

دانشگاه صنعتی امیر کبیر
(پلی تکنیک تهران)
دانشکده مهندسی کامپیوتر

پروژه کارشناسی
گرایش فناوری اطلاعات

ارکستریشن خودکار برای دسترس پذیری بالا در محیط محاسبات ابری با
استفاده از ابزارهای متن باز

نگارش
محمد پویا خرسندی

استاد راهنما
دکتر بهادر بخشی سراسکانرود

تیر ۱۳۹۹

چکیده

در سال‌های اخیر، با توجه به افزایش تقاضا برای سرویس‌های ابری، نیازمند پلتفرم‌هایی برای مدیریت مطلوب و بهینه این سرویس‌ها هستیم. همچنین به دلیل آنکه کاربران زیادی به این سرویس‌ها متصل می‌شوند تا خدمات مدنظر را دریافت کنند، باید بتوان حجم زیادی از درخواست‌ها را مدیریت کرده و پاسخ داد. در این پروژه با استفاده از پلتفرم کورنتیز که یک سیستم قدرتمند در حوزه مدیریت سرویس‌های ابری است، قصد داریم، محیطی برای مدیریت خودکار مایکروسرویس‌ها فراهم کنیم به نحوی که مایکروسرویس‌ها در زمان‌های اوج کاری به صورت خودکار مقیاس‌پذیر گردند. مقیاس‌پذیر کردن مایکروسرویس‌ها با راه‌اندازی ماشین‌های کمکی جدید در زمان افزایش تقاضا انجام می‌گیرد تا بتوانیم بار کاری بر روی یک ماشین را کاهش و بین ماشین‌های دیگر تقسیم کنیم. این کار باعث می‌شود که دسترس‌پذیری مایکروسرویس‌ها بیشتر شده و همچنین زمان پاسخ‌گویی مایکروسرویس‌ها کمتر شود که در نتیجه می‌توان تعداد درخواست‌های بیشتری را پاسخ داد. برای دستیابی به این هدف و بخصوص خودکارسازی فرایند مقیاس‌پذیری، اجزا مانیتورینگ بار شامل metric-server و HPA به همراه api-server کورنتیز و محیط عملیاتی کورنتیز شامل مدیریت پیکربندی آبجکت‌های کورنتیز و بارگذاری آنها راه‌اندازی می‌شوند. در اینجا از فناوری کانتینر برای میزبانی مایکروسرویس استفاده شده است و پیاده‌سازی محیط پایلوت سرویس بر پایه وب سرویس انجام شده است. برای تست عملکرد سیستم از مولد بار siege برای تولید بار استفاده شده است و برای اندازه‌گیری وضعیت سیستم و معیارهای کارایی، سیستم‌های مختلفی مانند Prometheus و linkerd در کنار پلتفرم کورنتیز راه‌اندازی شده‌اند.

واژه‌های کلیدی: سرویس‌های ابری، پلتفرم کورنتیز، مقیاس‌پذیری، دسترس‌پذیری، زمان پاسخ‌گویی، میزان درخواست در ثانیه

فهرست مطالب

صفحه	عنوان
۱.....	چکیده.....
ب.....	فهرست مطالب.....
۱.....	فصل اول: مقدمه.....
۲.....	۱-۱ هدف و اهمیت کار.....
۳.....	۲-۱ ساختار پایان نامه.....
۴.....	فصل دوم: مفاهیم ماشین مجازی، کانتینر، داکر و کوبرنتیز.....
۴.....	۱-۲ ماشین‌های مجازی و کانتینرها.....
۴.....	۱-۱-۲ ماشین‌های مجازی.....
۴.....	۱-۱-۱-۲ مشکلات ماشین‌های مجازی.....
۵.....	۲-۱-۲ کانتینرها.....
۶.....	۱-۲-۱-۲ فواید استفاده از کانتینرها.....
۷.....	۲-۲ داکر (Docker).....
۷.....	۱-۲-۲ اجزای اصلی داکر.....
۸.....	۲-۲-۲ کارکرد کلی پلتفرم داکر.....
۹.....	۳-۲-۲ نحوه اجرا شدن کانتینر در داکر.....
۱۲.....	۴-۲-۲ میکروسرویس‌ها.....
۱۳.....	۳-۲ پلتفرم کوبرنتیز (Kubernetes).....

