**فصل پنجم: روش های خودکار سازی در پلتفرم کوبرنتیز**

در این فصل ما انواع روش های خودکار سازی در کوبرنتیز را معرفی میکنیم و توضیح میدهیم. ولی روشی که ما برای خودکارسازی در این پروژه استفاده میکنیم روش سوم که خودکار سازی سرویس ها است را استفاده میکنیم. روش اول به دلیل آنکه منابع سخت افزاری ما محدود است؛ امکان استفاده از این روش وجود ندارد و روش دوم به دلیل آنکه با روش سوم تداخل دارد و در حال حاضر این امکان وجود ندارد که روش دوم و سوم را با هم استفاده کنیم؛ نمیتوانیم از روش دوم در این پروژه بهره ببریم.

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

یکی از ابزار کوبرنتیز(Kubernetes) است که به ما این اجازه را میدهد سایز کلاستر خود را که شامل تعدادی گره 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 گره به کلاستر اضافه میکند. و هنگامی که منابع ما از یک حدی به بعد مورد استفاده قرار نگرفتند ؛ گره 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 گره 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 گره 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 گره 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 کوبرنتیز ما با استفاده از PodDistruptionBugdet می توانیم 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 قصد این را داشت که گره ای 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 یا 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 گره استفاده میکنیم و این سه گره 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.

2 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 خودکارساز 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:

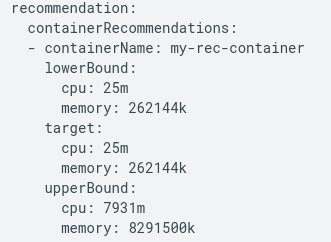
./hack/vpa-up.sh

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



شکل۱ - فایل Yaml برای ساختن خودکار ساز

همانطور که در شکل بالا مشاهده می کنیم 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.



شکل۲ - خروجی خودکارساز تخصیص منابع (VPA)

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.



شکل۳ - اعمال مقیاس پذیری تخصیص منابع برای پاد مد نظر

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 خودکارساز سرویس ها هنوز سازگار نیست و نمی توان 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 با هم 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 معیارهای محاسباتی و حافظه ای استفاده کنیم؛ در نتیجه از این خودکار ساز در پروژه استفاده نخواهیم کرد.

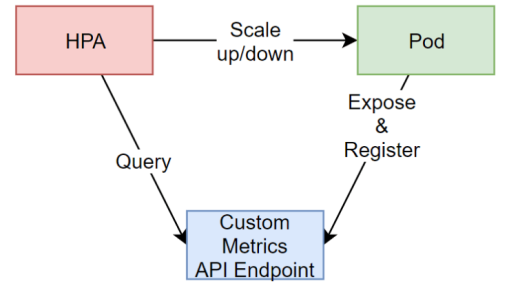
## 3 خودکار سازی سرویس ها (HPA):

3-1 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 پادهای 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.



شکل۴ - نحوه عملکرد خودکار ساز

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.

desiredReplicas = ceil[currentReplicas \* ( currentMetricValue / 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 ار 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.

3-1-1 نصب Metrics server

برای راه اندازی metrics-server از پکیج منیجر Helm استفاده میکنیم که این پکیج منجیر 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.

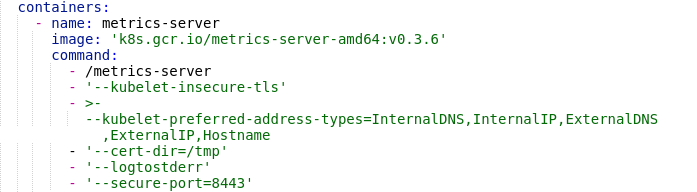
ابتدا برای نصب این پکیج باید یک سری مقادیر را تغییر دهیم به این منظور که در حین نصب دچار مشکل نشویم. دستور زیر را وارد میکنیم و مقادیر را تغییر میدهیم.

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

دو مقدار زیر باید به این گونه باشند.

الف) hostNetwork: true

ب)



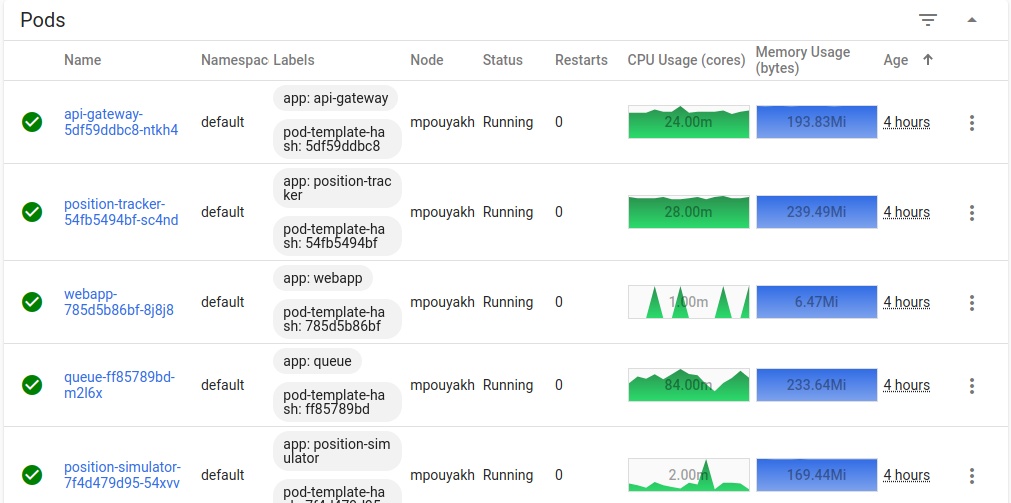
شکل۵ - اضافه کردن پارامترهای اضافی به فایل Yaml سرویس Metrics server

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

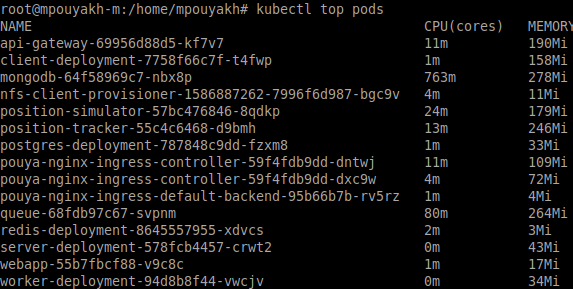
بعد از این تغییرات نوبت به نصب این پکیج میرسد که با استفاده از دستور زیر metrics server را نصب میکنیم.

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

حال میتوانیم با استفاده از داشبورد 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 داشبورد و command line آمده است.



شکل۶ - نمای کلی از نمایش معیار های محاسباتی و حافظه ای در داشبورد کوبرنتیز



شکل۷ - نمایش معیار های محاسباتی و حافظه ای در ترمینال

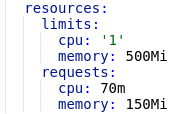
همانطور که در شکل های بالا مشاهده می کنیم میزان مصرفی هر پاد مشخص است که برای قسمت cpu منظور از m ؛ milicore است که هر یک cpu برابر است باm 1000.

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 ؛ نمی توانیم 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.

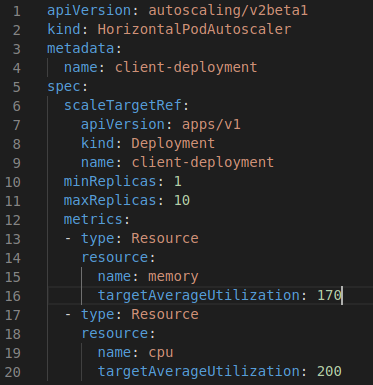
حال با بررسی مقدار مصرفی این پاد و منابعی که در اختیار داریم ما درخواست ها و محدودیت های زیر را برای این پاد در نظر گرفته ایم.



