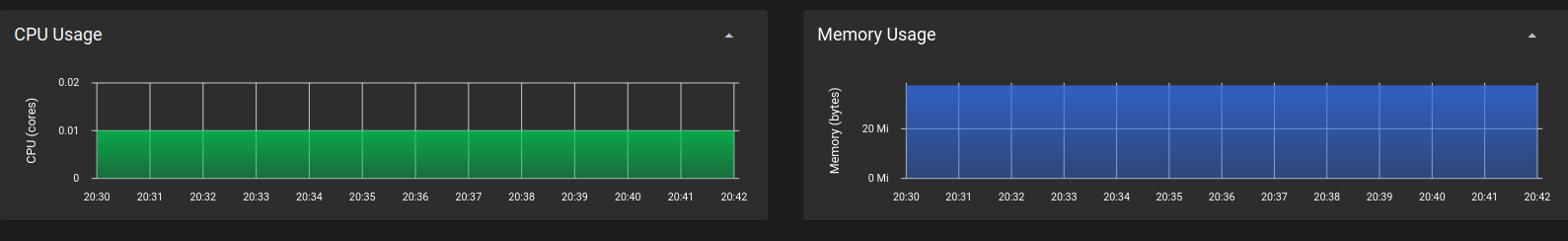
Παρατηρούμε ότι έχουμε κατά μέρο 10 mebibyte (10.48576 MB), χρησιμοποιούνται στη ram και πολύ μικρή χρήση cpu.

### Για 10 replica sets, 20 χρήστες

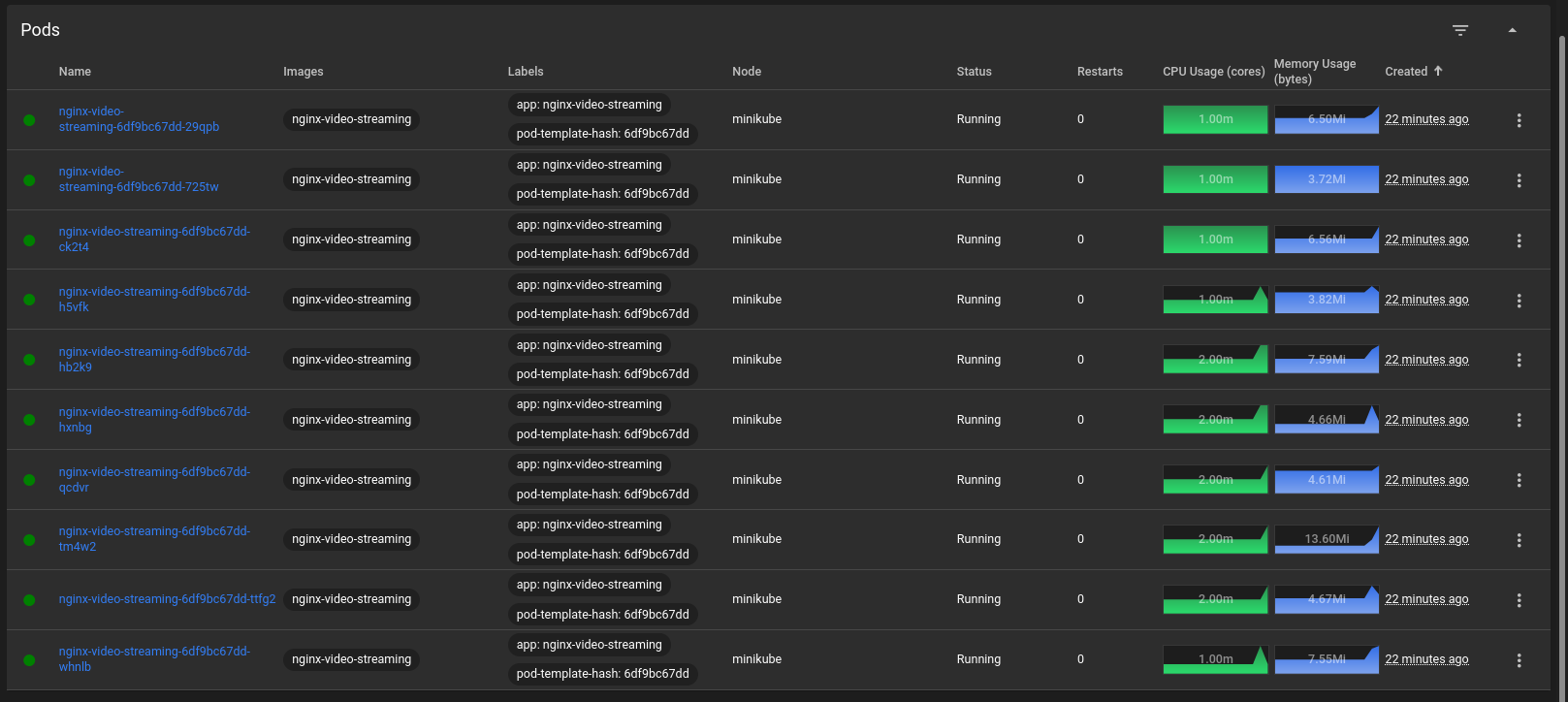
Για 10 replica sets έχουμε το παρακάτω αποτέλεσμα του artillery.