۱۳.....	۲-۳-۱ توضیح پلتفرم کورننتیز و مزیت‌های استفاده از آن
۱۴.....	۲-۳-۲ اجزا و ساختار پلتفرم و گره رهبر در کورننتیز
۱۷.....	۲-۳-۳ معماری و اجزا تشکیل دهنده گره کارگر
۲۰.....	۲-۴ جمع‌بندی
۲۱.....	فصل سوم : توضیح مایکروسرویس‌ها و ایمج کردن آنها در داکر
۲۱.....	۳-۱ توضیح مایکروسرویس‌ها و ساختارشان
۲۴.....	۳-۲ ایمج کردن مایکروسرویس‌ها
۲۴.....	۳-۲-۱ نوشتن Dockerfile و توضیح مراحل ایمج کردن
۳۰.....	۳-۳ جمع‌بندی
۳۱.....	فصل چهارم : راه‌اندازی پلتفرم کورننتیز
۳۱.....	۴-۱ راه‌اندازی کلاستر کورننتیز
۳۵.....	۴-۲ آبجکت‌های کورننتیز
۳۵.....	۴-۲-۱ RBAC
۳۸.....	۴-۲-۲ Pod ها، Replica set ها و Deployment ها
۴۱.....	۴-۲-۳ Service ها
۴۳.....	۴-۲-۳-۱ Nodeport و LoadBalancer
۴۵.....	۴-۲-۳-۲ Ingress
۴۷.....	۴-۲-۴ Persistent Volumes
۴۸.....	۴-۲-۴-۱ Dynamic NFS Provisioning
۵۰.....	۴-۳ جمع‌بندی

فصل پنجم: روش‌های مقیاس‌پذیری خودکار در پلتفرم کوبرنتیز.....	۵۱
۱-۵ مقیاس‌پذیری خودکار تعداد گره‌ها (cluster auto-scaler).....	۵۱
۲-۵ مقیاس‌پذیری خودکار تخصیص منابع (VPA).....	۵۲
۳-۵ مقیاس‌پذیری خودکار مایکروسرویس‌ها (HPA).....	۵۵
۱-۳-۵ معیارهای محاسباتی و حافظه‌ای.....	۵۵
۱-۳-۵-۱ نصب metrics server.....	۵۶
۲-۳-۵-۱ ساختن مقیاس‌پذیر کننده خودکار برای معیارهای محاسباتی و حافظه‌ای.....	۵۹
۲-۳-۵-۲ تولید بار (Load Generator).....	۶۰
۳-۳-۵ custom metrics.....	۶۲
۱-۳-۳-۵ نصب و ساختار linkerd.....	۶۳
۲-۳-۳-۵ معیارهای تاخیر در پاسخ (Response latency) و تعداد درخواست در یک ثانیه (RPS).....	۶۷
۳-۳-۳-۵ نحوه انجام تست‌ها برای custom metrics.....	۷۱
۴-۵ جمع‌بندی.....	۷۱
فصل ششم: نتایج مقیاس‌پذیری خودکار مایکروسرویس‌ها.....	۷۲
۱-۶ نتایج مقیاس‌پذیر کننده خودکار برای معیارهای محاسباتی و حافظه‌ای.....	۷۲
۲-۶ نتایج مقیاس‌پذیر کننده خودکار برای custom metrics.....	۷۴
۱-۲-۶ نتایج تست‌ها بر روی مایکروسرویس مد نظر.....	۷۵
۲-۲-۶ تحلیل نتایج.....	۸۳
۳-۶ جمع‌بندی.....	۸۶

۸۷	فصل هفتم: جمع‌بندی و پیشنهادات
۸۷	۱-۷ جمع‌بندی
۸۸	۲-۷ پیشنهادات
۸۹	منابع و مراجع

فصل اول: مقدمه

در محیط محاسبات ابری مقیاس بزرگ، مراکز داده ابر و کاربران نهایی از نظر جغرافیایی در سراسر جهان توزیع شده‌اند. بزرگترین چالش برای مراکز داده ابر این است که چگونه میلیون‌ها درخواست را که بطور مداوم از کاربران نهایی می‌رسند، بطور صحیح و موثر رسیدگی کنند و سرویس دهند. برای این منظور پلتفرم‌هایی در سطح^۱ *paas* که سرویس ابری به شکل پلتفرم است، توسعه پیدا کرده‌اند که می‌توانند این تعداد بار زیاد را مدیریت کنند. یکی از این پلتفرم‌ها، کوبرنتیز^۲ نام دارد که می‌تواند مجموعه وسیعی از امکانات سخت‌افزاری را مدیریت کند و با استفاده از فناوری‌های مجازی‌سازی مانند ماشین‌های مجازی و کانتینرها (کارگزارها) و توزیع آنها در بستر سخت‌افزار در سطح وسیع به میزبانی میکروسرویس‌ها اقدام نماید و بار زیادی را مدیریت کند. کوبرنتیز نسبت به راهکارهای قبلی و بطور خاص OpenStack مزایای قابل توجهی از جمله پیچیدگی کمتر و قابلیت انعطاف بیشتر برخوردار است. پیاده‌سازی اولیه کوبرنتیز بر اساس پیکربندی ثابت استوار بود و هر گونه تغییر در پیکربندی از جمله افزودن یا حذف یک کارگزار بطور دستی انجام می‌گیرد. اخیراً نیاز به پیکربندی پویا و وابسته به بار در دنیای محاسبات ابری مطرح شده است که به آن قابلیت مقیاس‌پذیری خودکار (Auto scaling) گفته می‌شود. وجود چنین قابلیت‌هایی باعث کاهش قابل توجه هزینه برای مشتریان و کاربران می‌شود چرا که نیازی به راه‌اندازی تعداد زیادی ماشین مجازی مطابق با شرایط پرباری سیستم نخواهد بود که در اکثر اوقات بلا استفاده می‌مانند بلکه تعداد گره‌های کارگزار با توجه به شرایط باری سیستم بطور خودکار افزایش و کاهش می‌یابد به نحوی که کارگزار بیکار که هزینه اضافی ایجاد می‌کند نداشته باشیم. برای راه‌اندازی چنین محیطی، اجزای مختلفی نیاز بوده توسعه یابد تا در کنار کوبرنتیز مانیتورینگ وضعیت بار و کیفیت سرویس را مانیتور کنند و معیارهای کیفیت سرویس را محاسبه کنند و گره‌های کارگزار را بدون دخالت مدیر سیستم اضافه یا کم کنند. سازگاری بین این اجزا که بطور مجزا از هسته اولیه پلتفرم کوبرنتیز توسعه یافته‌اند یک چالش جدی محسوب می‌شود. هدف این است که مجموع عوامل سیستم با همکاری هم با مدیریت میکروسرویس‌ها^۳ بتوانند بار زیادی را مدیریت کرده و میکروسرویس‌ها را به نحو مطلوبی مدیریت و مقیاس‌پذیر نمایند. این پلتفرم با ایجاد تعادل