شکل۸ - تعیین درخواست های محاسباتی و حافظه ای برای سرویس مد نظر

3-1-2 ساختن خودکار ساز برای metrics server

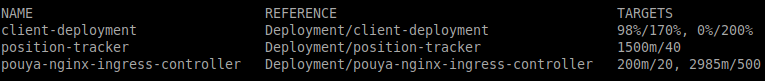
سپس فایل Yaml این خودکار ساز را نوشتیم که به صورت زیر است:



شکل۹ - فایل Yaml برای ساختن خودکار ساز

همانطور که مشخص است در خط اول ابتدا API مد نظر را مشخص کردیم که با استفاده از این API به این خودکار ساز می توانیم 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.



شکل۱۰ - نتیجه خودکارساز برای Metrics server

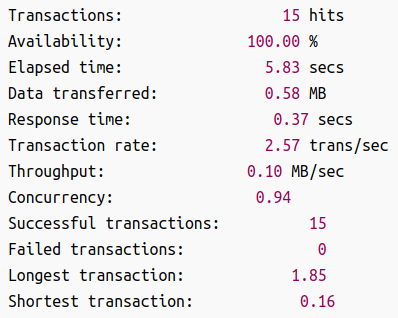
در خط اول این شکل مشاهده می کنیم که در قسمت Targets دو معیار ما آورده شده اند و تعداد Replica برابر ۱ است چون مقدار فعلی کمتر از مقدار تعیین شده است. بعد از آنکه بار بر روی این پاد اضافه کردیم مشاهده می کنیم به صورت خودکار؛ کوبرنتیز تعداد پاد ها را افزایش خواهد داد و سعی میکند مقدار فعلی را به زیر مقدار 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.

3-2 تولید بار (Load Generator):

حال سراغ تولید بار می رویم که برای این کار ما از ابزار siege استفاده میکنیم. برای اینکه بتوانیم تشخیص دهیم که توانایی وب اپلیکیشن یا سرویس ما چقدر است و چه میزان توانایی پاسخگویی دارد باید بر روی آن بار تولید کنیم. با استفاده از 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) میدهیم و به طور مثال۲۰ 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. و نتایج این تولید کننده بار برای مثال به صورت زیر است.



شکل۱۱ - نتیجه تولید کننده بار siege

در قسمت اول نشان میدهد که چه میزان در کل درخواست انجام شده است. در قسمت دوم میزان دسترس پذیر بودن سرویس ما را نشان میدهد که چه میزان از این درخواست ها جواب ۲۰۰ گرفته شده است. بخش مهمی که باید در این بخش در رابطه با این تولید کننده بار ذکر کرد؛ میانگین زمان پاسخ کل درخواست ها است (Response time) که در قسمت پنجم شکل آمده است و بخش Concurrency تعداد کاربری که همزمان به سرویس ما متصل هستند را نشان میدهد. بخش آخر به طولانی ترین و کوتاه ترین درخواست های ما از کل درخواست هایی که فرستاده شده است که در دو قسمت آخر آمده است؛ اشاره میکند.

همچنین این تولید کننده بار این امکان را به ما داده است که بتوانیم با استفاده از کانفیگ فایل این تولید کننده بار که در این مسیر است) siege/siege.conf. (~/؛ قابلیت های مختلف این ابزار را استفاده کنیم. به طور مثال چند نمونه از قابلیت های مهم این ابزار را در قسمت زیر اشاره میکنیم.

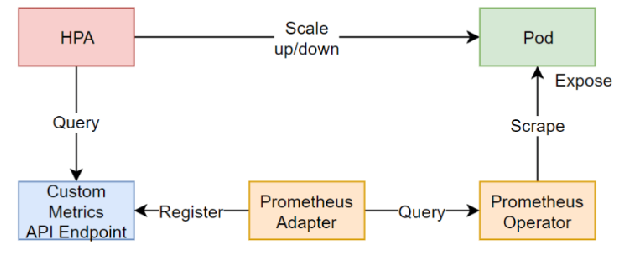
اگر ما در این کانفیگ فایل بخش parser را فعال کنیم یعنی مساوی true قرار دهیم؛ این تولید کننده بار فقط یک درخواست HTTP به همان مسیری که مشخص کردیم میدهد و دیگر بقیه بخش های این وب پیج را که شامل فایل های Javascript و CSS و عکس ها است؛ درخواست نمیدهد. همچنین می توانیم محدودیت تعداد کاربر همزمان را که به طور پیش فرض ۲۵۵ است را تغییر دهیم و افزایش دهیم. در بخش دیگر این کانفیگ فایل میتوانیم با استفاده از فعال کردن قسمت verbose ؛ نتیجه درخواست ها را در command line خود مشاهده کنیم. این تولید کننده بار بین درخواست ها به طور پیش فرض بین یک تا سه ثانیه به طور رندم فاصله ایجاد میکند که در بخش delay میتوانیم این زمان را افزایش دهیم. ولی برای آنکه ما عملکرد سرویس خود را بهتر تست کنیم و توان سرویس خود را ارزیابی کنیم این تاخیر را صفر در نظر گرفتیم.

3-3 Custom metrics

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 محاسباتی و حافظه ای پاد ها را استخراج کنیم و از طریق Resource metrics 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 Promethues 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 اولیکیشن هستند و باید در اپلیکیشن تنظیم شوند و آن معیار ها را در اختیار Prometheus قرار دهند.

شکل ۱۲ - نحوه ارتباطات اجزا مختلف برای استخراج Custom metrics

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 بخش های مختلف چگونه با هم ارتباط برقرار میکنند. همانطور که مشاهده می شود؛ Prometheus معیار ها را از پاد ها استخراج میکند و به صورت زمانی میتواند به ما نشان دهد. سپس Prometheus adapter این معیار ها را به گونه ای که ما میخواهیم در میاورد و از طریق API مشخص شده در شکل برای HPA در دسترس قرار میدهد.

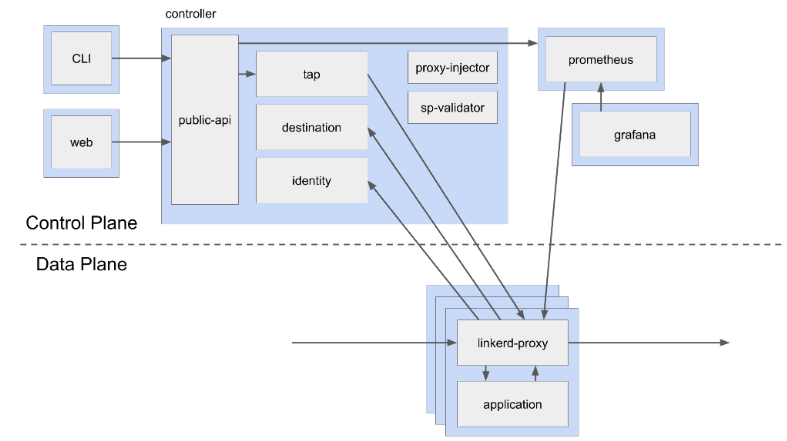
حال به سراغ مرحله بعد می رویم که در صفحه قبل اشاره کردیم. برای اینکه بعضی از معیار ها را بتوانیم تعریف کنیم و بر اساس آنها مقیاس پذیری را انجام دهیم باید کد اپلیکیشن خود را به گونه ای تغییر دهیم که بتواند این معیار ها را در اختیار Prometheus بگذارد. ولی ما با استفاده از Linkerd که یک Service Mesh برای کوبرنتیز است؛ میخواهیم بدون آنکه تغییری در سرویس های بدهیم بعضی از معیار های مفید را برای مقیاس پذیری استفاده کنیم.

