

## روش های خودکار سازی در پلتفرم کوبرنیتیز:

### ۱. خودکار سازی تعداد گره ها (cluster auto-scaler):

یکی از ابزار کوبرنیتیز (Kubernetes) است که به ما این اجازه را میدهد سایز کلاستر خود را که شامل تعدادی نود است؛ افزایش یا کاهش دهیم. این خودکار ساز در لایه Infrastructure کار میکند و هر موقع پاد های ما دچار کمبود منابع شدند و گره ای وجود نداشت که این پاد تخصیص یابد؛ آنگاه این ابزار یک نود به کلاستر اضافه میکند. و هنگامی که منابع ما از یک حدی به بعد مورد استفاده قرار نگرفتند؛ نود ها کم میشوند تا در مصرف منابع سخت افزاری صرفه جویی شود. این ابزار کلاستر ما را انعطاف پذیر و مقیاس پذیر میکند. به منظور اینکه این ابزار درست کار کند اقداماتی باید صورت پذیرد.

الف) سایز نود ها باید از لحاظ مقدار محاسبات و حافظه یکسان باشد تا این ابزار بتواند کار خودکار سازی را برای کلاستر انجام دهد.

ب) درخواست های محاسباتی و حافظه ای برای هر پاد باید مشخص باشد به این منظور که این ابزار بتواند مقدار استفاده (Utilization) از این نود را محاسبه کند و در صورت نیاز تعداد را کاهش یا افزایش دهد در غیر این صورت این ابزار دچار مشکل خواهد شد.

ج) باید یک حدی را مشخص کنیم که تعداد نود ها از این مقدار پایین نیایند و باعث نشود که سرویس های مهم و اساسی ما کم یا از بین بروند و معیار دسترس پذیری بالا (High availability) سرویس ها خدشه دار شوند. در کوبرنیتیز ما با استفاده از PodDisruptionBudget میتوانیم پاد ها را به گونه ای تنظیم کنیم که از یک مقدار مقدار مشخص کمتر نشوند و همیشه یک تعداد مشخص در حال اجرا باشند و هنگامی که ادمین کلاستر میخواست نودی را خاموش کند که این مقدار پاد کمتر از حد مجاز شود این اجازه را نمیدهد.

د) درخواست های محاسباتی و حافظه ای (CPU and Memory Requests) که برای پاد ها مشخص میکنیم باید نزدیک به واقعیت باشد (نه خیلی بیشتر و نه خیلی کمتر) تا این ابزار دچار اشتباه

محاسباتی نشود و منابع اضافی بیشتر یا کمتر استفاده نکند. این کار باعث میشود تا منابع به طور بهینه مصرف شود. برای اینکه درخواست های ما نزدیک به واقعیت باشد باید از ابزار VPA )

خودکار سازی (تخصیص منابع) استفاده کنیم یا خودمان بر اساس عملکرد سرویس در مواقع مختلف و زیر بار های مختلف تشخیص دهیم که چه میزان منابع برای این سرویس لازم است.

با توجه به اینکه ما در این پروژه از سه نود استفاده میکنیم و این سه نود را به صورت Local (محلی) با استفاده از ابزار Kubeadm راه اندازی کردیم و از فراهم آور های ابری (cloud providers) مانند AWS , Google Cloud و Azure استفاده نکرده ایم؛ امکان استفاده از این ابزار وجود ندارد.

## ۲. خودکار سازی تخصیص منابع (VPA):

این خودکار ساز با توجه به منابعی که یک پاد مصرف میکند؛ درخواست های محاسباتی و حافظه ای مناسبی را میتواند هم پیشنهاد بدهد یا خودش آن درخواست ها را برای پاد تنظیم کند. بنابراین زمانی که مصرف یک پاد زیاد باشد به آن نسبت درخواست ها را افزایش میدهد و بالعکس زمانی که مصرف کم باشد درخواست ها را کم میکند. این باعث میشود که منابع ما بهینه مدیریت شوند.

بخش مهم این خودکار ساز Recommender (پیشنهاد دهنده) است که با استفاده از Metrics-server که در بخش بعدی توضیح داده خواهد شد؛ میزان مصرفی محاسباتی و حافظه ای را میگیرد و بر اساس آن به ما بهترین پیشنهاد خود را میدهد. - نصب:

برای نصب این خودکار ساز ابتدا باید Metrics-server را نصب داشته باشید که در بخش بعدی نصب آن توضیح داده خواهد شد.

سپس از این آدرس با استفاده از git آن را clone کنید:

```
git clone https://github.com/kubernetes/autoscaler.git
```

و با رفتن به پوشه vertical-pod-autoscaler دستور را اجرا کنید:

```
./hack/vpa-up.sh
```

حال با استفاده از فایل Yaml زیر میتوانیم VerticalPodAutoscaler خود را بسازیم.

```

apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: my-rec-vpa
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind: Deployment
    name: my-rec-deployment
  updatePolicy:
    updateMode: "Off"

```

همانطور که در شکل بالا مشاهده میکنیم این خودکار ساز برای یک deployment تنظیم شده است و چون updateMode غیر فعال است؛ این خودکار ساز تنها مقادیری که برای Cpu و Memory مناسب هست را به ما پیشنهاد میدهد و این پیشنهاد را برای سرویس ما اعمال نمیکند.

```

recommendation:
  containerRecommendations:
  - containerName: my-rec-container
    lowerBound:
      cpu: 25m
      memory: 262144k
    target:
      cpu: 25m
      memory: 262144k
    upperBound:
      cpu: 7931m
      memory: 8291500k

```

این خودکار ساز همانطور که در شکل بالا آمده است؛ مقادیر حد پایین ؛ حد مطلوب و حد حداکثر را برای ما پیشنهاد داده است. حال اگر updateMode فعال شود مقدار مطلوب را در قسمت درخواست های محاسباتی و حافظه ای اعمال میکند.

```
resources:
  requests:
    cpu: 510m
    memory: 262144k
```

و در اینجا بعد از آنکه این خودکار ساز پاد را دوباره میسازد؛ این درخواست ها را در پاد مد نظر ما اعمال میکند و دیگر لازم نیست که ما بخواهیم خودمان به طور دستی این درخواست ها را برای پاد ها مشخص کنیم. مشکلی که در این خودکار ساز وجود دارد این است که چون هنوز در حالت آزمایشی قرار دارد با خودکار ساز سرویس ها هنوز سازگار نیست و نمیتوان از هر دو همزمان هنگامی که از معیار های محاسباتی و حافظه ای برای مقیاس پذیری استفاده میکنیم؛ استفاده کرد و فقط این دو زمانی باهم کار میکنند که ما از custom metrics استفاده کنیم که معیار هایی هستند که خودمان آنها را تعریف میکنیم و با استفاده از آنها سرویس های خود را مقیاس پذیر میکنیم. چون کلاستر ما کوچک است و خودمان با استفاده از میزان مصرفی پاد ها در زمان های مختلف میتوانیم درخواست های محاسباتی و حافظه ای مناسب را برای پاد های خود مشخص کنیم. به علاوه به دلیل آنکه میخواهیم از معیار های محاسباتی و حافظه ای استفاده کنیم؛ در نتیجه از این خودکار ساز در پروژه استفاده نخواهیم کرد.