^۱ Platform as a Service

^۲ Kubernetes

^۳ Microservices

بار، به رسیدن به بیشترین سطح رضایت کاربر و افزایش نرخ استفاده از منابع محاسباتی و حافظه‌ای به صورت بهینه کمک می‌کند.

۱-۱ هدف و اهمیت کار

هدف از انجام این پروژه مقیاس‌پذیر کردن میکروسرویس‌ها در زمان‌های پیک باری با استفاده از پلتفرم کوبرنتیز در مقیاس آزمایشگاهی و همچنین توضیح و راه‌اندازی فرآیند انجام مقیاس‌پذیری خودکار میکروسرویس‌ها با استفاده از پلتفرم کوبرنتیز و داکر است.

پلتفرم کوبرنتیز به ما کمک می‌کند که بتوانیم سرویس‌ها را به صورت ساختار میکروسرویس^۱ در بیاوریم و هر کدام را جداگانه در این پلتفرم بارگذاری کنیم. این روش به ما کمک می‌کند تا پیچیدگی‌ها در برنامه نویسی را کاهش دهیم و همچنین بتوانیم میکروسرویس‌ها را راحت‌تر و کاربردی‌تر مانیتور کنیم و در مواقعی که بار زیاد شد میکروسرویس مورد نظر را مقیاس‌پذیر کنیم و نه همه‌ی میکروسرویس‌های موجود را. این قابلیت همچنین باعث می‌شود در مواقعی که سیستم دچار اختلال و مشکل می‌شود راحت‌تر ایراد و عیب سیستم را پیدا کنیم، به دلیل آنکه عملکرد هر میکروسرویس مشخص است، در نتیجه خطاها هم مشخص است که مربوط به کدام میکروسرویس است.

امروزه شرکت‌های بزرگ برای مدیریت میکروسرویس‌ها از این پلتفرم استفاده می‌کنند تا خدمات خود را در زمان‌های پیک باری به نحوی که از دسترس خارج نشوند و زمان تاخیر پاسخشان زیاد نشود، ارائه کنند. همچنین این پلتفرم برای سازمان‌ها و شرکت‌های بزرگ به این دلیل مهم است که کار خودکارسازی^۲ را در ابعاد مختلف انجام می‌دهد و لازم نیست افراد سازمان‌ها یا شرکت‌ها کل روز میکروسرویس‌ها را مانیتور کنند. به طور مثال هنگامی که بار بر روی یک میکروسرویس زیاد شد، این پلتفرم میکروسرویس‌ها را به صورت خودکار مقیاس‌پذیر می‌کند و یا میکروسرویسی از دسترس خارج شد یکی دیگر از آن میکروسرویس اجرا می‌کند و یا سلامت میکروسرویس‌ها را مستمر چک می‌کند و در صورت بروز خطا به ما اطلاع داده می‌شود. پس با این پلتفرم هزینه مدیریت و نگهداری میکروسرویس‌ها کاهش پیدا می‌کند و همچنین دسترس‌پذیری^۳ میکروسرویس‌ها بیشتر می‌شود.