3-3-1 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 پروکسی هایی است که در کنار هر سرویس قرار میگیرد است و مسئول ارتباطات این پروکسی ها است. مسئولیت های دیگر این بخش مدیریت؛ جمع آوری اطلاعات از پروکسی ها؛ فراهم کردن ارتباطات بر اساس تکنولوژی TLS و فراهم کردن API هایی برای ادمین کلاستر است که بتواند داده هایی که از این پروکسی ها جمع آوری شده دسترسی پیدا کند و بتواند تغییراتی را در این پروکسی ها اعمال کند.

بخش دیگر این سیستم؛ data plane است که مربوط به پروکسی هایی است که در کنار سرویس های ما قرار میگیرد 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 را 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.



شکل۱۳ - ساختار کلی سیستم Linkerd

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 Promethues است که اطلاعلات 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 انکه 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

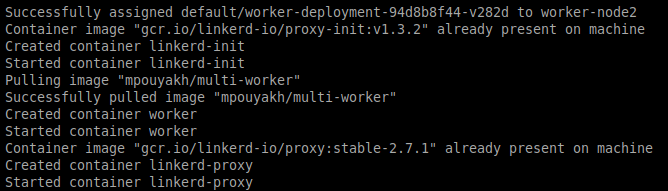
با استفاده از دستور زیر linkerd را نصب میکنیم:

linkerd install | kubectl apply -f -

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

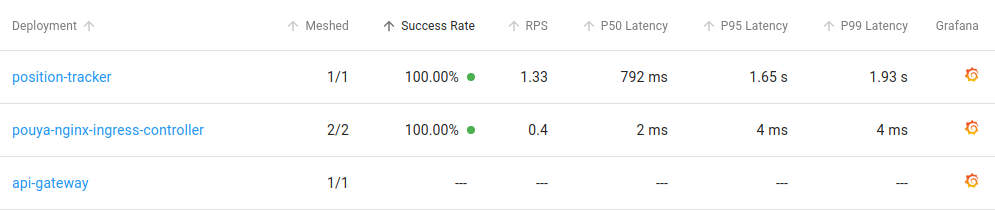
kubectl get -n default deploy/worker-deployment -o yaml | linkerd inject - | kubectl apply -f -

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

شکل۱۴ - اضافه شدن پروکسی linkerd به پاد مد نظر

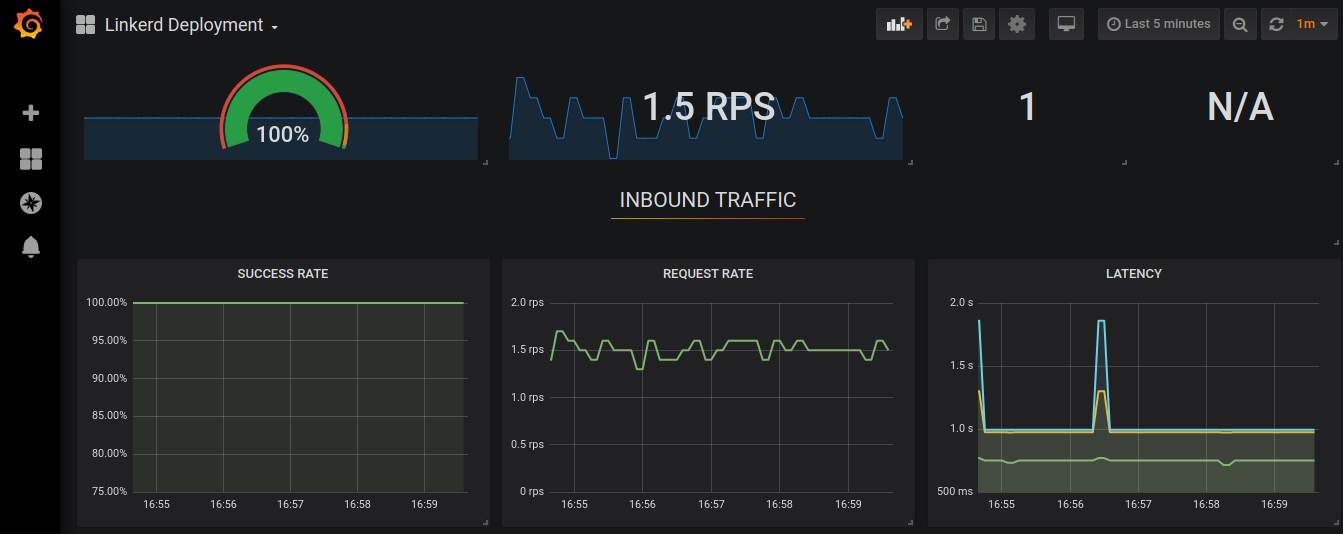
Linkerd همچنین یک web UI دارد که از طریق آن میتوانیم سرویس های اضافه شده را مشاهده کنیم. در شکل زیر این قابلیت linkerd را هم مشاهده کنیم.

با دستور "linkerd dashboard -- port 30000 "میتوانیم این دشبورد را بالا بیاوریم.



شکل۱۵ - نمای داشبورد Linkerd

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.

شکل۱۶ - نمای دشبورد Grafana

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.

3-3-2 معیار های تاخیر 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 باورهای 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 است و خیلی دقیق نیست. حال معیاری که میتوانیم رویش حساب کنیم و مقیاس پذیری را انجام دهیم معیار تاخیر در جواب (response latency) یک سرویس است که به ما نشان می دهد چه مدت طول می کشد تا سرویس ما به کاربر جواب بدهد. شرکت ها و کمپانی های بزرگ همه در صدد کم کردن میزان این تاخیر 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 تاخیر را دریافت میکنیم. همچنین بر اساس این معیار هم میتوانیم کار مقیاس پذیری را هم انجام دهیم.

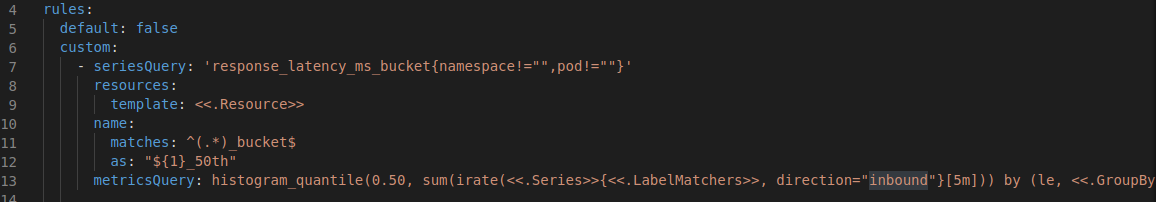
حال بعد از آنکه این دو معیار معرفی شد؛ سراغ استخراج این دو معیار با استفاده از Prometheus adapter میرویم که همان طور در بخش های قبل گفته شد مسئول گرفتن اطلاعات از Prometheus است و مرتب کردن اطلاعات به گونه که ما در کانفیگ فایل مشخص کردیم سپس با در دسترس قرار دادن اطلاعات از طریق API مربوطه برای HPA؛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

کانفیگ فایل به صورت زیر است :



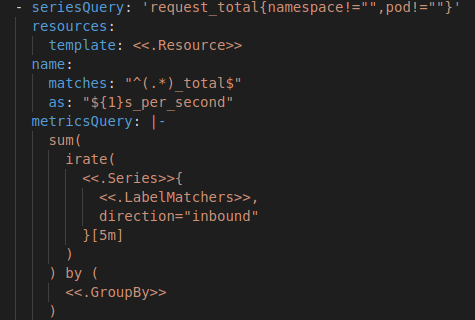
شکل۱۷ - معرفی سرویس Prometheus به Prometheus adapter

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

شکل۱۸ - بخش مشخص کردن معیار های مد نظر در کانفیگ فایل Promethues adapter

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 seriesQuesry 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 historgram\_quantile میتوانیم تاخیر p50 را همانطور 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.