Παρατηρούμε μικρές διαφορές από το replical set =3



Η χρήση πόρων και το κάθε replica ξεχωριστά έχουμε:

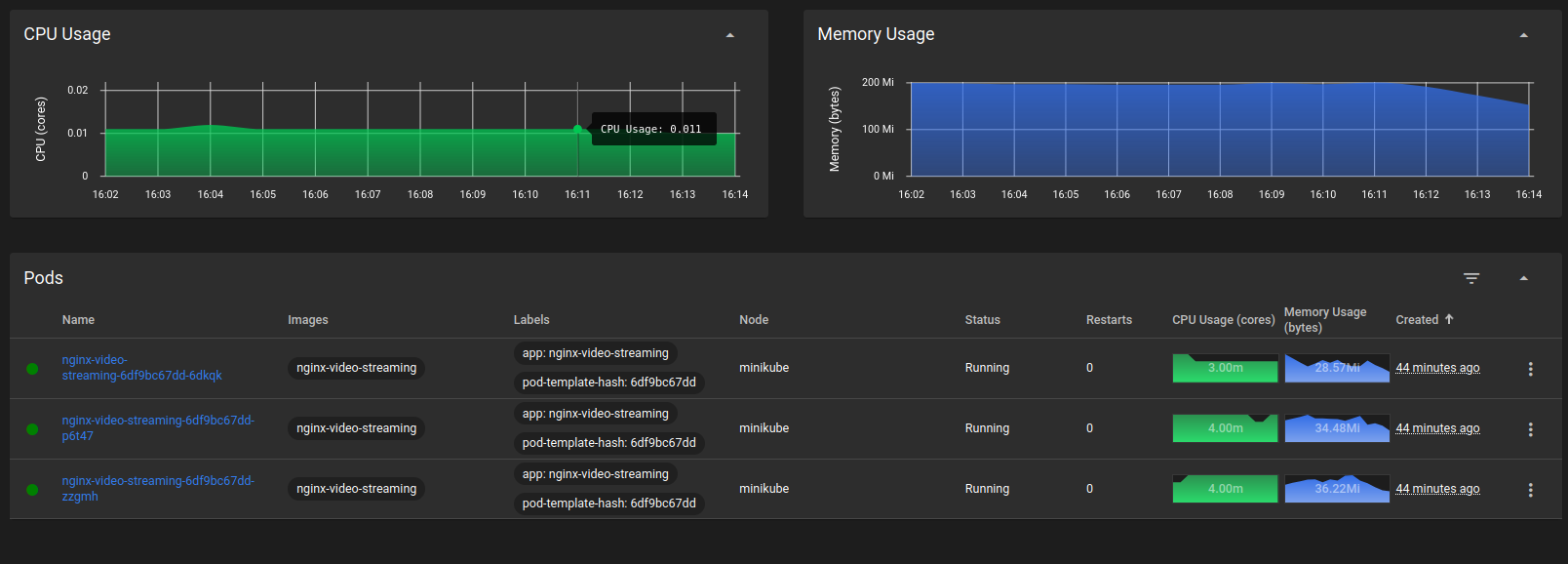


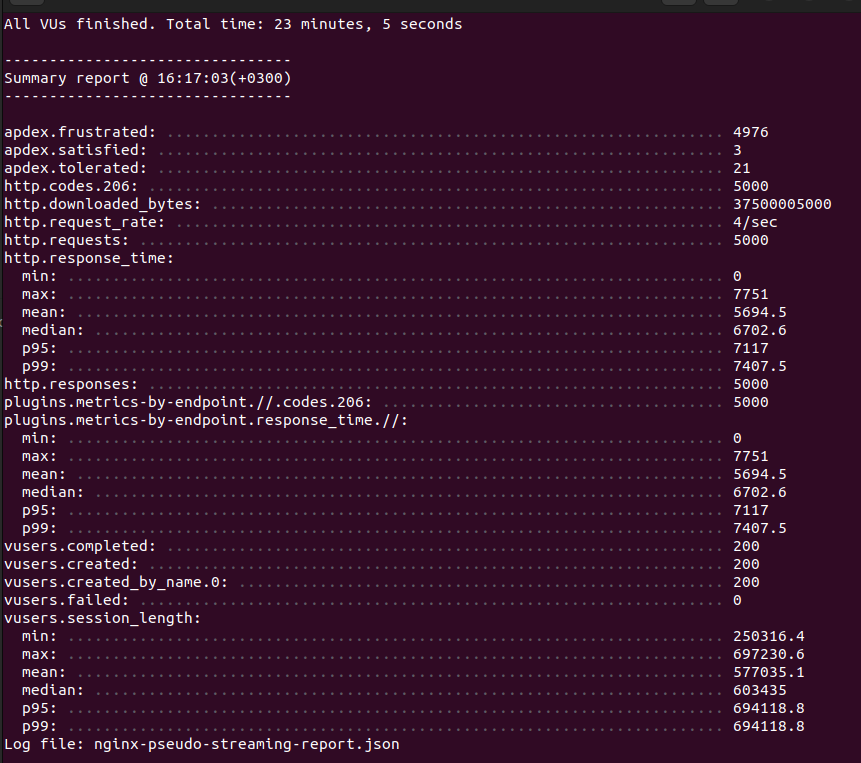
Όπου μπορούμε να δούμε αποτέλεσμα για 5 megabytes