¹ Microservice Architecture

² Automation

³ Availablity

۱-۲ ساختار پایان نامه

در ابتدا در فصل دوم، توضیح مختصری از پلتفرم‌های داکر و کوبرنتیز برای آشنا شدن با ساختار و اجزا این پلتفرم‌ها، ارائه می‌کنیم. در فصل سوم، سرویس‌ها را به صورت ساختار میکروسرویس درمی‌آوریم و با استفاده سیستم داکر، میکروسرویس‌ها را ایمیج می‌کنیم که بتوانیم در پلتفرم کوبرنتیز بارگذاری، اجرا و مدیریت کنیم. در فصل ۴، ابتدا نحوه راه‌اندازی کلاستر کوبرنتیز را توضیح می‌دهیم و عملکرد و فایل‌های پیکربندی^۱ آبجکت‌های استفاده شده برای این پروژه را شرح می‌دهیم. در فصل ۵ و ۶، معیارهایی^۲ که برای مقیاس‌پذیری^۳ مناسب است را بررسی می‌کنیم و تعدادی از آنها را معرفی می‌کنیم. برای آنکه این معیارها را استخراج کنیم باید ابزارهای متفاوتی را در کنار پلتفرم کوبرنتیز خود نصب و راه‌اندازی کنیم. بعد از انجام نصب و راه‌اندازی، تست‌های مقیاس‌پذیری خودکار را بر اساس معیارهای مختلف بر روی میکروسرویس مدنظر انجام می‌دهیم. دو معیاری که برای در این پروژه اهمیت دارد معیارهای زمان پاسخ میکروسرویس‌ها و تعداد درخواستی که میکروسرویس در ثانیه می‌تواند جواب بدهد، هستند. با استفاده از این دو معیار کار مقیاس‌پذیری را انجام می‌دهیم و سپس عملکرد میکروسرویس‌ها را بعد از مقیاس‌پذیری مورد تحلیل و بررسی قرار می‌دهیم و نتایج این تست‌ها را به صورت جدول و گراف به نمایش می‌گذاریم.

¹ Configuration File

² Metrics

³ Scalability

فصل دوم: مفاهیم ماشین مجازی، کانتینر، داکر و کوبرنتیز

در این فصل به توضیح مفاهیم پایه سیستم داکر^۱ و پلتفرم کوبرنتیز می‌پردازیم. به دلیل آنکه میکروسرویس‌ها را در این دو محیط پیاده می‌کنیم، لازم است که با این مفاهیم آشنا باشیم.

۱-۲ ماشین‌های مجازی و کانتینرها

قبل از ورود به مباحث اصلی داکر، آشنایی با مشخصات ماشین‌های مجازی^۲ و مشکلات آنها می‌تواند دلایلی که باعث بوجود آمدن کانتینرها شدند را بیشتر مشخص کند.

۱-۱-۲ ماشین‌های مجازی

قبل از بوجود آمدن کانتینرها^۳، روش اصلی برای ایجاد محیط ایزوله برای مدیریت نرم‌افزارها استفاده از ماشین‌های مجازی بود. در این روش هر نرم‌افزار و سیستم‌هایی که برای اجرا به آنها نیاز داشت در یک ماشین مجازی مستقل با سیستم عامل مجزا نصب می‌شود. چند ماشین مجازی متفاوت می‌توانند بر روی یک سیستم سخت‌افزاری نصب شده و مشخصات این سیستم (مانند فضای هارد یا رم) را به صورت مجزا استفاده کنند. در این حالت در هنگام تعریف هر ماشین مجازی مقدار منابع سخت‌افزاری که توسط ماشین مجازی قابل استفاده است مشخص شده و در صورت نیاز در هر زمان امکان تغییر این موارد وجود دارد.

۱-۱-۱-۲ مشکلات ماشین‌های مجازی

الف) ماشین‌های مجازی با توجه به نیاز به نصب کامل سیستم عامل و امکانات مورد نظر هر نرم‌افزار به صورت مستقل، معمولاً حجم بالایی از فضای هارد سرور اصلی را اشغال می‌کنند.

¹ Docker

² Virtual Machines

³ Containers

ب) راه‌اندازی^۱ و اجرای چند ماشین مجازی همزمان نیاز به استفاده از امکانات سخت‌افزاری بالایی را بر روی سرور اصلی ایجاد می‌کند و در غیر این صورت باعث کندی یا ناپایداری سرور خواهد شد.

ج) راه‌اندازی هر ماشین مجازی به دلیل آنکه باید سیستم عامل مستقل آن ماشین به صورت کامل راه‌اندازی شود مدت زمان زیادی طول می‌کشد.

د) ماشین‌های مجازی به صورت کلی کمک چندانی به انتقال سیستم‌های نرم‌افزاری از یک سرور به سرور دیگر نمی‌کنند و همچنین در صورتی که سیستم عامل‌ها احتیاج به به روز رسانی داشته باشند این کار برای سیستم عامل هر ماشین مجازی باید به صورت مستقل انجام شود.

۲-۱-۲ کانتینرها

کانتینرها بر خلاف ماشین‌های مجازی که ایزوله‌سازی^۲ محیط اجرای نرم‌افزار را در سطح سخت‌افزار سرور انجام می‌دهند، ایجاد این محیط را به سطح سیستم عامل^۳ نصب شده بر روی سرور منتقل می‌کنند. به همین دلیل کانتینرها از نظر استفاده از منابع سرور بسیار کارآمدتر عمل می‌کنند چرا که در این حالت سیستم عامل مجزایی بر روی سرور اصلی نصب نمی‌شود و همین‌طور منابع نرم‌افزاری مورد نیاز در صورتی که به صورت مشترک در چند کانتینر استفاده می‌شوند می‌توانند بر روی سرور اصلی نصب شده باشند.

^۱ Boot

^۲ Isolation

^۳ Operating System

الف) کانتینرها از سیستم عامل مستقل برای اجرای نرم‌افزارها استفاده نمی‌کنند و می‌توانند منابع نرم‌افزاری مشترک را با هم به اشتراک بگذارند به همین دلیل حجم فضایی که توسط کانتینرها اشغال می‌شود بسیار کمتر از ماشین‌های مجازی است.