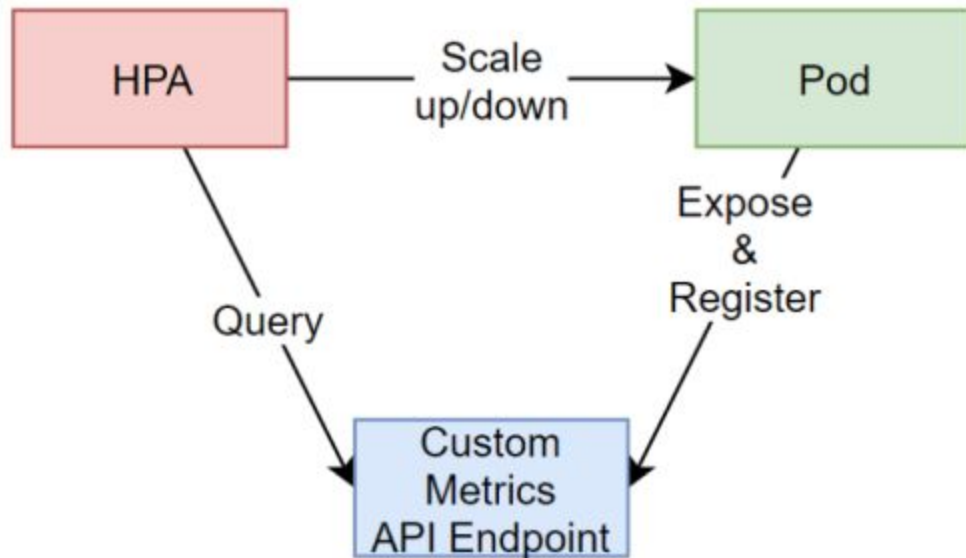
### ۳. خودکار سازی سرویس ها (HPA):

#### الف) Metrics server

بخش سوم و بخش مهم این پروژه در این قسمت است که ما میخواهیم سرویس های خود را بر اساس معیار های مختلف مقیاس پذیر کنیم به منظور آنکه سرویس های خود را Responsive (پاسخگو) نگه داریم و از تاخیر زیاد و بار زیاد بر روی یک سرویس جلوگیری شود.

برای اینکه ما بتوانیم پاد های خود را بر اساس معیار های محاسباتی و حافظه ای (Cpu and Memory metrics) مقیاس پذیر کنیم؛ باید از یکی از ابزار هایی که میتوان در کنار kubernetes-api نصب کرد Metrics-server است که اطلاعات را که از گره ها با استفاده از Kubelet که در هر گره وجود دارد

جمع آوری میکند و از طریق API (metrics.k8s.io) در دسترس HPA قرار میدهد که بتواند میزان مصرفی پاد ها رصد کند و هر موقع میزان مصرفی محاسباتی یا حافظه ای ما از حد مشخص شده رد شد؛ این خودکار ساز شروع به افزایش پاد ها کند.



این شکل یک شمای کلی از کارکرد HPA بیان میکند که در این خودکار ساز با استفاده از metrics server که معیار ها را از هر گره استخراج میکند؛ کار مقیاس پذیری را انجام میدهد.

HPA برای آنکه مشخص کند چه تعداد پاد لازم است تا افزایش یا کاهش دهد از الگوریتم ساده زیر استفاده میکند.

$$\text{desiredReplicas} = \text{ceil}[\text{currentReplicas} * (\text{currentMetricValue} / \text{desiredMetricValue})]$$

که تعداد پاد های فعلی را در تقسیم مقدار الان بر مقدار مطلوب ضرب میکند. برای آنکه بیشتر مشخص شود. به طور مثال اگر مقدار مطلوب ما برای یک پاد ۱۰ درخواست در ثانیه باشد و مقدار حال حاضر ۲۰ درخواست بر ثانیه است باید پاد ها را دو برابر کند و اگر از این مقدار کمتر شد پاد ها را کاهش میدهد.

در این خودکار ساز یک مفهومی به Cooldown Period وجود دارد به این معنی که یک زمانی طول میکشد که این خودکار ساز پاد ها را افزایش دهد یا کاهش دهد. این به این منظور است که شاید پیک های گذرا رخ دهد و لازم نباشد که مقیاس پذیری صورت بگیرد. همین طور در موقع کاهش ممکن است بار زیادی در حال حاضر وجود دارد ولی یک وقفه در بار بیفتد و دوباره بار زیاد ادامه پیدا کند. پس این زمان مهم است که این خودکار ساز از کم شدن بار یا زیاد شدن بار اطمینان حاصل کند و بیهوده کار مقیاس پذیری را انجام ندهد. مقدار پیش فرض ۵ دقیقه است.

حال به سراغ راه اندازی این خود کار ساز با استفاده از metrics server میرویم.

## - نصب Metrics server

برای راه اندازی metrics-server از پکیج منیجر Helm استفاده میکنیم که این پکیج منیجر Third-party software است که به کلاتسر ما دسترسی پیدا میکند و سرویس هایی که نیاز داریم را برای ما نصب میکند و کار را برای ما بسیار آسان میکند و لازم نیست کلی کانفیگ فایل و آبجکت های مختلف را در کلاستر خود نصب کنیم.

ابتدا برای نصب این پکیج باید یکسری مقادیر را تغییر بدهیم به این منظور که در حین نصب دچار مشکل نشویم. دستور زیر را وارد میکنیم و مقادیر را تغییر میدهیم.

```
helm show values stable/metrics-server > /tmp/metrics-server.values
```

دو مقدار زیر باید به این گونه باشند.

```
hostNetwork: true
```


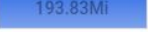
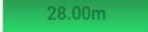






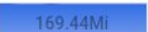
```
containers:
- name: metrics-server
  image: 'k8s.gcr.io/metrics-server-amd64:v0.3.6'
  command:
    - /metrics-server
    - '--kubelet-insecure-tls'
    - >-
      --kubelet-preferred-address-types=InternalDNS,InternalIP,ExternalDNS
      ,ExternalIP,Hostname
    - '--cert-dir=/tmp'
    - '--logtostderr'
    - '--secure-port=8443'
```

در قسمت command باید خط دوم و سوم را اضافه کنیم.

بعد از این تغییرات نوبت به نصب این پکیج میرسد که با استفاده از دستور زیر metrics server را نصب میکنیم.

```
helm install metrics-server stable/metrics-server --namespace metrics --values
/tmp/metrics-server.values
```

حال میتوانیم با استفاده از دشبورد کوبرنتیز مقدار مصرفی محاسباتی و حافظه ای برای هر پاد را مشاهده کنیم. و در بخش command line با استفاده از دستور kubectl top pods میزان مصرفی را مشاهده کنیم که در صفحه بعدی شکل های دشبورد و command line آمده است.