περίπου χρήση,όπως και παραπάνω χρήση της ram.

### 

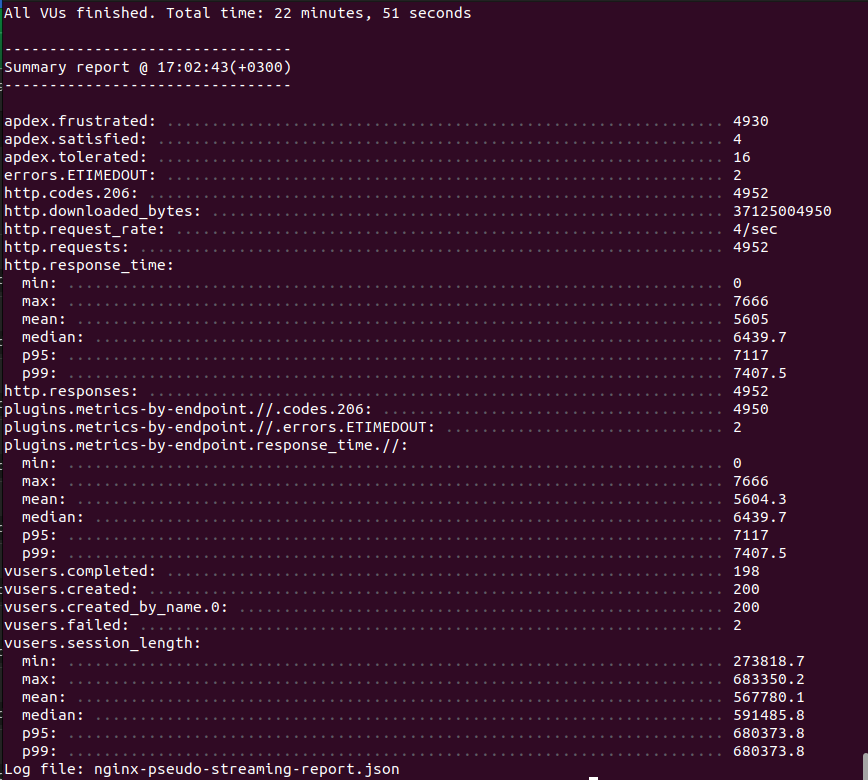
Θα μεγαλώσουμε τον αριθμό χρηστών σε 200 χρήστες, ωστέ για να δούμε τις αλλαγές και ίσως βελτιώσεις που θα προκύψουν μεταξύ 3 και 10 replica pods