Containers vs. VMs

شکل ۲-۱: تفاوت کانتینر و ماشین مجازی [۱]

۲-۲ داکر (Docker)

داکر ابزاری است که برای توسعه، راه‌اندازی و اجرای راحت‌تر نرم‌افزارها بوسیله کانتینر طراحی شده است. کانتینرها به توسعه‌دهنده‌ها^۱ اجازه می‌دهند که نرم‌افزارهای خود را به همراه تمام مواردی که برای اجرای آنها احتیاج دارند (کتابخانه‌های نرم‌افزاری و غیره) به صورت یک پکیج آماده کرده و به سرور منتقل کنند. با این روش توسعه‌دهنده می‌تواند مطمئن باشد که نرم‌افزار آماده شده در هر سیستم عاملی که بر روی سرور نصب شده باشد و با هر تنظیماتی که در سیستم عامل ایجاد شده باشد، به درستی کار خواهد کرد و تغییر سیستم عامل یا تنظیمات آن اشکالی در اجرای نرم‌افزار ایجاد نخواهد کرد.

در نگاه اول داکر از نظر کارکرد مشابه ماشین مجازی به نظر می‌رسد. برخلاف ماشین مجازی، بجای راه‌اندازی یک سیستم عامل کاملاً مجزا بر روی سرور، داکر به نرم‌افزارها اجازه می‌دهد که از هسته سیستم عامل اصلی که بر روی سرور نصب شده است استفاده کنند و تنها مواردی که مستقل از سیستم عامل سرور عمل می‌کنند نیازهای اختصاصی نرم‌افزار می‌باشند که بر روی سرور نصب نشده و در کانتینر داکر نصب شده‌اند. این امر افزایش قابل توجهی در عملکرد سیستم ایجاد کرده و حجم کانتینرها را به نسبت ماشین مجازی به مقدار زیادی کاهش می‌دهد.

۲-۲-۱ اجزای اصلی داکر

سه مفهوم اصلی که در هنگام استفاده از داکر باید با آنها کاملاً آشنا باشید عبارتند از داکر فایل^۲، ایمج^۳ و کانتینر مجموعه این مفاهیم، فرایند اصلی داکر و روش استفاده از آن برای مدیریت نرم‌افزارها را مشخص می‌کند.

الف) داکر فایل

داکر فایل یک فایل متنی حاوی تمام دستوراتی است که با اجرای آنها تمام نیازمندی‌ها و تنظیمات مربوط به نرم‌افزاری که می‌خواهید توسط داکر اجرا شود در یک بسته داکر به نام داکر ایمج ایجاد می‌شود. این فایل دستورالعمل مربوط به ایجاد این بسته را در اختیار داکر قرار می‌دهد و داکر با استفاده از این دستورالعمل و با استفاده از دستور `docker build` این بسته را ایجاد می‌کند.

¹ Developers

² Docker File

³ Image

ب) داکر ایمیج

داکر ایمیج در تعریف ساده بسته‌ای است که با استفاده از آن می‌توان کانتینرهای داکر را ایجاد کرد. به عبارت دیگر داکر ایمیج بسته‌ای است که پس از ایجاد آن توسط دستورات داکر فایل امکان تغییر آن وجود ندارد و با استفاده از آن می‌توان هر تعداد کانتینر مورد نیاز برای اجرای نرم‌افزار مورد نظر را راه‌اندازی کرد.

داکر ایمیج‌ها در رجیستری داکر^۱ ذخیره می‌شوند. این رجیستری می‌تواند یک رجیستری خصوصی باشد که فقط شما به آنها دسترسی دارید یا یک رجیستری عمومی مانند داکرهاب (Docker Hub) که به سایر افراد اجازه می‌دهد از ایمیج ایجاد شده توسط شما برای اجرای نرم‌افزارهای موجود در آن استفاده کنند. استفاده از رجیستری‌های عمومی به صورت متداول در هنگام انتشار نرم‌افزارهای رایگان و متن‌باز^۲ کاربرد دارند و این امکان را در اختیار دیگران قرار می‌دهند که فقط با دانلود ایمیج و ایجاد یک کانتینر از روی آن به راحتی و بدون نگرانی از نیازمندی‌ها و تنظیمات نرم‌افزار، از آن استفاده کنند.

ج) داکر کانتینر

با استفاده از داکر ایمیج و دستور `docker run` می‌توان یک نمونه‌ی اجرایی از نرم‌افزار مورد نظر را به صورت داکر کانتینر ایجاد کرد. بنابراین داکر کانتینر به صورت کلی یک نسخه‌ی آماده‌ی اجرا از نرم‌افزار موجود در داکر ایمیج هستند که هدف نهایی استفاده از داکر و اجرای نرم‌افزار توسط آنها را محقق می‌کنند.