شکل۱۹ - تعیین معیار درخواست در ثانیه و مشخص کردن نحوه دریافت اطلاعات این معیار

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 بیندازیم 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.

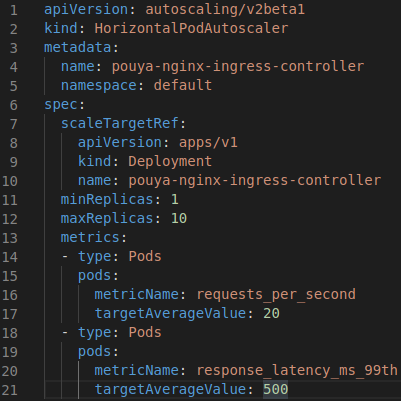
kubectl get --raw /apis/custom.metrics.k8s.io/v1beta1

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.



شکل۲۰ - مقادیر Custom metrics api

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



شکل۲۱ - فایل Yaml برای ساختن خودکارساز

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 API 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 می نویسیم 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؛ امکان آنکه تعداد بسیار بالا بگذاریم وجود ندارد و تا ۳۰ پاد سیستم ما به خوبی کار میکند و بیشتر منابع حافظه ای و محاسباتی ما اجازه تولید پاد های جدید را نخواهند داد. 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 تاخیر بر حسب رویکرد تیم فنی یک سازمان باید تعیین شود و مشخص کنند که چه میزان تاخیر برای سرویسشان مناسب است و از لحاظ رقابتی جه میزان تاخیر مناسب است. در اینجا ما برای تست 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.

3-3-3 نحوه انجام تست برای custom metrics

برای تست و بررسی این خودکار ساز بر اساس این معیار ها؛ میخواهیم زیاد شدن بار و افزایش تاخیر و سایر معیار هایی را که از تولید کننده بار دریافت می کنیم را مورد تحلیل و بررسی قرار دهیم؛ که آیا با افزایش تعداد پاد ها توسط خودکار ساز؛ کمکی به کم شدن این تاخیر خواهد کرد و همچنین تعداد بیشتری درخواست را این سرویس ما می تواند پاسخ بدهد. برای این تست ما چند سناریو مشخص را بر اساس محدودیت های نرم افزاری و سخت افزاری که داریم؛ مشخص کردیم. با استفاده از تولید کننده بار خود؛ برای تعداد کاربر ۳۰۰ ؛ ۴۰۰؛ ۵۰۰ و ۷۰۰ میخواهیم رفتار سیستم خود را تحلیل کنیم. و به علاوه در هر مرحله تعداد ماکزیمم پاد ها را برای خودکار ساز محدود میکنیم تا بهتر بتوانیم نسبت افزایش پاد و کم شدن تاخیر و سایر معیار های بدست آمده را بررسی کنیم. تعداد این پاد ها ۱؛ ۳؛ ۶ و ۱۰ در نظر گرفتیم که خودکار ساز ما بر اساس این تعداد پاد ها محدود خواهد شد. به این معنی که برای هر یک از آن چهار تعداد مشخص کاربر ما؛ خودکار ساز خود را به۱ پاد؛ ۳ پاد ؛ ۶ پاد و ۱۰ پاد محدود میکنیم و تا سقف همین تعداد امکان تولید پاد ها را دارد و نه بیشتر. در آخر؛ نتایج هر بخش را جداگانه دریافت میکنیم و بر اساس نتایجی که از تولید کننده بار برای این ۱۶ باری که این تولیدکننده بار را اجرا کردیم گرفتیم؛ میتوانیم رفتار سیستم خود را تحلیل کنیم که این تحلیل و بررسی نتایج در فصل بعدی آمده است.

بر اساس سیستم Linkerd میزان تاخیر پاسخ سرویس خود و تعداد درخواست در ثانیه را میتوانیم دریافت کنیم و بر اساس سیستم Grafana؛ گراف های این معیارها را میتوان برای زمان های مختلف مشاهده کرد. سپس ما برای هر اجرای تولید کننده بار بر اساس تعداد کاربر های متفاوت و محدودیت هایی که برای پاد ها گذاشته ایم؛ دو گراف از تاخیر پاسخ و تعداد درخواست بر ثانیه ذخیره میکنیم. به علاوه تولید کننده بار هم نتایج مختلفی را به ما میدهد که در بخش Load generator این نتایج را معرفی کردیم. بنابراین بر اساس نتایج تولید کننده بار و این دو گراف بدست آمده از Grafana؛ تحلیل های خود را انجام میدهیم.

4 جمع بندی:

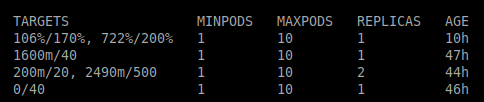
در این فصل ما سه روش اصلی برای خودکارسازی در کوبرنتیز را توضیح دادیم و پیاده سازی روش سوم را که بخش اصلی این پروژه هست را به طور دقیق و مفصل شرح دادیم که چگونه با استفاده از معیار های مختلف سرویس های خود را در پلتفرم کوبرنتیز مقایس پذیر کنیم.

فصل ششم: نتایج خودکار سازی سرویس ها

در این فصل ما نتایج بدست آمده از مقیاس پذیری بر اساس معیارهای مختلف را به نمایش خواهیم گذاشت. که در ابتدا نتایج برای معیار های محاسباتی و حافظ ای را نشان می دهیم. سپس نتایج مقیاس پذیری سرویس ها بر اساس معیارهای زمان پاسخ و تعداد درخواست در ثانیه را به صورت گراف ها و جداول مختلف نشان خواهیم داد و تحلیل و بررسی خواهیم کرد.

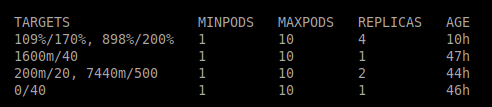
1 نتایج خودکار ساز برای معیار های محاسباتی و حافظه ای

حال نتایج عملکرد این خودکار ساز برای معیار های محاسباتی و حافظه ای را در عکس های زیر مشاهده میکنیم. برای اینکه میزان مصرفی محاسباتی سرویس ما زیاد شود از ابراز siege که قبلا توضیح داده بودیم استفاده میکنیم.



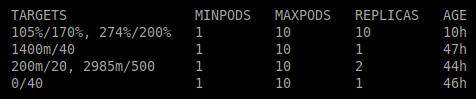
شکل۲۲ - خروجی خودکار ساز

در اینجا مشاهده میکنیم که بار بر روی پاد ما زیاد شده است و تقریبا 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.



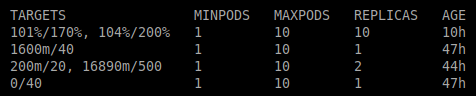
شکل۲۳ - مقیاس پذیری سرویس مد نظر

در مرحله بعدی می بینیم 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.



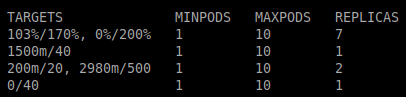
شکل ۲۴ - مقیاس پذیری سرویس و کم شدن بار بر روی هر سرویس

وقتی ماکزیمم 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.



شکل۲۵ - تقسیم بار بر روی سرویس ها و کم شدن مقدار مصرفی هر پاد

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 حدودا ۵ دقیقه طول میکشد تا تعداد پاد ها را کم کند.



و همین طور که بار کمتر میشود؛ این خودکار ساز تعداد پاد ها را هم کمتر میکند تا منابع سخت افزاری ما بهینه مصرف شوند و اتلاف منابع نداشته باشیم.