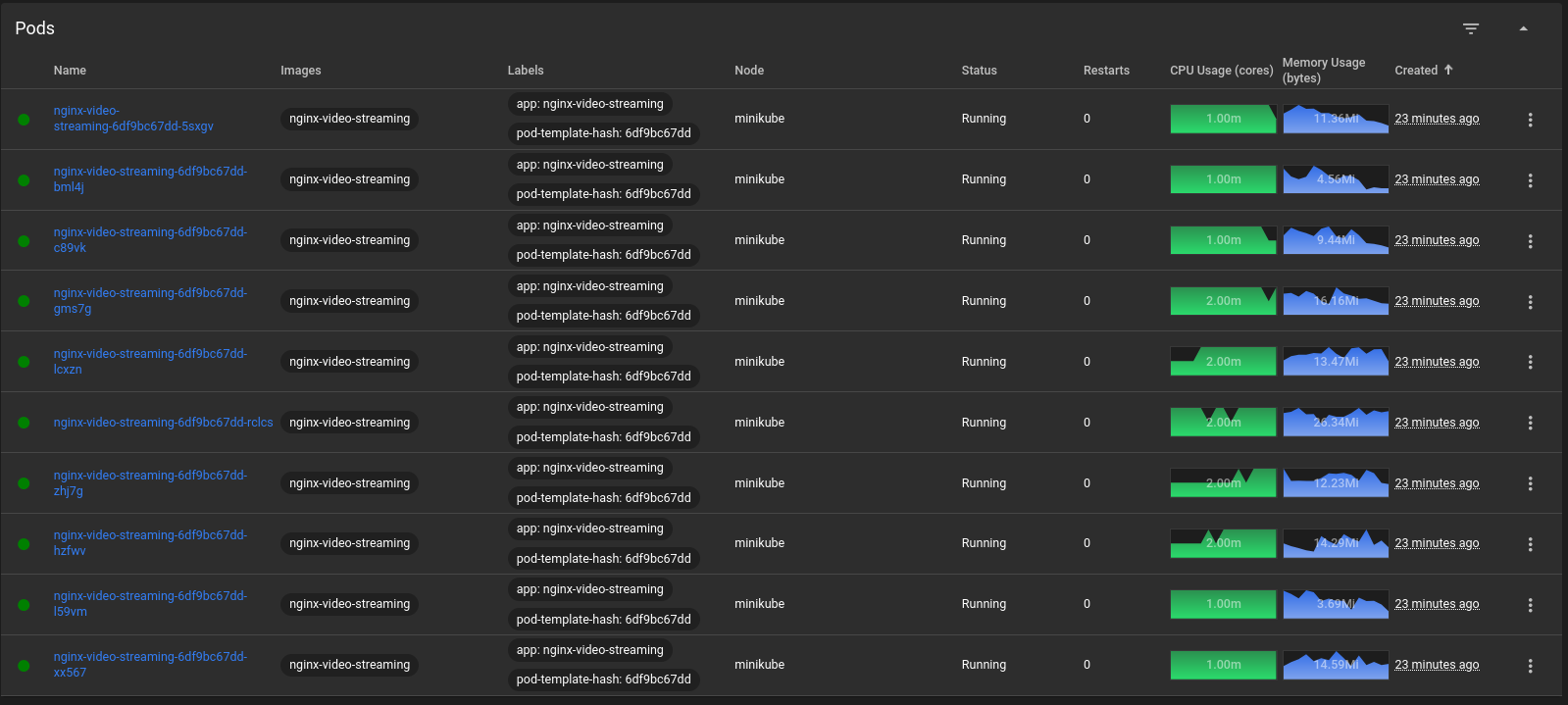
### Για 3 replica sets, 200 χρήστες

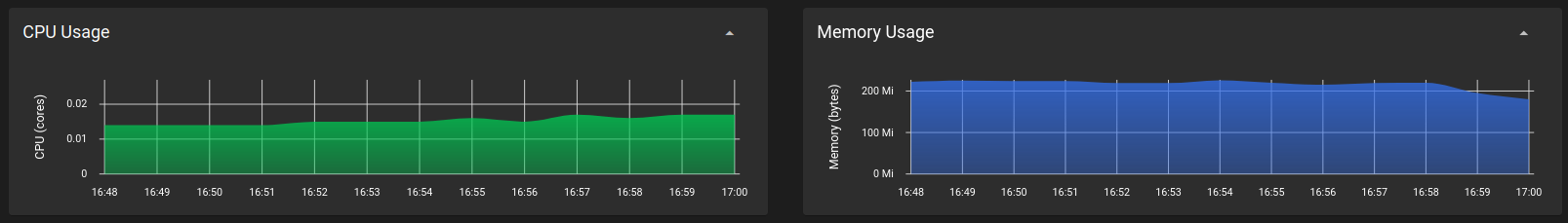




### Για 10 replica sets, 200 χρήστες







Μπορούμε να δούμε ότι οι αριθμοί εξυπηρέτησης των ψηφιακών χρηστών είναι παρόμοιοι, αλλά δίνεται η δυνατότητα να γίνει περισσότερο χρήση των πόρων που έχουμε. Λόγω της φύσης της εφαρμογής μας, που δεν απαιτεί κάποια ιδιαίτερη χρήση πόρων, δεν το χρειαζόμαστε.

Βλεπουμε μονο μια μικρη μειωση χρονου και καθε pod χρησιμοποιεί λιγοτερους πορους.

# Horizontal Pod Scaler

apiVersion: autoscaling/v2

kind: HorizontalPodAutoscaler

metadata:

name: nginx-video-streaming