شکل ۲-۲: نحوه کارکرد سیستم داکر [۳]

۲-۲-۲ کارکرد کلی پلتفرم داکر

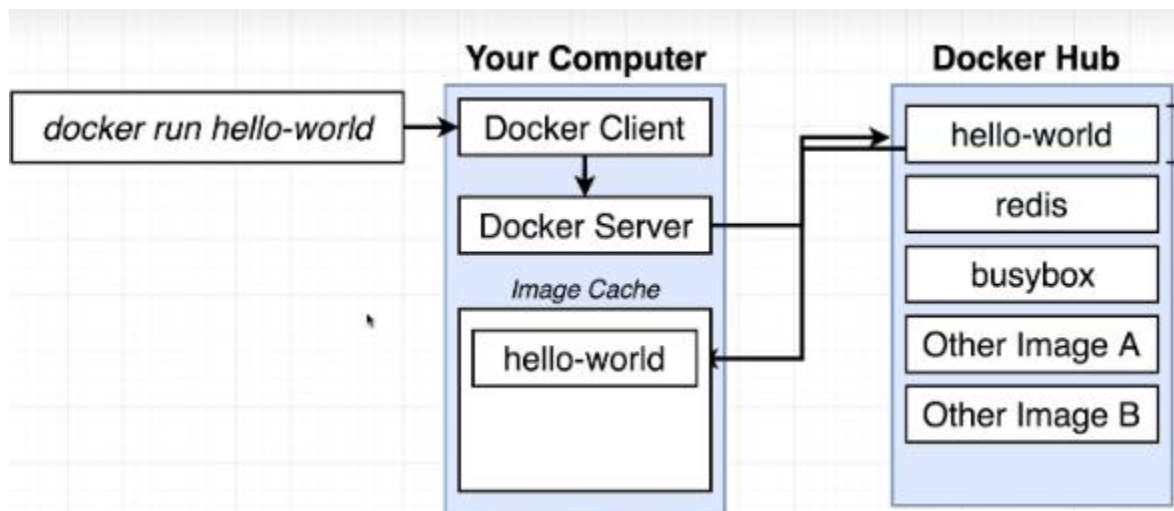
داکر از دو بخش داکر کلاینت و داکر سرور تشکیل شده است که بخش کلاینت دستورات را از کاربر می‌گیرد و به داکر سرور برای اجرای آن دستور می‌فرستد. داکر کلاینت به `CLI`^۳ سرور متصل شده است که از آن طریق دستورات را دریافت می‌کند. با استفاده از داکر سرور می‌توانیم ایمیج‌های خود را با استفاده از داکر فایل‌ها درست

^۱ Dokcer Hub

^۲ Open Source

^۳ Command Line

کنیم و به داکرهاب منتقل کنیم. این کار باعث می‌شود در هر سرور دیگری بتوانیم این ایمج‌ها را به راحتی اجرا کنیم. حال در شکل زیر این ارتباط را مشاهده می‌کنیم.

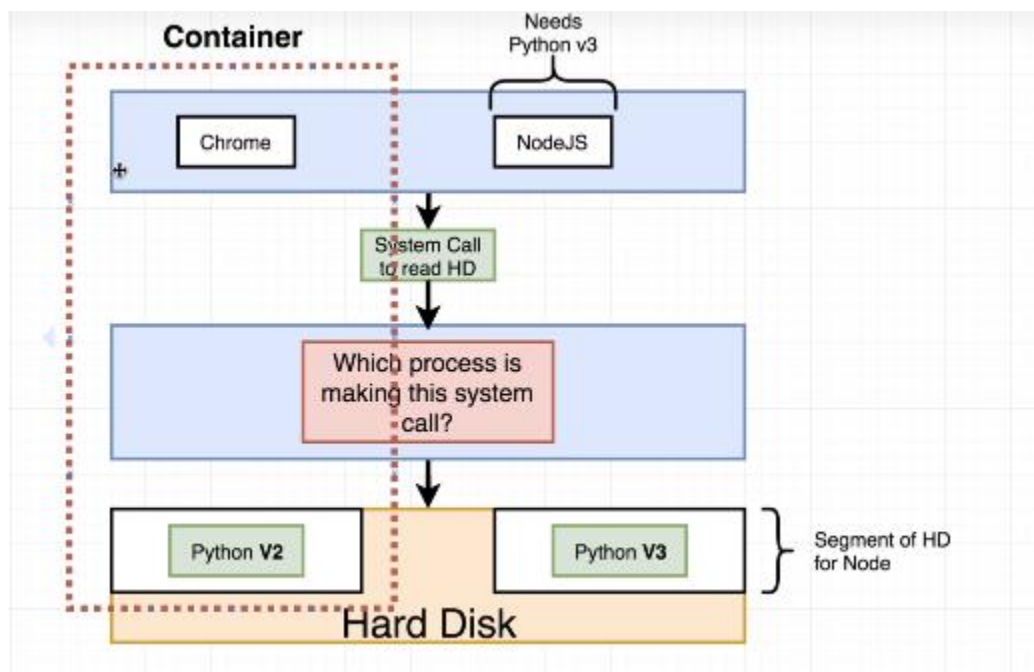


شکل ۲-۳: نحوه اجرای یک ایمج از داکرهاب [۲]

همان طور که در شکل بالا مشاهده می‌شود با اجرای دستور داکر به داکر کلاینت متصل می‌شویم و سپس اگر ایمج قبلاً از داکرهاب دانلود شده بود، دیگر نیازی به متصل شدن به داکرهاب نیست و داکر سرور این ایمج را اجرا خواهد کرد. در غیر این صورت باید متصل شود و ایمج را دانلود و سپس اجرا کند.