Name	Namespac Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Usage (bytes)	Age	↑
✓ api-gateway-5df59ddbc8-ntkh4	default app: api-gateway pod-template-hash: 5df59ddbc8	mpouyakh	Running	0	 24.00m	 193.83Mi	4 hours	⋮
✓ position-tracker-54fb5494bf-sc4nd	default app: position-tracker pod-template-hash: 54fb5494bf	mpouyakh	Running	0	 28.00m	 239.49Mi	4 hours	⋮
✓ webapp-785d5b86bf-8j8j8	default app: webapp pod-template-hash: 785d5b86bf	mpouyakh	Running	0	 1.00m	 6.47Mi	4 hours	⋮
✓ queue-ff85789bd-m2l6x	default app: queue pod-template-hash: ff85789bd	mpouyakh	Running	0	 84.00m	 233.64Mi	4 hours	⋮
✓ position-simulator-7f4d479d95-54xvv	default app: position-simulator pod-template-hash: 7f4d479d95	mpouyakh	Running	0	 2.00m	 169.44Mi	4 hours	⋮

```

root@mpouyakh-m:/home/mpouyakh# kubectl top pods
NAME                                                    CPU(cores)   MEMORY
api-gateway-69956d88d5-kf7v7                          11m          190Mi
client-deployment-7758f66c7f-t4fwp                    1m           158Mi
mongodb-64f58969c7-nbx8p                              763m        278Mi
nfs-client-provisioner-1586887262-7996f6d987-bgc9v    4m           11Mi
position-simulator-57bc476846-8qdkp                   24m          179Mi
position-tracker-55c4c6468-d9bmh                      13m          246Mi
postgres-deployment-787848c9dd-fzxm8                  1m           33Mi
pouya-nginx-ingress-controller-59f4fdb9dd-dntwj       11m          109Mi
pouya-nginx-ingress-controller-59f4fdb9dd-dxc9w       4m           72Mi
pouya-nginx-ingress-default-backend-95b66b7b-rv5rz    1m           4Mi
queue-68fdb97c67-svpmn                                80m          264Mi
redis-deployment-8645557955-xdvcs                     2m           3Mi
server-deployment-578fcb4457-crwt2                    0m           43Mi
webapp-55b7fbcf88-v9c8c                               1m           17Mi
worker-deployment-94d8b8f44-vwcjv                     0m           34Mi

```

همانطور که در شکل های بالا مشاهده میکنیم میزان مصرفی هر پاد مشخص است که برای قسمت cpu منظور از m ؛ milicore است که هر یک cpu برابر است با 1000m.

حال با انجام این بخش؛ سراغ مرحله بعد که کار با HPA است میرویم و برای یک سرویس این مقیاس پذیر کردن را انجام میدهیم.

برای اینکه خودکار ساز ما در این بخش کار کند؛ باید برای پاد هایی که میخواهیم مقیاس پذیری را انجام دهیم درخواست های محاسباتی و حافظه ای مناسبی را برای پاد های خود تعیین کنیم و چون در این بخش که مقیاس پذیر کردن بر اساس میزان

مصرفی cpu و memory است ؛ نمیتوانیم از VPA استفاده کنیم باید خودمان بر اساس منابعی که در کل در اختیار داریم و سابقه مصرفی پاد مورد نظر ؛ یک درخواست نزدیک به واقعیت تعیین کنیم.

برای تست این مقیاس پذیری ما از client-deployment استفاده میکنیم که بخش فرانت اند وب اپلیکیشن ما است.

حال با بررسی مقدار مصرفی این پاد و منابعی که در اختیار داریم ما درخواست ها و محدودیت های زیر را برای این پاد در نظر گرفته ایم.

```
resources:
  limits:
    cpu: '1'
    memory: 500Mi
  requests:
    cpu: 70m
    memory: 150Mi
```

## - ساختن خودکار ساز

و سپس فایل Yaml این خودکار ساز را نوشتیم که به صورت زیر است:

```
1  apiVersion: autoscaling/v2beta1
2  kind: HorizontalPodAutoscaler
3  metadata:
4    name: client-deployment
5  spec:
6    scaleTargetRef:
7      apiVersion: apps/v1
8      kind: Deployment
9      name: client-deployment
10   minReplicas: 1
11   maxReplicas: 10
12   metrics:
13     - type: Resource
14       resource:
15         name: memory
16         targetAverageUtilization: 170
17     - type: Resource
18       resource:
19         name: cpu
20         targetAverageUtilization: 200
```

همانطور که مشخص است در خط اول ابتدا API مد نظر را مشخص کردیم که با استفاده از این API به این خودکار ساز میتوانیم دسترسی پیدا کنیم و در قسمت بعدی مشخص میکنیم که چه نوع آبجکتی (Object) میخواهیم درست کنیم. بخش مهم دیگر این فایل قسمتی است که معیار ها را مشخص میکنیم. در این فایل دو معیار cpu و memory را تعیین کردیم که مقدار مطلوب ما برای cpu مساوی دو برابر (200%) درخواستی که در صفحه قبل مشخص کردیم که برابر 140m میشود. و برای memory ما ۱۷۰ درصد مقدار درخواستی که تعیین کردیم؛ برابر 255MB است.

بعد از ساختن این آبجکت در کوبرنتیز؛ خودکار ساز ما به این شکل در خواهد آمد.



NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS
client-deployment	Deployment/client-deployment	98%/170%, 0%/200%	1	10	1
position-tracker	Deployment/position-tracker	1500m/40	1	10	1
pouya-nginx-ingress-controller	Deployment/pouya-nginx-ingress-controller	200m/20, 2985m/500	1	10	2

در خط اول این شکل مشاهده میکنیم که در قسمت Targets دو معیار ما آورده شده اند و تعداد Replica برابر ۱ است چون مقدار فعلی کمتر از مقدار تعیین شده است. بعد از آنکه بار بر روی این پاد اضافه کردیم مشاهده میکنیم به صورت خودکار کوبرنتیز تعداد پاد ها را افزایش خواهد داد و سعی میکند مقدار فعلی را به زیر مقدار تعیین شده ببرد. همچنین بعد از آنکه بار کم شد این خودکار ساز بعد از ۵ دقیقه که زمانی است که خودکار ساز صبر میکند تا تعداد پاد ها را کاهش دهد؛ تعداد پاد ها را کم میکند.

### - تولید بار

حال سراغ تولید بار میرویم که برای این کار ما از ابزار siege استفاده میکنیم. این ابزار درخواست های Http را به میزانی که مد نظر است در یک بازه زمانی مشخص شده برای پاد ما میفرستد. بعد از نصب این پکیج در سیستم خود آدرس پاد خود را به این تولید کننده بار (load generator) میدهم و ۲۰ کاربر همزمان درخواست های خود را برای این سرویس به مدت ۱۰ دقیقه میفرستیم و بعد نتیجه را مشاهده کنیم.

siege -q -c 20 -t 10 <http://10.102.108.164:3000>

آدرس پاد ما در دستور بالا آمده است که بر روی پورت ۳۰۰۰ میدهد.

### - نتایج خودکار ساز

حال نتایج عملکرد این خودکار ساز را در عکس های زیر مشاهده میکنیم.

TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
106%/170%, 722%/200%	1	10	1	10h
1600m/40	1	10	1	47h
200m/20, 2490m/500	1	10	2	44h
0/40	1	10	1	46h

در اینجا مشاهده میکنیم که بار بر روی پاد ما زیاد شده است و تقریباً ۳.۵ برابر حدی که برای خودکار ساز خود تعیین کرده ایم شده است. ولی معیار ما از لحاظ حافظه ای کمتر از حد مجاز است و این مقیاس پذیری بر اساس معیار محاسباتی انجام میپذیرد.

TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
109%/170%, 898%/200%	1	10	4	10h
1600m/40	1	10	1	47h
200m/20, 7440m/500	1	10	2	44h
0/40	1	10	1	46h

در مرحله بعدی میبینیم که بار زیاد تر شده است و خودکار ساز تعداد بیشتری از این پاد را میسازد و هنوز مقدار مصرفی پاد ها بیشتر از حد تعیین شده است که یک پاد مجاز است مصرف کند. پس انتظار میرود که خودکار ساز پاد های بیشتری تولید کند.

TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
105%/170%, 274%/200%	1	10	10	10h
1400m/40	1	10	1	47h
200m/20, 2985m/500	1	10	2	44h
0/40	1	10	1	46h

وقتی ماکسیمم تعداد مجاز پاد ها تولید شد؛ میبینیم که درصد مصرفی هر پاد کمتر پایین تر آمده است و بار بر روی این پاد ها بخش شده است و یک پاد بار زیادی را تحمل نمیکند. و در شکل زیر خواهیم دید که میزان مصرفی هر پاد از مقدار تعیین شده کمتر شده است.

TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
101%/170%, 104%/200%	1	10	10	10h
1600m/40	1	10	1	47h
200m/20, 16890m/500	1	10	2	44h
0/40	1	10	1	47h

بعد از اینکه بار کمتر شد؛ این خودکار ساز شروع به کمتر کردن تعداد پاد ها بر اساس مرور زمان میکند. و همانطور که گفته شد حدوداً ۵ دقیقه طول میکشد تا تعداد پاد ها را کم کند.

TARGETS	MINPODS	MAXPODS	REPLICAS
103%/170%, 0%/200%	1	10	7
1500m/40	1	10	1
200m/20, 2980m/500	1	10	2
0/40	1	10	1

و همین طور که بار کمتر میشود؛ این خودکار ساز تعداد پاد ها را هم کمتر میکند.

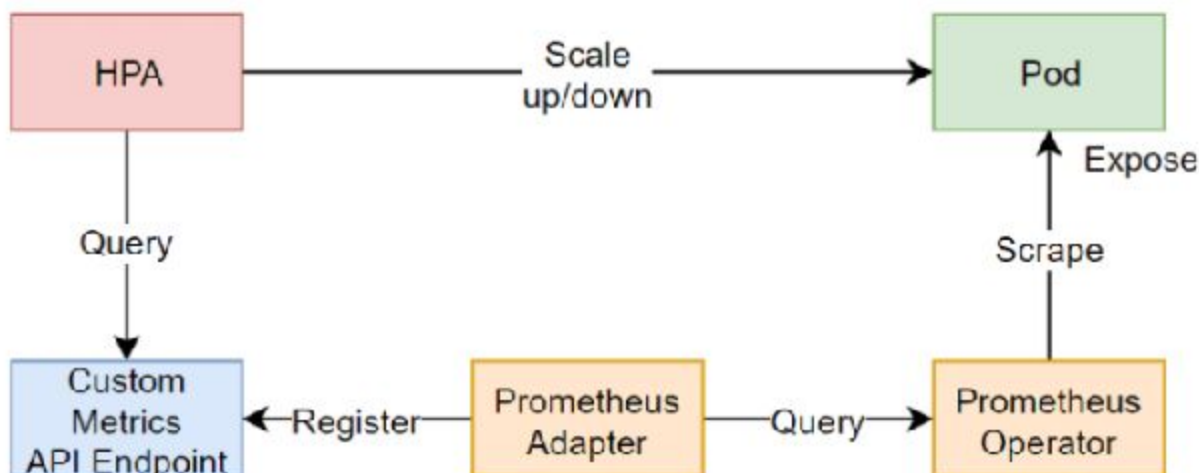
TARGETS	MINPODS	MAXPODS	REPLICAS
104%/170%, 0%/200%	1	10	4
1500m/40	1	10	1
200m/20, 2980m/500	1	10	2
0/40	1	10	1

### Custom metrics (ب)

در این بخش ما قصد این را داریم که اپلیکیشن خود را بر اساس معیار های بیشتری بتوانیم مقیاس پذیر کنیم. در بخش قبل ما با استفاده از metrics server تنها میتوانستیم معیار های محسباتی و حافظه ای پاد ها را استخراج کنیم و از طریق metric API در دسترس HPA قرار دهیم. ولی اگر بخواهیم سرویس های خود را بر اساس معیار های دیگری مقیاس پذیر کنیم باید سراغ راه حل های دیگر برویم.

راه حلی که برای این منظور وجود دارد ؛ Prometheus adapter است که یک نوع Metric API server است که وظیفه آن را دارد که معیار هایی که Prometheus به عنوان معیار جمع آور (Metrics collector) انجام میدهد را از طریق Custom Metrics API در اختیار HPA قرار بدهد تا بتواند بر اساس معیار ها کار مقیاس پذیری را انجام دهد. همچنین Prometheus adapter یک فایل کانفیگ به عنوان ورودی دریافت میکند که در آن فایل مشخص شده است ما چه معیار هایی را میخواهیم و چگونه آن را به گونه ای که مورد مطلوب ما هست تحویل ما بدهد.

مورد بعدی که وجود دارد خود Prometheus معیار هایی از نود ها و پاد ها جمع آوری میکند ولی بعضی از معیار ها مخصوص خود اپلیکیشن هستند و باید در اپلیکیشن تنظیم شوند و آن معیار ها را در اختیار Prometheus قرار دهند.