spec:

scaleTargetRef:

apiVersion: apps/v1

kind: Deployment

name: nginx-video-streaming

minReplicas: 1

maxReplicas: 10

metrics:

- type: Resource

resource:

name: memory

target:

type: AverageValue

averageValue: 7Mi

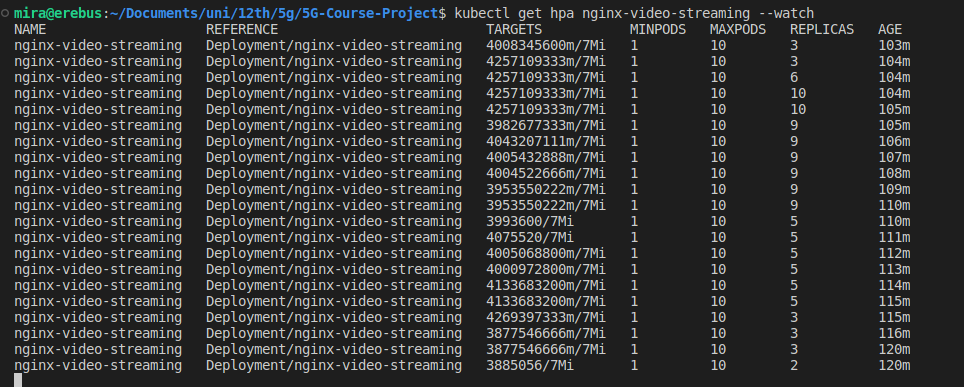
κανουμε deploy και το παραπάνω manifest: kubectl apply -f hpa.yaml

Εντολή με την οποία εκτελούμε το horizontal pod autoscaling :

kubectl run -i --tty load-generator --rm --image=busybox:1.28 --restart=Never -- /bin/sh -c

while sleep 0.01; do wget -q -O- http://10.111.204.122; done

Παρακάτω βλέπουμε ένα παράδειγμα εκτέλεσης της τεχνικής αυτής.