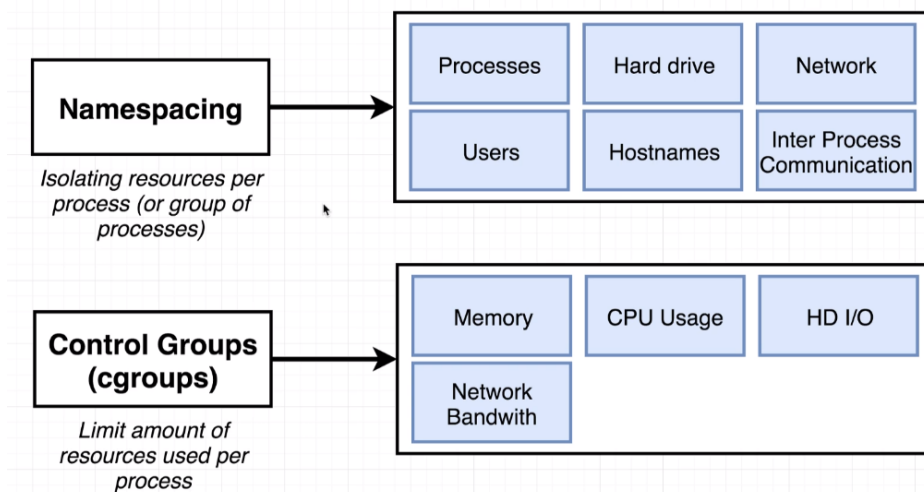
۲-۳-۳ نحوه اجرا شدن کانتینر در داکر

همان طور که گفته شد کانتینر یک نرم افزار است که با تمام مواردی لازم دارد اجرا شود ایزوله می‌شود و توسط container runtime اجرا می‌شود. در مرحله بعدی، می‌خواهیم ببینیم این عملیات چگونه صورت می‌پذیرد.



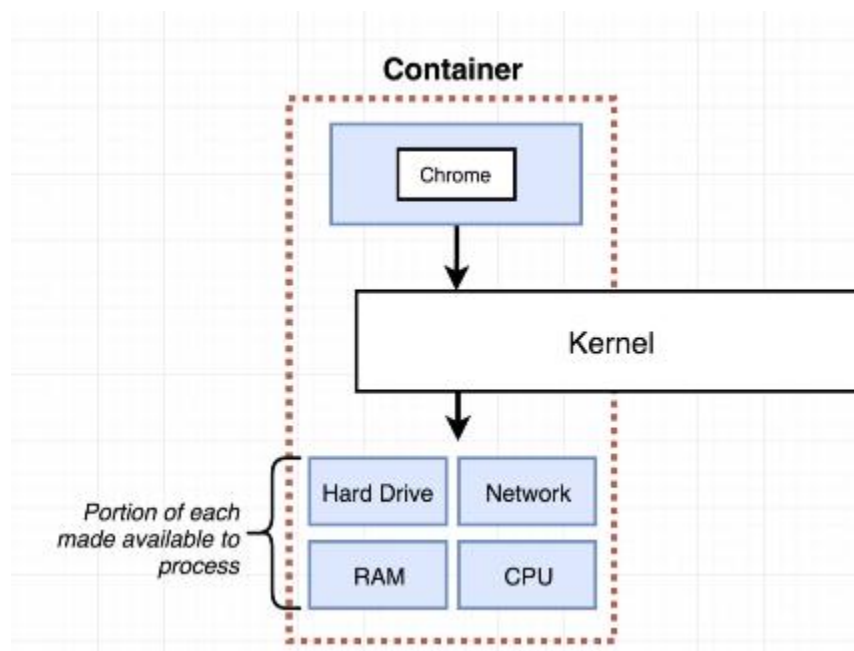
شکل ۲-۴: فضای یک کانتینر در سرور [۲]

در شکل زیر مشاهده می‌کنیم برای اینکه این برنامه اجرا شود به Python V2 نیاز دارد. پس قسمتی از دیسک به این منظور برای این برنامه جدا می‌شود و در آن این برنامه ذخیره می‌شود. پس کانتینر به این صورت عمل می‌کند که یک برنامه ای است که نیازمندی‌های نرم افزاری و سخت افزاری اش توسط سیستم عامل مدیریت می‌شود و مفهومی به اسم namespace وجود دارد که برای هر کانتینر یک فضای مخصوص از لحاظ حافظه، شبکه ای و موارد دیگر در نظر گرفته می‌شود که هر موقع این برنامه درخواستی از سیستم عامل داشت، سیستم عامل تشخیص بدهد کدام برنامه است و هنگامی که این برنامه نیاز به این برنامه‌ها برای اجرا شدن داشت باید به کدام قسمت حافظه باید درخواست بدهد.