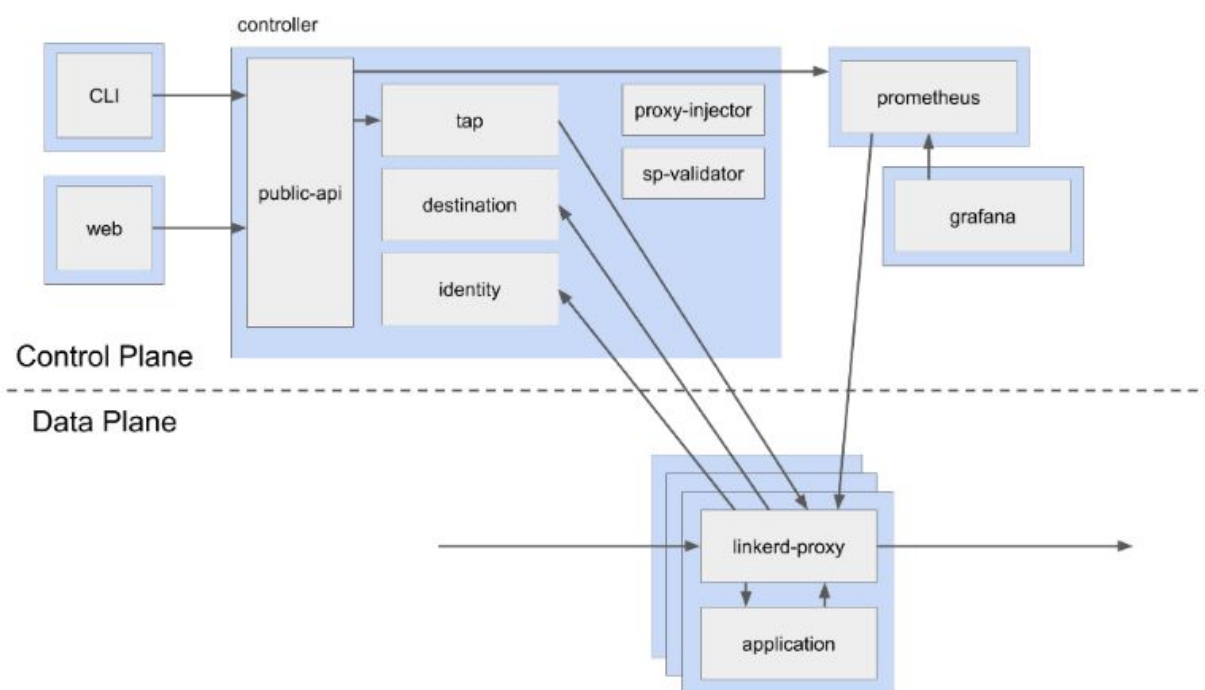


در این شکل به خوبی نشان داده شده است که چگونه ما با استفاده از Prometheus adapter می‌خواهیم کار مقیاس پذیری را انجام دهیم و بخش‌های مختلف چگونه با هم ارتباط برقرار می‌کنند. همانطور که مشاهده می‌شود؛ Prometheus معیارها را از پادها استخراج می‌کند و به صورت زمانی می‌تواند به ما نشان دهد. سپس Prometheus adapter این معیارها را به گونه‌ای که ما می‌خواهیم در می‌آورد و از طریق API مشخص شده در شکل برای HPA در دسترس قرار می‌دهد. حال به سراغ مرحله بعد می‌رویم که در صفحه قبل اشاره کردیم. برای اینکه بعضی از معیارها را بتوانیم تعریف کنیم و بر اساس آنها مقیاس پذیری را انجام دهیم باید کد اِپلیکیشن خود را به گونه‌ای تغییر دهیم که بتواند این معیارها را در اختیار Prometheus بگذارد. ولی ما با استفاده از Linkerd که یک Service Mesh برای کوبرنتیز است؛ می‌خواهیم بدون آنکه تغییری در سرویس‌های بدهیم بعضی از معیارهای مفید را برای مقیاس پذیری استفاده کنیم.

## - نصب و ساختار linkerd

ابتدا توضیح مختصری از اینکه این Service Mesh چگونه کار می‌کند داده می‌شود سپس سراغ استفاده از آن می‌رویم. ساختار این سیستم به این گونه است که دو بخش دارد. بخش اول Control Plane نام دارد که مسئول مدیریت پروکسی‌هایی است که در کنار هر سرویس قرار می‌گیرد است و مسئول ارتباطات این پروکسی‌ها است. مسئولیت‌های دیگر این بخش مدیریت؛ جمع‌آوری اطلاعات از پروکسی‌ها؛ فراهم کردن ارتباطات بر اساس تکنولوژی TLS و فراهم کردن API‌هایی برای ادمین کلاستر است که بتواند داده‌هایی که از این پروکسی‌ها جمع‌آوری شده دسترسی پیدا کند و بتواند تغییراتی را در این پروکسی‌ها اعمال کند.

بخش دیگر این سیستم؛ data plane است که مربوط به پروکسی هایی است که در کنار سرویس های ما قرار میگیرید و درخواست هایی که به سرویس ها میشود را دریافت میکند و به سرویس میفرستد سپس بر اساس جواب هایی که میگیرد تصمیماتی میتواند بگیرد و آن ها را انجام دهد به طور مثال اگر جواب نگرفت دوباره درخواست ار بفرستد و نتیجه جواب ها را ذخیره کند یا زمان تاخیر هر سرویس را محاسبه کند و کار های دیگر. در شکل زیر شمای کلی این سیستم را مشاهده میکنیم.



همانطور که در شکل آمده است Control Plane بخش های مختلفی دارد که یکی از بخش های مهم آن که ما با آن کار داریم بخش Prometheus است که اطلاعات را از پروکسی ها دریافت میکند و در اختیار ما میگذارد. بخش دیگری که ما استفاده میکنیم سیستمی به نام grafana است که ابزار قوی ای برای تصویر سازی داده ها است و با استفاده از این ابزار میتوانیم داده های جمع آوری شده از طریق Prometheus را به طرز مفید و زیبایی ببینیم. پس linkerd این دو ابزار مهم را هم برای ما نصب میکند و لازم نیست تا جداگانه این دو ابزار را نصب و راه اندازی کنیم. حال به سراغ نصب linkerd میرویم:

ابتدا برای آنکه این سیستم را نصب کنیم؛ باید command line مخصوص linkerd را نصب کنیم تا بتوانیم به linkerd دسترسی پیدا کنیم و عملیات های مختلف انجام دهیم. با دستور زیر میتوانیم این CLI را نصب کنیم.

```
curl -sL https://run.linkerd.io/install | sh
```

سپس باید به path با استفاده از دستور زیر اضافه کنیم.

```
export PATH=$PATH:$HOME/.linkerd2/bin
```

بعد از آنکه Linkerd به کلاستر ما دسترسی پیدا کرد و با استفاده از دستور زیر همه ی پیش نیاز های کلاستر ما را بررسی کرد و همه پیشنیاز ها آماده بود سراغ مرحله آخر که اضافه کردن پروکسی linkerd به سرویس ها است.

Linkerd check --pre

با استفاده از دستور زیر linkerd را نصب میکنیم:

linkerd install | kubectl apply -f -

بعد از نصب linkerd میتوانیم با دستور زیر این پروکسی را در سرویس خود اضافه کنیم.

kubectl get -n default deploy/worker-deployment -o yaml | linkerd inject - | kubectl apply -f -

و در شکل زیر اضافه شدن این پروکسی را در سرویس خود میتوانیم مشاهده کنیم.

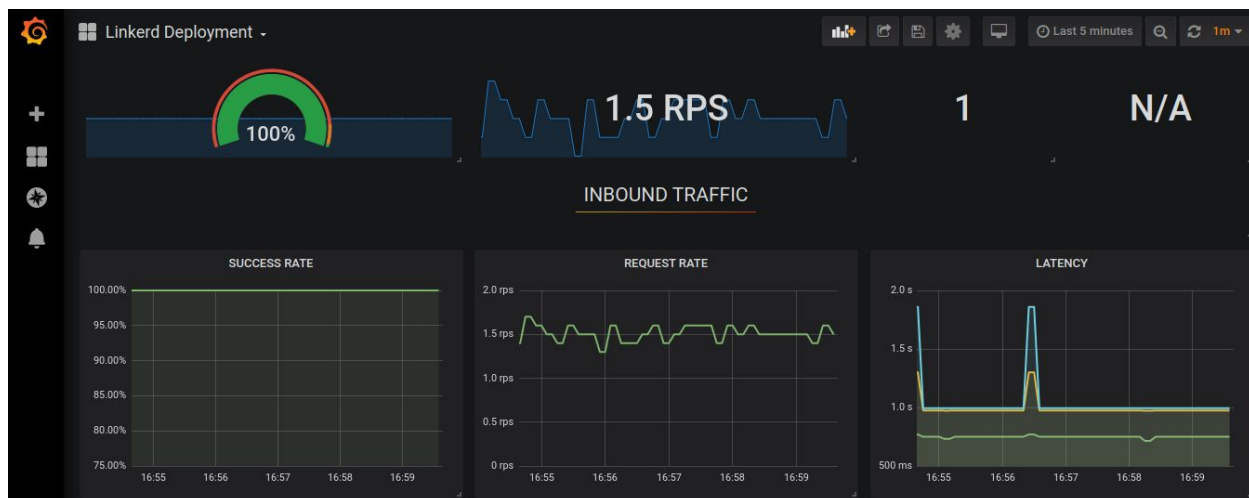
```
Successfully assigned default/worker-deployment-94d8b8f44-v282d to worker-node2
Container image "gcr.io/linkerd-io/proxy-init:v1.3.2" already present on machine
Created container linkerd-init
Started container linkerd-init
Pulling image "mpouyakh/multi-worker"
Successfully pulled image "mpouyakh/multi-worker"
Created container worker
Started container worker
Container image "gcr.io/linkerd-io/proxy:stable-2.7.1" already present on machine
Created container linkerd-proxy
Started container linkerd-proxy
```

Linkerd همچنین یک web UI دارد که از طریق آن میتوانیم سرویس های اضافه شده را مشاهده کنیم. در شکل زیر این قابلیت linkerd را هم مشاهده کنیم.