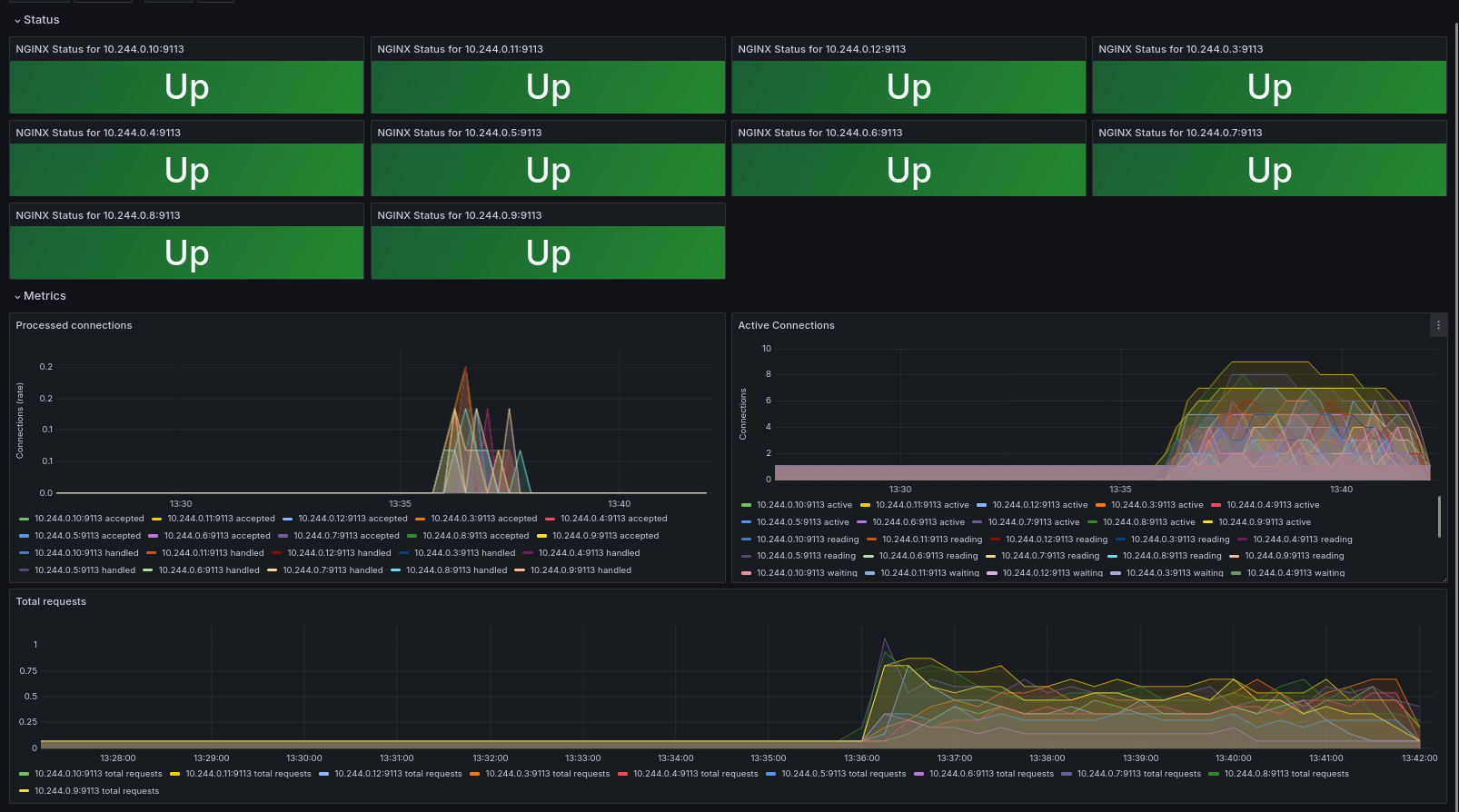
Μπορούμε να δούμε ότι το horizontal pod scaler προσαρμόζεται ανά κάποια χρονικό διάστημα, για να κρατήσει ένα μέσο όρο κοντά στο 7mibyte για κάθε pod. Βλέπουμε ότι είναι μεταβλητή διαδικασία, όπου στην αρχή ανέβαινε στο maximum αριθμό και μετά πάει στα 9 pods, και όσο οι αιτήσεις εξυπηρετούνται, πέφτουν τα αντίγραφα, μέχρι που φτάνει στο μικρότερο αριθμό αφού έχει εξυπηρετήσει ένα σημαντικό αριθμό. Έχει ίδιες δυνατότητες και δυναμική με το να έχουμε 10 pods από την αρχή, αλλά έτσι εξοικονομούνται πόροι όταν δεν υπάρχει κάποιο request, ή είναι λίγοι οι χρήστες.