شکل ۲۶ - کم کردن سرویس ها توسط خودکار ساز بعد از کم شدن بار

برای معیار حافظه ای هم به همین صورت عمل میکند و وقتی پاد ما از حدی که مشخص کردیم بیشتر حافظه مصرف کرد؛ تعداد پاد ها را افزایش میدهد تا این میزان مصرفی برای هر پاد کمتر شود. تفاوتش با معیار محاسباتی در این است که برای افزایش میزان مصرفی یک سرویس از تولید کننده بار siege استفاده نمی کنیم و در عوض از ابزار stress استفاده میکنیم که با استفاده از این ابزار می توانیم میزان مصرفی حافظه برای یک سرویس را افزایش دهیم. برای اینکار ابتدا باید وارد کانتینر شویم که با استفاده از دستور زیر این کار را انجام میدهیم.

Kubectl exec -it [pouya-nginx-ingress-controller-59f4fdb9dd-sffsd](http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/#/pod/default/pouya-nginx-ingress-controller-59f4fdb9dd-sffsd?namespace=default) -- sh

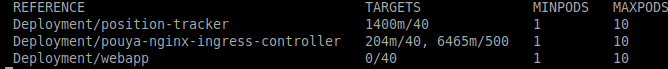
سپس بعد از نصب ابزار stress در کانتینر خود؛ از طریق دستور زیر میزان مصرفی حافظه را افزایش میدهیم.

stress --vm 2 --vm-bytes 200M

در این دستور ما مشخص میکنیم که دو thread هر کدام به میزان ۲۰۰ مگابایت شروع به مصرف حافظه بکنند. هنگامی که میزان مصرفی از حد تعیین شده رد شد؛ خودکار ساز ما پاد ها مثل معیار محاسباتی افزایش میدهد و بعد از تمام شدن بار؛ پاد ها را کاهش میدهد.

2 نتایج خودکار ساز برای custom metrics

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.

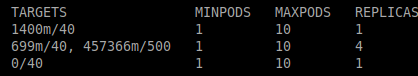
شکل ۲۷ - شمای کلی خودکار ساز

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 ۲۰۴mili 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 ۶ میلی ثانیه است.

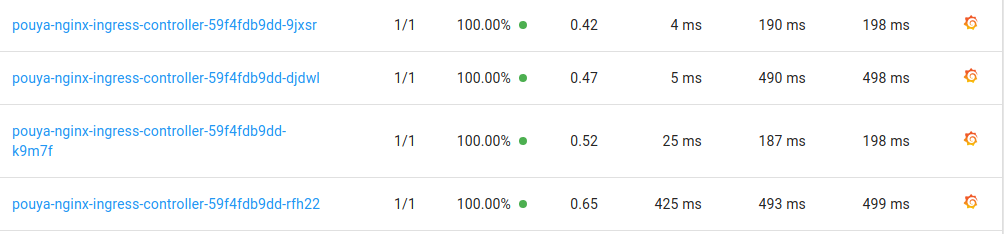
شکل ۲۸ - نتیجه خودکار ساز بعد از اعمال بار

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.



شکل ۲۹ - نتیجه خودکار ساز بعد از مقیاس پذیری سرویس

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.



شکل۳۰ - مربوط به نتایج Linkerd

میزان تاخیرهای P95P ,99 و P50 را برای این پاد ها در شکل بالا امده 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.

۱.۲. انجام تست ها بر روی سرویس مد نظر:

حال بعد از آنکه نحوه عملکرد این خودکار ساز را مشاهده کردیم؛ سراغ انجام سناریو های مربوط به اعمال بار بر روی سرویس خود میرویم که به ترتیب این بار ها را با تعداد کاربر ۳۰۰؛ ۴۰۰؛ ۵۰۰ و ۷۰۰ تست میکنیم و ماکزیمم پاد ها را هم به ۱؛ ۳؛ ۶ و ۱۰ محدود میکنیم. زمان انجام تست ها برای همه ی اجرا ها ۴ دقیقه است.

الف) بار با ۳۰۰ کاربر:

بعد از اعمال بار با ۳۰۰ کاربر با استفاده از تولید کننده بار برای ماکزیمم پاد های مختلف؛ نتایج گوناگونی را استخراج کردیم که به صورت جدول در شکل زیر آمده است.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Max pods =10 | Max pods = 6 | Max pods = 3 | Max pods = 1 |  |
| 51139 | 50642 | 41892 hits | 27954 hits | Transactions |
| 99.76% | 99.78% | 99.72% | 99.53% | Availability |
| 1.37 secs | 1.39 secs | 1.68 secs | 2.52 secs | Response Time |
| 213 trans/sec | 211 trans/sec | 174 trans/sec | 116.74 trans/sec | Transaction Rate |
| 293.13 | 293.42 | 294.71 | 294.18 | Concurrency |
| 8.10 | 7.94 | 28.46 | 32.07 | Longest Transaction |

جدول۱ - نتایج تولید کننده بار برای ۳۰۰ کاربر

ب) بار با ۴۰۰ کاربر:

حال نتایج را برای این تعداد کاربر در جدول زیر مشاهده میکنیم.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Max pods =10 | Max pods = 6 | Max pods = 3 | Max pods = 1 |  |
| 56838 | 55394 | 42354 hits | 29059 hits | Transactions |
| 99.51% | 99.53% | 99.42% | 99.30% | Availability |
| 1.65 secs | 1.69 secs | 2.22 secs | 3.23 secs | Response Time |
| 236.87 trans/sec | 231.48 trans/sec | 177.07 trans/sec | 121.35 trans/sec | Transaction Rate |
| 389.93 | 390.98 | 392.51 | 391.52 | Concurrency |
| 16.11 | 22.01 | 43.65 | 52.59 | Longest Transaction |

جدول۲ - نتایج تولید کننده بار برای ۴۰۰ کاربر

ج) بار با ۵۰۰ کاربر:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Max pods =10 | Max pods = 6 | Max pods = 3 | Max pods = 1 |  |
| 58167 hits | 56160 hits | 42422 hits | 28115 hits | Transactions |
| 99.40% | 99.35% | 99.17% | 98.66% | Availability |
| 2.01 secs | 2.08 secs | 2.77 secs | 4.16 secs | Response Time |
| 243.25 trans/sec | 234.66 trans/sec | 177.04 trans/sec | 117.50 trans/sec | Transaction Rate |
| 488.10 | 488.48 | 489.52 | 488.90 | Concurrency |
| 42.30 | 46.28 | 55.58 | 75.58 | Longest Transaction |

جدول۳ - نتایج تولید کننده بار برا ۵۰۰ کاربر

د) بار با ۷۰۰ کاربر:

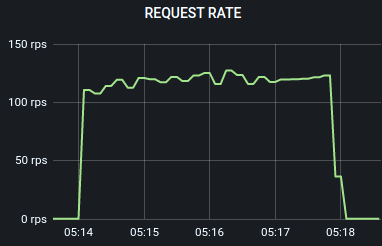
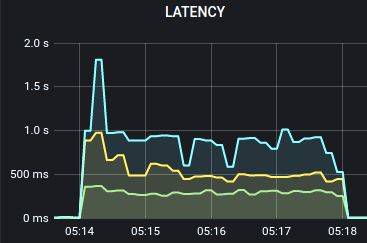
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Max pods =10 | Max pods = 6 | Max pods = 3 | Max pods = 1 |  |
| 57457 hits | 57885 hits | 42071 hits | 26705 hits | Transactions |
| 98.95% | 99.01% | 98.03% | 96.74% | Availability |
| 2.84 secs | 2.80 secs | 3.87 secs | 5.97 secs | Response Time |
| 239.59 trans/sec | 241.54 trans/sec | 175.32 trans/sec | 111.45 trans/sec | Transaction Rate |
| 679.88 | 675.66 | 678.45 | 665.38 | Concurrency |
| 91.88 | 119.60 | 110.88 | 135.24 | Longest Transaction |