با دستور `linkerd dashboard --port 30000` میتوانیم این داشبورد را بالا بیاوریم.

Deployment ↑	↑ Meshed	↑ Success Rate	↑ RPS	↑ P50 Latency	↑ P95 Latency	↑ P99 Latency	Grafana
<a href="#">position-tracker</a>	1/1	100.00% ●	1.33	792 ms	1.65 s	1.93 s	🔧
<a href="#">pouya-nginx-ingress-controller</a>	2/2	100.00% ●	0.4	2 ms	4 ms	4 ms	🔧
<a href="#">api-gateway</a>	1/1	---	---	---	---	---	🔧

با استفاده از `linkerd` میتوان میزان تاخیر هر سرویس به علاوه اینکه چه میزان درخواست در یک ثانیه دریافت میکند را مشاهده کنیم و از طریق `grafana` میتوان تاریخچه این میزان درخواست ها و تاخیر ها را مشاهده کرد که در شکل زیر خواهیم دید.



سه نوع تاخیری که وجود دارد به این منظور است که مشخص کند که این تاخیر برای چند درصد از مواقع درست است و وقتی تاخیر P99 است؛ این به این منظور است که تنها یک درصد درخواست ها تاخیرشان پایین این تاخیر نوشته شده است و P95 هم به همین شکل یعنی فقط ۵ درصد درخواست ها تاخیرشان از این میزان کمتر از این تاخیر ذکر شده است.

## - معیار های تأخیر در پاسخ (Response latency) و تعداد درخواست در یک ثانیه (RPS)

حال می‌خواهیم بر اساس یک سری معیار های مفید تر دیگری کار مقیاس پذیری را انجام دهیم. درست است که معیار محاسباتی و حافظه ای تا حدودی اطلاعات خوبی را به ما درباره اینکه چقدر بار بر روی یک سرویس ما گذاشته میشود؛ ولی خیلی دقیق نیست و وقتی میزان مصرفی cpu بالا میرود شاید دلایل دیگری برای این بالا رفتن بار وجود دارد و تنها بار اضافه شده بر روی سرویس ما نیست. ممکن است این بالا رفتن cpu به دلیل این است که بعضی از بار های ما Memory bound و IO bound هستند. همچنین cpu ها مختلف هستند و هر کدام ممکن است مقدار مصرفی که نشان دهند متفاوت باشد و این معیار نسبی هست و خیلی دقیق نیست. حال معیاری که میتوانیم رویش حساب کنیم و مقیاس پذیری را انجام دهیم معیار تأخیر در جواب (response latency) یک سرویس است که به ما نشان میدهد چه مدت طول میکشد تا سرویس ما به کاربر جواب بدهد. شرکت ها و کمپانی های بزرگ همه در صدد کم کردن میزان این تأخیر هستند تا بتوانند به رضایت مشتری را جذب کنند. پس این معیار بسیار اساسی و مهم است. شما اگر یک سرویس با شکل و شمایل بسیار زیبا هم داشته باشید ولی زمان پاسخ زیاد باشد؛ مشتری راضی نخواهد بود و امکان از دست دادن آن مشتری زیاد است.

معیار دیگری که میتواند مفید باشد تعداد درخواست هایی که در یک ثانیه (RPS) به سرویس ما وارد میشود چقدر است و بر اساس چه میزان بار ما آن تأخیر را دریافت میکنیم. همچنین بر اساس این معیار هم میتوانیم کار مقیاس پذیری را هم انجام دهیم.

حال بعد از آنکه این دو معیار معرفی شد؛ سراغ استخراج این دو معیار با استفاده از Prometheus adapter میرویم که همان طور در بخش های قبل گفته شد مسئول گرفتن اطلاعات از Prometheus است و مرتب کردن اطلاعات به گونه که ما در کانفیگ فایل مشخص کردیم و سپس با در دسترس قرار دادن اطلاعات از طریق API مربوطه برای HPA؛ کار مقیاس پذیری را انجام میدهیم.

کانفیگ فایل به صورت زیر است :

```
1 prometheus:
2 | url: http://linkerd-prometheus.linkerd.svc
3
```

در بخش اول ما سرویس prometheus خود را معرفی میکنیم تا این adapter بتواند اطلاعات خود را از آن بگیرد.



```

4 rules:
5   default: false
6   custom:
7     - seriesQuery: 'response_latency_ms_bucket{namespace!="",pod!=""}'
8       resources:
9         template: <<.Resource>>
10        name:
11          matches: ^(.*)_bucket$
12          as: "${1}_50th"
13        metricsQuery: histogram_quantile(0.50, sum(irate(<<.Series>>{<<.LabelMatchers>>, direction="inbound"}[5m])) by (le, <<.GroupBy
14

```

در این بخش ما قوانین ما rule های خود را مینویسیم و معیاری که می‌خواهیم را مشخص می‌کنیم. همچنین این اطلاعات به چه صورت به ما داده شود را مشخص می‌کنیم. با استفاده از بخش seriesQuery مشخص می‌کنیم که کدام معیار را از Prometheus می‌خواهیم استخراج کنیم و می‌توانیم اسم این معیار را با استفاده از بخش as تغییر می‌دهیم. در بخش metricsQuery ما با استفاده از تابع histogram\_quantile می‌توانیم تاخیر p50 را همان طور که در بخش قبل اشاره کردیم؛ می‌توانیم محاسبه کنیم. با استفاده از تابع sum هم این داده ها را در بازه زمانی ۵ دقیقه جمع می‌کنیم و به تابع histogram\_quantile به عنوان ورودی می‌دهیم که تاخیر را محاسبه کند. برای تاخیر های دیگر هم به همین شکل عمل می‌کنیم و فقط باید اسم معیار را تغییر بدهیم.

برای معیار تعداد درخواست در یک ثانیه هم به همین گونه ای که در شکل نشان داده شد عمل می‌کنیم و فقط تابع histogram\_quantile را ندارد.

```

- seriesQuery: 'request_total{namespace!="",pod!=""}'
  resources:
    template: <<.Resource>>
  name:
    matches: ^(.*)_total$
    as: "${1}s_per_second"
  metricsQuery: |-
    sum(
      irate(
        <<.Series>>{
          <<.LabelMatchers>>,
          direction="inbound"
        }[5m]
      )
    ) by (
      <<.GroupBy>>
    )

```

بعد از آنکه این کانفیگ فایل را نوشتیم حال سراغ نصب Prometheus adapter با استفاده از پکیج منیجر Helm می‌رویم که دستورش به شکل زیر است.

```
helm --namespace linkerd install stable/prometheus-adapter -f  
hpa/prometheus-adapter.yml
```

این adapter را در همان جایی که بخش های مختلف linkerd نصب شده است؛ نصب میکنیم و فایل کانفیگ را هم در هنگام نصب به این adapter میدهیم.