# Prometheus operator

Για τις ανάγκες της εργασίας χρειαστηκε να υπάρχει ένας τρόπος αναπαράστασης με γραφικές το cpu usage και memory usage. To minikube dashboard είχε αυτη την δυνατότητα ενεργοποιώντας τα metrics-server , βεβαια δεν φαινόντουσαν πολύ αναλυτικά. Θεωρήσαμε ότι θα είχαμε πιο λεπτομερή πιο ακριβής παρατηρήσεις με την χρήση κάποιου άλλου εργαλείου. Ενα από αυτα τα εργαλεία ήταν η παρακολουθηση των pod με το prometheus και grafana. Γρήγορα ανακαλύψαμε ότι υπάρχουν έτοιμα helm charts που φαινομενικα με μια απλή εγκατάσταση θα μας έδειχναν όσα θέλαμε όπως το [kube-prometheus-stack](https://github.com/prometheus-community/helm-charts/tree/main/charts/kube-prometheus-stack).

Μετα από πολλές προσπάθειες , και πολλά διαφορετικά tutorials δεν μπορούσαμε να καταλάβουμε γιατί δεν λειτουργούσε. Θεωρήσαμε πως αυτό που χρειάζοταν ήταν το [nginx-prometheus-exporter](https://github.com/nginxinc/nginx-prometheus-exporter) , καθως οι μετρικές που έχει είναι ιδιωτικές μόνο για τον server. Οπότε προσπαθήσαμε να στήσουμε τον nginx-prometheus-exporter μαζι με την εικόνα μας , το οποίο και επιτευχθηκε. Πάλι όμως συνεχίζαμε να έχουμε το ίδιο θέμα ενώ βλέπαμε πως τα δεδομένα του nginx prometheus exporter ηταν διαθεσιμα για scraping.