جدول۴ - نتایج تولید کننده بار برای ۷۰۰ کاربر

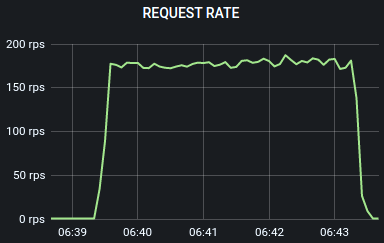
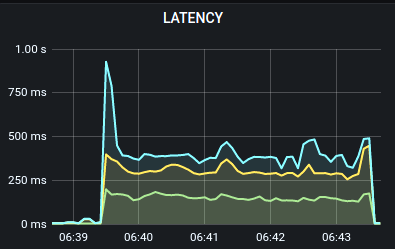
حال بعد از آنکه نتایج تولید کننده بار را مشاهده کردیم سراغ نتایجی که Linkerd به نشان داده است میرویم. سیستم Linkerd نتایج زمان پاسخ و تعداد درخواست در ثانیه را برای سرویس ما را با استفاده از Grafana به صورت شکل های زیر به نمایش میگذارد که در شکل های پایین این نتایج برای تعداد کاربر و ماکزیمم پادها نشان داده میشود. دلیلی که میزان زمان پاسخ برای تولید کننده بار و Linkerd متفاوت است؛ این است که زمان پاسخ در نتیجه تولید کننده بار؛مساوی جمع زمان پاسخ وب سرور ما و پاسخ سرویسی که وب سرور ما آن درخواست را به آن سرویس داده است. در صورتی که Linkerd تنها زمان پاسخ خود وب سرور ما را میدهد که همان سرویس nginx ما است.

الف) تعداد کاربر ۳۰۰:

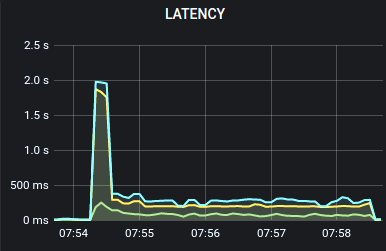
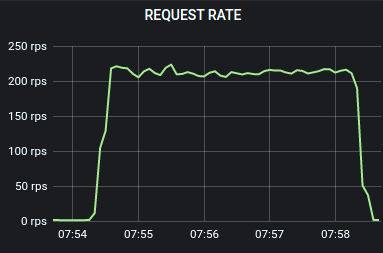
Max pods =1



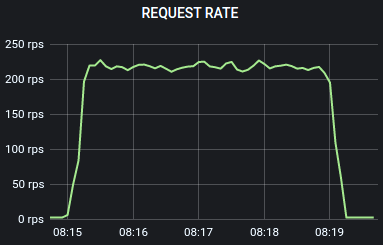
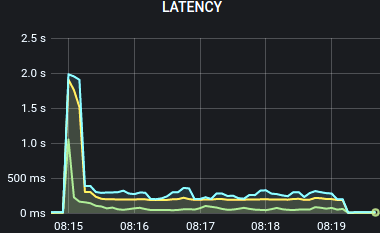
Max pods =3



Max pods = 6

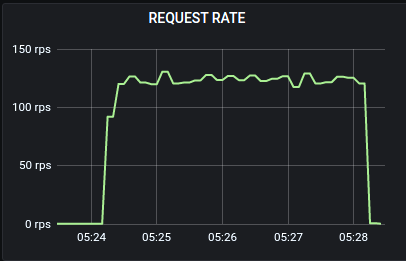
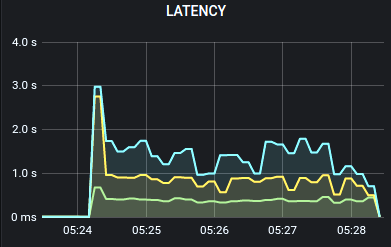


Max pods = 10

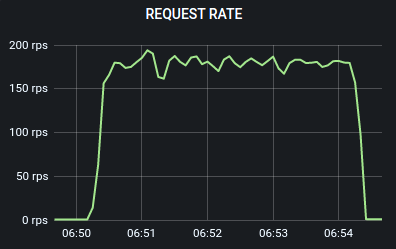
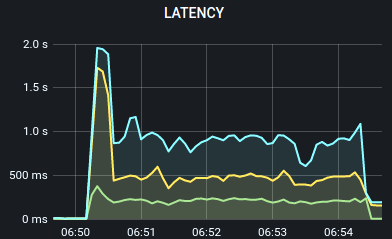


ب) تعداد کاربر ۴۰۰:

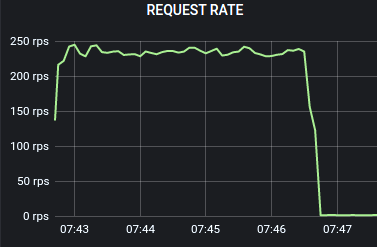
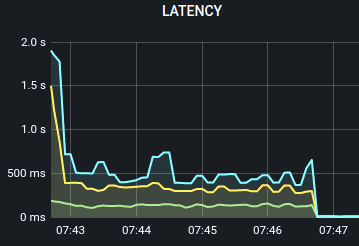
Max pods = 1



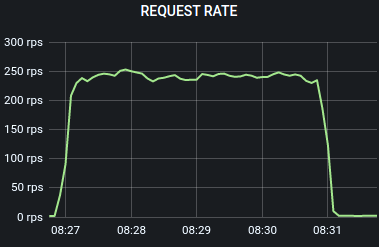
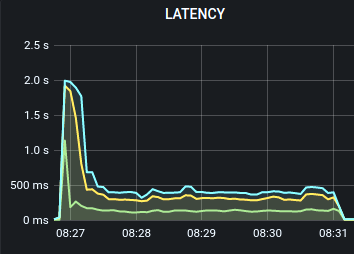
Max pods = 3



Max pods = 6

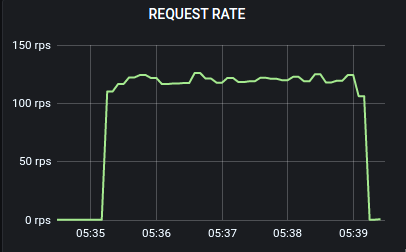
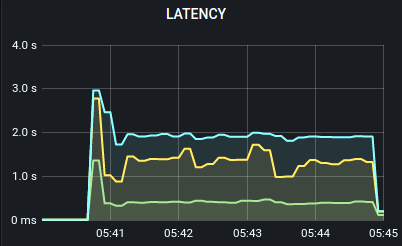


Max pods = 10

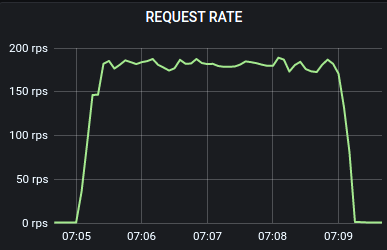
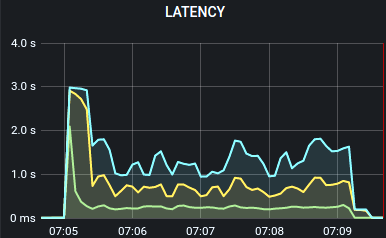


ج) تعداد کاربر ۵۰۰:

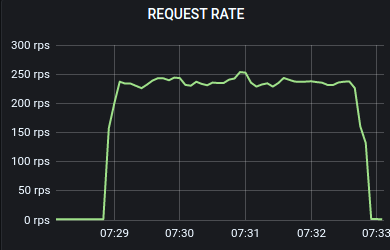
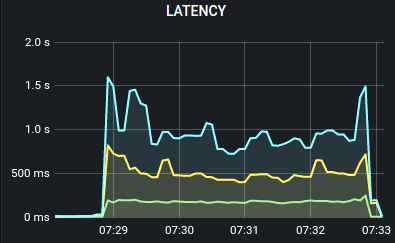
Max pods = 1