حال نگاهی میندازیم به مقادیری که در custom metrics api که HPA از این طریق میتواند به معیار ها دسترسی پیدا کند؛ وجود دارد. با استفاد از دستور زیر میتوانیم این مقادیر را نگاه کنیم.

```
kubectl get --raw /apis/custom.metrics.k8s.io/v1beta1
```

همان طور که در شکل زیر معلوم است در API میتوانیم اسم این معیار ها را ببینیم و همچنین مقادیرشان را مشاهده کنیم. که در شکل زیر بعضی از این معیار ها آمده است.

```
{  
  "name": "jobs.batch/response_latency_ms_99th",  
  "singularName": "",  
  "namespaced": true,  
  "kind": "MetricValueList",  
  "verbs": [  
    "get"  
  ]  
},  
{  
  "name": "pods/response_latency_ms_99th",  
  "singularName": "",  
  "namespaced": true,  
  "kind": "MetricValueList",  
  "verbs": [  
    "get"  
  ]  
},  
{  
  "name": "jobs.batch/requests_per_second",  
  "singularName": "",  
  "namespaced": true,  
  "kind": "MetricValueList",  
  "verbs": [  
    "get"  
  ]  
}
```

حال بعد از اینکه توانستیم این معیار ها را دسترس پذیر کنیم؛ سراغ نوشتن فایل Yaml برای خودکار ساز خود میکنیم.

```

1  apiVersion: autoscaling/v2beta1
2  kind: HorizontalPodAutoscaler
3  metadata:
4    name: pouya-nginx-ingress-controller
5    namespace: default
6  spec:
7    scaleTargetRef:
8      apiVersion: apps/v1
9      kind: Deployment
10     name: pouya-nginx-ingress-controller
11   minReplicas: 1
12   maxReplicas: 10
13   metrics:
14   - type: Pods
15     pods:
16       metricName: requests_per_second
17       targetAverageValue: 20
18   - type: Pods
19     pods:
20       metricName: response_latency_ms_99th
21       targetAverageValue: 500

```

برای این که ما بتوانیم از custom metrics استفاده کنیم؛ از API شده در شکل بالا استفاده میکنیم و این API هم به custom metrics متصل میشود و اطلاعات معیار ها را دریافت میکند. سپس سرویسی را که میخواهیم مقیاس پذیر کنیم را مشخص میکنیم و API ای که از طریق آن در دسترس هست را مینویسیم که همان apps/v1 است. در مرحله بعد رینج تعدادی که این سرویس ما میتواند مقیاس پذیر شود را مشخص میکنیم که از تعداد یک تا ۱۰ است. با توجه به اینکه در منابع محاسباتی و حافظه ای محدودیت داریم بیشتر از این تعداد سیستم ما دچار مشکل میشود و کند میشود. در مرحله آخر معیار هایی که میخواهیم بر اساس آنها مقیاس پذیری را انجام دهیم را معین میکنیم و سپس حدی را که اگر از آن گذشت کار مقیاس پذیری را انجام دهد را تعیین میکنیم. این مقادیر باید بر حسب آنالیز هایی که بر روی سرویس ها میشود تعیین شود. یعنی باید ببینیم در زیر بار های مختلف چه عملکردی دارد و میزان تاخیر پاسخ چقدر است و تا چه میزان تاخیر مناسب و مورد قبول است. در اینجا ما تا ۵۰۰ میلی ثانیه را زمان مناسبی برای تاخیر در پاسخ سرویس خود در نظر گرفته ایم و اگر از این حد گذشت؛ خودکار ساز ما شروع به افزایش تعداد پاد میکند تا این تاخیر کمتر شود. همچنین معیار دیگری که تعداد درخواست در ثانیه است را هم در نظر گرفته ایم که اگر از این تعداد درخواست در یک ثانیه برای یک پاد بیشتر شد این بار با پاد های اضافه شده تقسیم کند تا عملکرد بهتری از نظر پاسخ دهی داشته باشد.

## - نتایج خودکار ساز

حال نتایج این خودکار ساز را برای یک سرویس خود مورد بررسی قرار میدهیم. این سرویس همان وب سرور ما است به نام pouya-nginx-ingress-controller که درخواست ها را بین سرویس های مختلف ما پخش میکند. این سرویس نقش مهمی را در سیستم ما ایفا میکند و اگر درست کار نکند؛ کاربران ما نمیتوانند به بقیه سرویس های ما دسترسی پیدا کنند. پس

ما باید اطمینان حاصل کنیم که تاخیر این سرویس در حد قابل قبولی است و کار خود را به موقع انجام میدهد. حال وضعیت این سرویس را قبل از اعمال بار نگاه میکنیم که به صورت زیر است.

REFERENCE	TARGETS	MINPODS	MAXPODS
Deployment/position-tracker	1400m/40	1	10
Deployment/pouya-nginx-ingress-controller	204m/40, 6465m/500	1	10
Deployment/webapp	0/40	1	10

حال با بار پنج کاربر همزمان که درخواست می فرستند شروع میکنیم که با استفاده از load generator ای که در بخش قبل استفاده کردیم این بار را تولید میکنیم.

ابتدا یک توضیحی در مورد اعداد این خودکار ساز بدهیم که عدد ۴۰ در روبروی سرویس ما به این معنی است که اگر از ۴۰ درخواست بر ثانیه بیشتر شد؛ این خودکار ساز پاد ها را افزایش دهد. و عدد ۲۰۴m هم یعنی ۲۰۴m requests که برابر ۰.۲ درخواست بر ثانیه است. پس یک درخواست بر ثانیه برابر ۱۰۰۰m است. همچنین کوبرنتیز برای اینکه با اعداد اعشاری کار نکند اعداد را در هزار ضرب میکند. عددی که در کنار عدد ۵۰۰ نوشته شده است را باید بر ۱۰۰۰ تقسیم کنیم که برابر ۰.۵ میلی ثانیه است.

REFERENCE	TARGETS	MINPODS	MAXPODS
Deployment/position-tracker	1400m/40	1	10
Deployment/pouya-nginx-ingress-controller	499m/40, 586750m/500	1	10

بعد از اعمال این بار میبینیم که میزان پاسخ دهی بیشتر از حد مجاز شده است و باید این خودکار ساز ما تعداد پاد ها بیشتر کند تا این تاخیر کمتر شود.