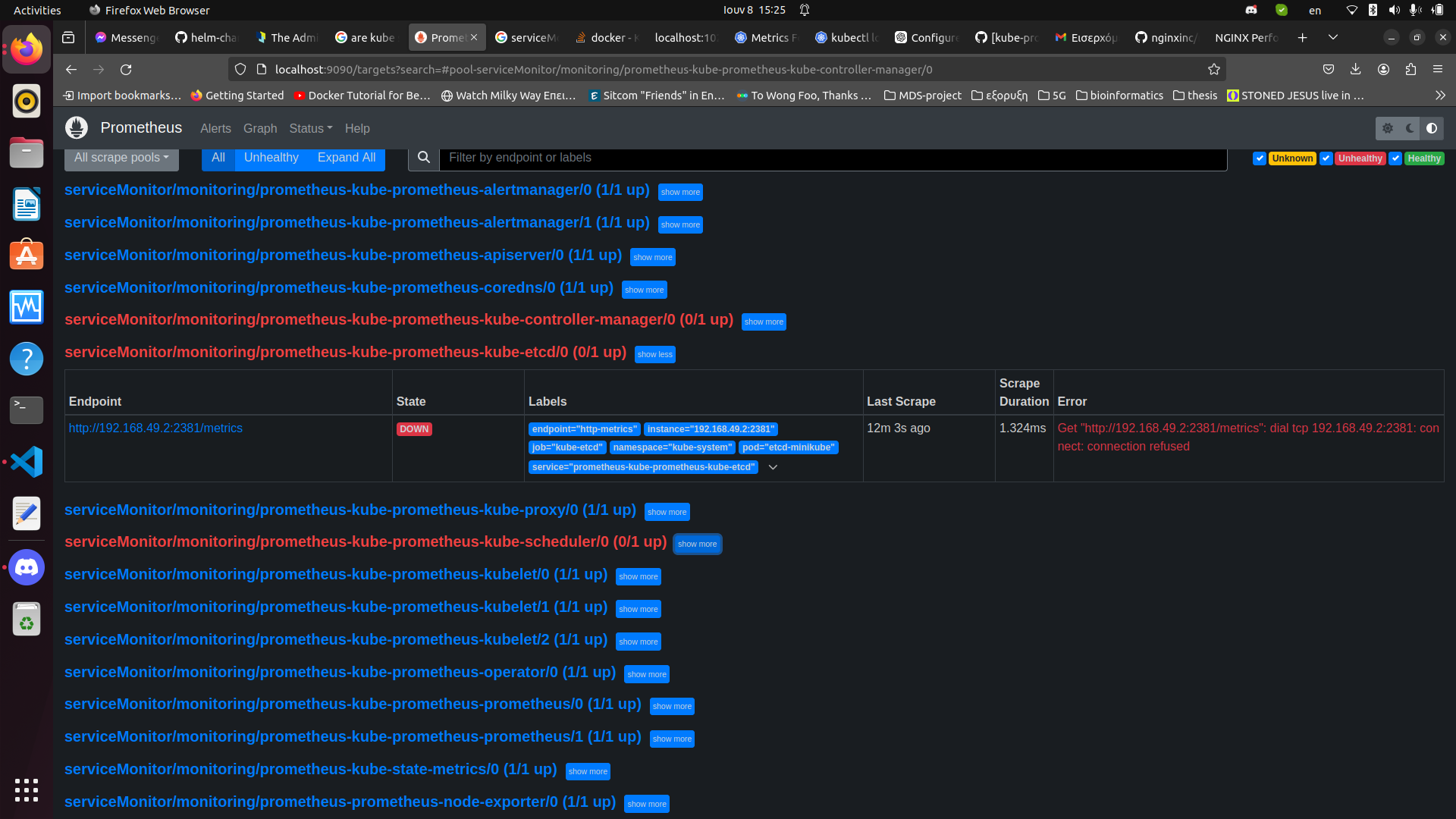
Δοκιμασαμε μια αλλη τεχνική , αυτή ήταν ο [Prometheus Operator.](https://prometheus-operator.dev/) (που περιγραφεται αναλυτικα και στο readme του github πως στήθηκε). Καθώς ακουλουθήσαμε ρητα τις οδηγίες για αυτή την τεχνική μπορέσαμε και στήσαμε ένα grafana dashboard.



Παρολαυτα μέσα σε ολα αυτά ειχαμε συνειδητοποιήσει πως οι μετρικές που γινόντουσαν expose από τον nginx-prometheus-exporter δεν ηταν οι επιθυμητες , για τις μετρησεις του cpu και του memory. Αυτες οι μετρικές θα φαινόντουσα με την βοήθεια του [kube-state-metrics](https://github.com/kubernetes/kube-state-metrics) και [node-exporter.](https://github.com/prometheus/node_exporter)

Αυτά υπήρχαν στο helm chart, για να τα σετάρουμε τωρα μαζι με το prometheus operator χρειαζοταν πολύ μεγαλύτερη επεξεργασια και configurations.

Παρατηρήσαμε μάλιστα πως ο λόγος που δεν δουλευε το αρχικο kube-prometheus-stack ήταν λόγω καποιων μετρικών που δεν γινόντουσαν σωστα scrape από το prometheus , όχι όμως των μετρικών του nginx όπως θεωρήσαμε αλλα αυτων που φαίνονται στην εικόνα παρακάτω.



Προσπαθήσαμε να αλλάξουμε κάποια configurations αλλα δεν παρατηρήσαμε κάποια αλλαγή.