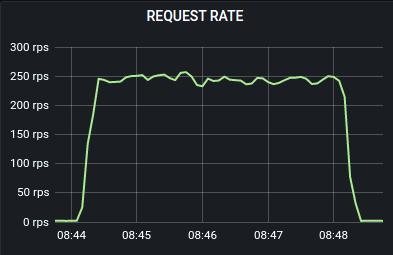
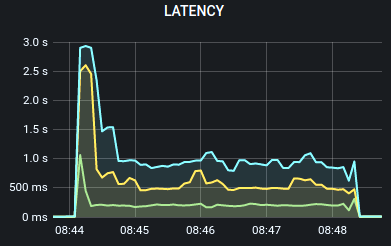
Max pods = 3



Max pods = 6

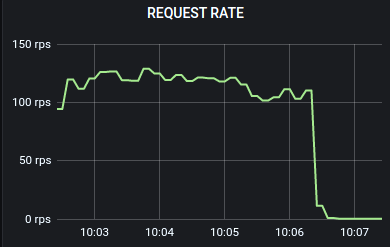
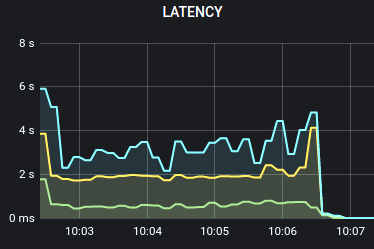


Max pods = 10

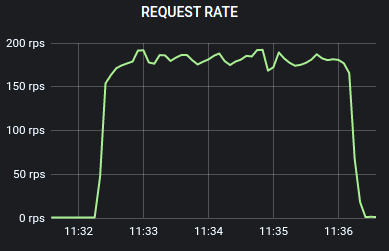
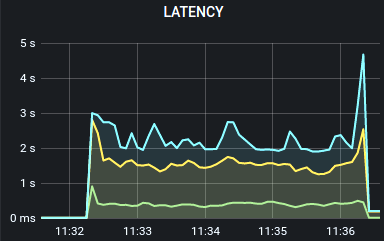


د) تعداد کاربر ۷۰۰:

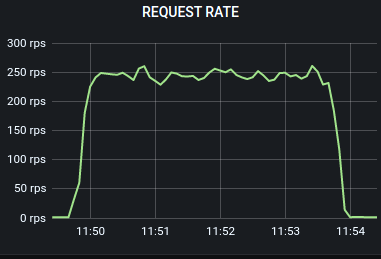
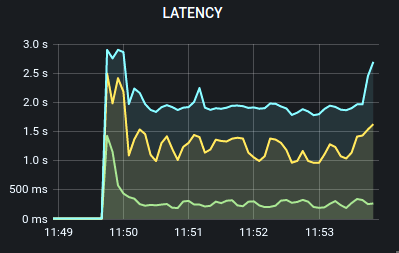
Max pods = 1



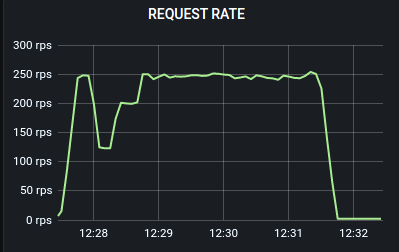
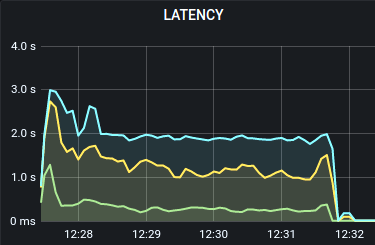
Max pods = 3



Max pods = 6



Max pods = 10



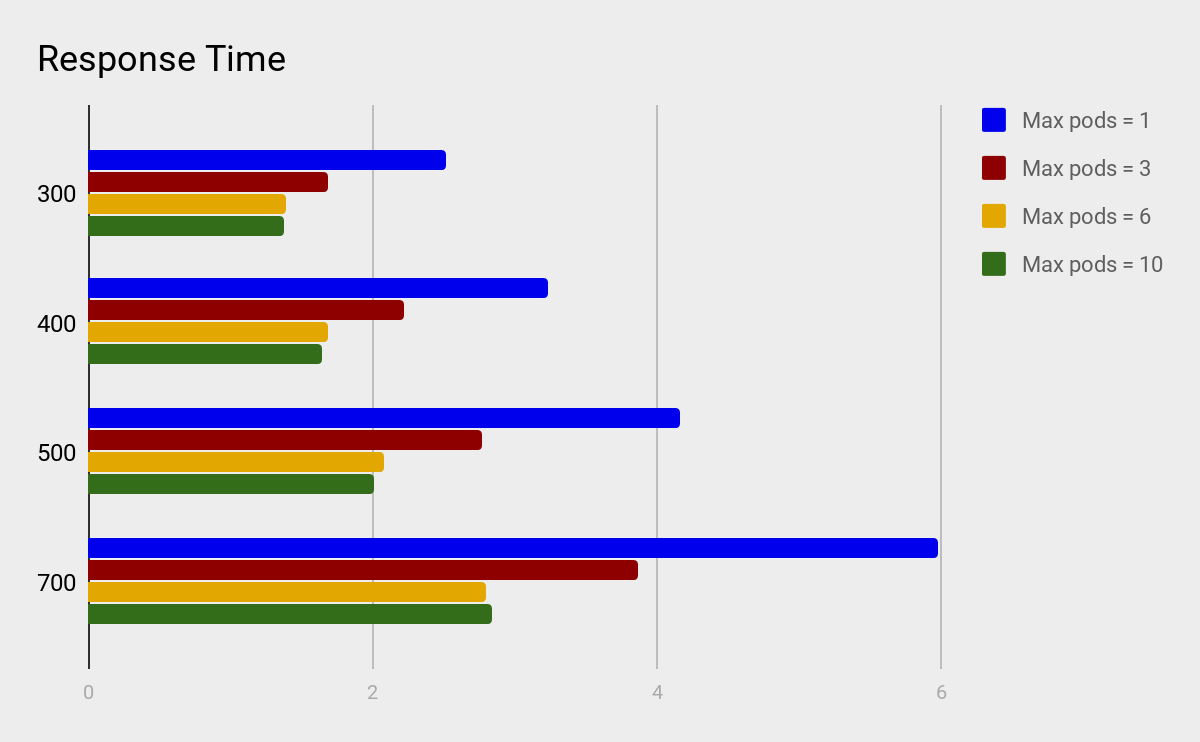
2-2 تحلیل نتایج:

بعد از آنکه نتایج را بدست آوردیم و گراف های مربوط به نتایج را هم مشاهده کردیم؛ سراغ تحلیل این نتایج میرویم. همان طور که در نتایج آمده است برای ۳۰۰ ؛ ۴۰۰؛ ۵۰۰ و ۷۰۰ کاربر تعداد ۳ پاد و ۶ پاد میزان زمان پاسخ را نسبت به یک پاد کمتر کرده است و این نشان میدهد که با تقسیم بار بر روی پاد ها؛ سرویس ما عملکرد بهتری دارد و زمان پاسخ کمتری نسبت به داشتن یک پاد نشان میدهد. ولی با توجه به اینکه منابع سخت افزاری ما محدود است این عملکرد تا یک جایی میتواند بهبود یابد و از یک حدی بیشتر دیگر افزایش پاد ها کمکی به بهبود زمان پاسخ ما نخواهد کرد و این به این منظور است که هر پاد از لحاظ نرم افزاری تعداد محدودی درخواست در ثانیه را می تواند جواب بدهد و هنگامی که بیش از حد توان خود بار دریافت می کند؛ خطا در جواب هم بالا میرود پس با تقسیم بار بر روی پاد ها می توانیم این مشکل را حل کنیم و باعث شود که هر پاد تعداد کمتری درخواست را پاسخ دهد. در آن طرف قضیه منابع سخت افزاری ما محدود است و هر پادی که تولید میشود یک بخشی از حافظه و CPU ما را اشغال میکند و اگر این منابع کم بیاید؛ سیستم ما کند میشود و دیگر قادر نخواهد بود به سرعت به درخواست ها جواب بدهد. پس اگر پاد ها را از یک حدی بیشتر کنیم؛ درست است که بار خیلی کمتر بر روی هر پاد می افتد و لی به دلیل آنکه منابع سخت افزاری هم محدود تر میشود؛ پاد های ما هم سرعتشان کمتر میشود. پس باید ما یک نقطه بهینه در سیستم خود پیدا کنیم که چه میزان پاد برای درخواست های ما مناسب است و برای تعداد درخواست هایی که ما در نظر گرفتیم ۶ پاد عملکرد سیستم ما رو بیشتر بهبود بخشیده و بیشتر از آن خیلی در نتایج فرقی ایجاد نمیکند.