TARGETS	MINPODS	MAXPODS	REPLICAS
1400m/40	1	10	1
699m/40, 457366m/500	1	10	4
0/40	1	10	1

همان طور که میبینیم این خودکار ساز تعداد پاد ها را افزایش داده و میزان تاخیر کمتر از حد مجاز شده است. در شکل عملکرد هر پاد را جداگانه مشاهده میکنیم.

pouya-nginx-ingress-controller-59f4fdb9dd-9jsr	1/1	100.00% ●	0.42	4 ms	190 ms	198 ms
pouya-nginx-ingress-controller-59f4fdb9dd-djdl	1/1	100.00% ●	0.47	5 ms	490 ms	498 ms
pouya-nginx-ingress-controller-59f4fdb9dd-k9m7f	1/1	100.00% ●	0.52	25 ms	187 ms	198 ms
pouya-nginx-ingress-controller-59f4fdb9dd-rfh22	1/1	100.00% ●	0.65	425 ms	493 ms	499 ms

میزان تاخیرهای P99 , 95P و P50 را برای این پاد ها در شکل بالا آمده است و همچنین چه میزان خطا در پاسخ دهی داشتند که در این شکل خطایی در پاسخ دهی نبوده و وضعیت پاسخ دهی ۱۰۰ درصد است. همچنین تعداد درخواست برای هر پاد مشخص است که در ردیف چهارم از سمت راست آمده است. تولید کننده بار ما اطلاعات مفیدی را در رابطه با این سرویس در اختیار ما قرار داده است که در شکل زیر میتوانیم مشاهده کنیم.

```
root@mpouyakh-m:/home/mpouyakh# siege -c 5 -t 20 http://mpouyakh.com
** SIEGE 4.0.4
** Preparing 5 concurrent users for battle.
The server is now under siege...^C
Lifting the server siege...
Transactions:          1016 hits
Availability:          99.03 %
Elapsed time:          805.41 secs
Data transferred:     125.85 MB
Response time:         3.95 secs
Transaction rate:      1.26 trans/sec
Throughput:            0.16 MB/sec
Concurrency:           4.99
Successful transactions: 1016
Failed transactions:    10
Longest transaction:   15.87
Shortest transaction:   0.58
```

همانطور که در شکل بالا مشاهده میکنیم؛ تعداد کل درخواست هایی که فرستاده؛ چه میزان سرویس ما در دسترس بوده که برابر ۹۹ درصد است و تاخیری که در اینجا مشاهده میکنیم مربوط به شبکه ما میشود و به دلیل آنکه Throughput ما کم است و اینترنت ما ضعیف است این تاخیر زیاد شده است و هر چقدر اینترنت ما قوی تر باشد این تاخیر به تاخیر پاد ما نزدیک تر است. همچنین مقدار داده که به این پاد ها منتقل کرده است و مواردی دیگری که در شکل میتوان مشاهده کرد. حال تعداد کاربرانی که همزمان درخواست میدهیم را به بیست افزایش میدهیم و وضعیت خودکار سار خود و پاد ها مشاهده کنیم.

REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS
Deployment/position-tracker	1400m/40	1	10	1
Deployment/pouya-nginx-ingress-controller	644m/40, 1056397m/500	1	10	10
Deployment/webapp	0/40	1	10	1

همانطور که مشاهده میکنیم برای این بار؛ خودکار ساز ما تعداد پاد ها را به ده افزایش داده است و تاخیر حدود یک ثانیه است و مطلوب ما نیست. در شکل زیر وضعیت پاد ها را هم میتوانیم ببینیم که بعضی از پاد ها میزان تاخیر زیادی دارند و این بیانگر این است که تعداد پاد بیشتری مورد نیاز است تا این تاخیر کمتر شود.

pouya-nginx-ingress-controller-59f4fdb9dd-7wcl8	1/1	96.88% ●	0.53	4 ms	1.85 s	1.97 s	🔧
pouya-nginx-ingress-controller-59f4fdb9dd-dfhrd	1/1	100.00% ●	0.33	10 ms	185 ms	197 ms	🔧
pouya-nginx-ingress-controller-59f4fdb9dd-djdw1	1/1	96.55% ●	0.48	3 ms	1.85 s	1.97 s	🔧
pouya-nginx-ingress-controller-59f4fdb9dd-hlt4d	1/1	96.30% ●	0.45	3 ms	900 ms	980 ms	🔧
pouya-nginx-ingress-controller-59f4fdb9dd-kqkxw	1/1	100.00% ●	0.45	3 ms	180 ms	196 ms	🔧
pouya-nginx-ingress-controller-59f4fdb9dd-m4ftp	1/1	100.00% ●	0.28	3 ms	4 ms	4 ms	🔧
pouya-nginx-ingress-controller-59f4fdb9dd-pj5zb	1/1	86.21% ●	0.48	1.25 s	2.75 s	2.95 s	🔧

همان طور از شکل بالا هست هم میزان در دسترس بودن پایین آمده است و هم تاخیر در بعضی پاد ها بالا رفته است. در شکل زیر هم وضعیت کلی این دوره از اعمال بار را مشاهده میکنیم.

```

root@mpouyakh-m:/home/mpouyakh# siege -c 20 -t 30 http://mpouyakh.com
** SIEGE 4.0.4
** Preparing 20 concurrent users for battle.
The server is now under siege...^C
Lifting the server siege...
Transactions:          1704 hits
Availability:          78.17 %
Elapsed time:          784.77 secs
Data transferred:     191.59 MB
Response time:         9.03 secs
Transaction rate:      2.17 trans/sec
Throughput:            0.24 MB/sec
Concurrency:           19.60
Successful transactions: 1704
Failed transactions:    476
Longest transaction:   49.24
Shortest transaction:  1.24

```

در این شکل هم مشخص است که میزان دسترس پذیری سرویس ما کم شده است و حدود ۸۰ درصد است مه مطلوب ما نیست. حال تعداد پاد ها را به ۱۵ افزایش میدهیم و ببینیم که آیا نتیجه بهتر میشود یا خیر. حال بعد از اعمال این بار به مدت نیم ساعت با ۲۰ کاربر همزمان که درخواست میفرستند؛ نتایج در شکل های زیر بررسی میکنیم.

REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS
Deployment/position-tracker	1400m/40	1	10	1
Deployment/pouya-ingress-controller	546m/40, 553909m/500	1	15	15
Deployment/webapp	0/40	1	10	1

بعد از مدتی که پاد ها اضافه شد و در شرایط استقرار قرار گرفتند؛ مشاهده میکنیم که این تاخیر کمتر شده و نزدیک ۵۰۰ میلی ثانیه شده است البته بعضی مواقع تا ۷۰۰ میلی ثانیه هم میرفت ولی دیگر به یک ثانیه نرسید. همینطور این بهبود نتیجه در اطلاعات تولید کننده بار هم مشهود است که در شکل زیر با هم میبینیم.



```

root@mpouyakh-m:/home/mpouyakh# siege -q -c 20 -t 30 http://mpouyakh.com
^C
Lifting the server siege...root@mpouyakh-m:/home/mpouyakh# siege -c 20 -t 30 http://mpouyakh.com
** SIEGE 4.0.4
** Preparing 20 concurrent users for battle.
The server is now under siege...^[A
Lifting the server siege...
Transactions:          4909 hits
Availability:          84.16 %
Elapsed time:          1799.09 secs
Data transferred:     582.39 MB
Response time:         7.27 secs
Transaction rate:      2.73 trans/sec
Throughput:            0.32 MB/sec
Concurrency:           19.84
Successful transactions: 4909
Failed transactions:   924
Longest transaction:   43.88
Shortest transaction:  1.09

```

با توجه به این شکل؛ مشاهده میکنیم که میزان دسترس پذیر بودن ما بیشتر از قبل شده و همچنین زمان پاسخ سرویس ما با وجود آنکه مدت بیشتری از سری قبل زیر بار بود که حدوداً ۵ دقیقه بود؛ کمتر است. همچنین چون اینترنت ما سرعتش بیشتر شده میبینیم که تاخیر هم به نسبت کمتر شده است و هر چه میزان گزردهی ما بیشتر باشد این تاخیر هم به تاخیر پاد ما نزدیک میشود.