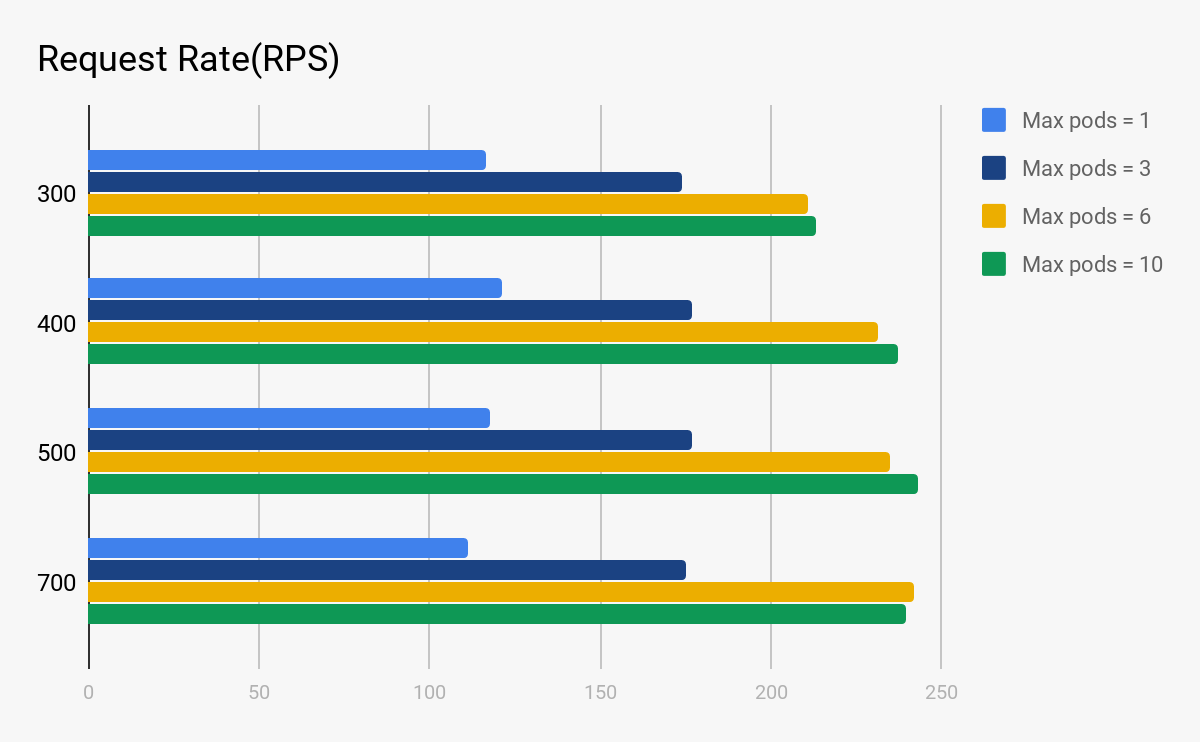
همچنین وقتی تعداد پاد ها را افزایش میدهیم و زمان پاسخ سرویس ما کمتر میشود به تبع آن میزان درخواست بیشتری در ثانیه را می تواند جواب دهد که این نتایج در گراف ها و جداول قسمت نتایج آمده است و همچنین جدولی برای مقایسه تعداد درخواست های مختلف و تاثیر افزایش پاد ها بر روی میزان تعداد درخواست در ثانیه را نشان میدهد. نکته ای که وجود دارد با توجه به مساله ای که در مورد مساله عملکرد نرم افزاری و محدودیت منابع سخت افزاری مطرح کردیم. با افزایش تعداد پاد ها تا یک حدی؛ عملکرد نرم افزاری ما بهتر میشود و پاد های ما بیشتر میتوانند پاسخگو باشند و تعداد بیشتری درخواست در ثانیه را جواب بدهند ولی بعد از یک حدی از تعداد پاد ها به دلیل محدودیت های سخت افزاری؛ سرویس ما نمی تواند تعداد بیشتری درخواست در ثانیه را جواب بدهد به دلیل آنکه دچار کمبود در منابع سخت افزاری می شویم و همین باعث میشود که تعداد بیشتر پاد ها به افزایش تعداد درخواست در ثانیه کمکی نکند.

همان طور که از نتایج ما مشخص است؛ با افزایش پادها میزان دسترس پذیری سیستم ما هم افزایش پیدا کرده است. درست است که این افزایش برای بار های پایین تر قابل توجه نیست ولی برای بار ۵۰۰ و ۷۰۰ این افزایش تا حدودی ملموس است و به دلیل آنکه بیشتر از ۷۰۰ سیستم دچار مشکل میشد تولید کننده بار ما قادر به ادامه نبود؛ امکان افزایش بار بیش از این وجود نداشت. در حالت کلی وقتی بار بر روی پاد ها تقسیم میشود و سرویس ما سریع تر بتواند درخواست ها را پاسخ بدهد و تعداد بیشتری درخواست پاسخ داده شود؛ به تبع آن دسترس پذیری هم افزایش میابد.

قسمت آخر از نتایج ما که میتواند قابل توجه باشد این است که با افزایش پاد ها حداکثر زمان پاسخ هم کم شده است. زمان پاسخی که در نتیجه تولید کننده بار آمده است متوسط همه زمان پاسخ ها است و بعضی از درخواست ها به دلیل آنکه بر روی سرویس ما بار زیادی افتاده است؛ طول میکشد پاسخ داده شود. ولی هنگامی که تعداد پاد ها را افزایش میدهیم و بار بین پاد ها تقسیم میشود؛ مشاهده میکنیم که حداکثر زمان پاسخ هم کاهش میابد و باعث بهبود عملکرد سرویس ما میشود.

****

شکل ۳۱ - مقایسه بارهای مختلف و ماکزیمم پاد ها برای زمان پاسخ



شکل ۳۲ - مقایسه بارهای مختلف و ماکزیمم پاد ها برای تعداد درخواست در ثانیه

3 جمع بندی:

در این فصل ما نتایج قیاس پذیری برای معیار های محاسباتی و حافظه ای را بدست آوردیم. همچنین نحوه عملکرد خودکار ساز برای این معیار ها هم نشان داده شد. در قسمت دوم مقیاس پذیری بر اساس معیار های زمان پاسخ و تعداد درخواست در ثانیه انجام شد. به دلیل آنکه این دو معیار در قسمت دوم اهمیت بیشتری دارند و عملکرد سرویس ها را بهتر و کاربردی تر نشان میدهند؛ ما سناریو های مختلفی برای مقیاس پذیری سرویس مد نظر خود در نظر گرفتیم و نتایج آن را به صورت گراف و جدول به نمایش گذاشتیم.