شکل ۲-۵: Namespacing و cgroups در داکر [۲]

در شکل بالا مشاهده می‌شود که با استفاده از namespaces، داکر سرور محیط ایزوله برای منابع کانتینر فراهم می‌کند و دسترسی کانتینر به آن namespace محدود می‌شود. Cgroupها هم مشخص می‌کنند که هر برنامه که به صورت کانتینر در آمده است چه مقدار اجازه دارد که از منابع مختلفی که در سرور وجود دارد، در شکل بالا هم برخی از این منابع آمده است، استفاده کند. شکل زیر به صورت ملموس‌تر نشان می‌دهد که یک کانتینر در سیستم چگونه است. با استفاده از درخواست‌هایی که برنامه، که یک پروسس در سیستم عامل هست، به کرنل می‌دهد، سیستم عامل با استفاده از منابعی که برای این پروسس در نظر گرفته است جواب می‌دهد و منابع را در اختیار این پروسس قرار می‌دهد.



شکل ۲-۶: نحوه ایزوله شدن یک کانتینر در داکر [۲]

۲-۲-۴ مایکروسرویس‌ها

بعد از اینکه سیستم داکر شهرت گرفت، این پلتفرم برنامه نویسان را به این سمت سوق داد که برنامه‌های خود را به صورت مایکروسرویس بنویسند به این معنی که سرویس خود را به بخش‌های کوچک‌تر تقسیم می‌کنیم و هر کدام وظیفه انجام کاری را دارند. این نوع ساختار برنامه نویسی باعث شد تا برنامه‌هایی که پیچیدگی زیاد دارند را با تقسیم به برنامه‌های کوچک‌تر این پیچیدگی کمتر شود و راحت‌تر و سریع‌تر بتوانند کار عیب‌یابی و رفع اشکالات را انجام دهند. همچنین با استفاده از این ساختار مایکروسرویزی می‌توان از ارکستریتورها استفاده کرد و این مایکروسرویس‌ها را به صورت خودکار مدیریت کنیم به طور مثال هنگامی که بار بر روی یک مایکروسرویس زیاد شد تعداد بیشتری از این مایکروسرویس‌ها ساخته شود و سرویس دهی بهبود یابد. در این پروژه از این ساختار استفاده خواهیم کرد و سرویس‌ها را به مایکروسرویس‌ها درخواهیم آورد تا بتوان بر اساس پلتفرم کوبرنتیز که در بخش بعدی به آن خواهیم پرداخت، این مایکروسرویس‌ها را مدیریت کنیم.

۲-۳ پلتفرم کوبرنتیز (Kubernetes)

۲-۳-۱ توضیح پلتفرم کوبرنتیز و مزیت‌های استفاده از آن

کوبرنتیز (Kubernetes) (که به شکل k8s نیز ارجاع می‌شود) سامانه‌ای متن‌باز برای خودکارسازی دیپلوی^۱، مقیاس و مدیریت برنامه‌های کانتینرسازی شده در سراسر زیرساخت است که در ابتدا توسط گوگل توسعه داده شد و سپس در سال ۲۰۱۵ به بنیاد CNCF^۲ اهدا شد.

این پلتفرم وظیفه اجرا و مدیریت کانتینرها را بر روی گروهی از سرورهای موجود در یک یا چند مرکز داده‌ها^۳ به عهده دارد. کوبرنتیز در واقع نسل سوم از این فناوریست که در شرکت گوگل از ابتدا به زبان گو^۴ پیاده سازی شده است. دو نسل قبلی آن برگ^۵ نام داشته که پیاده سازی آن به زبان سی پلاس پلاس بوده است و گوگل همچنان از آن در محیط عملیاتی استفاده می‌کند.

مزیت کلیدی کوبرنتیز در این است که بدون نیاز به یک تیم بزرگ برای راه‌اندازی و نگهداری، می‌توان آن را در مقیاس وسیع برای اجرای تعداد زیادی برنامه کاربردی به کار گرفت. از مزایای دیگر آن قابلیت اجرا بر روی بسترهای متفاوت است، از سرورهای یک مرکز داده‌های خصوصی گرفته تا سرویسهای ابری عمومی، یا حتی ترکیبی^۶ از هر دو. به طور کلی هر شرکتی که یک یا چند سرویس نرم افزاری اجرا می‌کند به طور بالقوه در مرحله اول به کانتینرها و سپس به سیستمی مانند کوبرنتیز نیاز دارد. دلیل اصلی نیاز به کانتینرها امکان جداسازی برنامه‌ها (isolation) از یکدیگر در بهترین سطح ممکن است تا فرآیند تولید، تست و در نهایت اجرا بر روی یک زیرساخت مشترک تسهیل شود.

در مرحله بعد نیاز به کوبرنتیز پیدا می‌شود تا اجرای این کانتینرها بر روی دسته ای^۷ از ماشینها را تا حد زیادی اتوماتیک کند. در واقع کوبرنتیز مانند سیستم عاملیست که بر روی تمام سرورهای شما به صورت یکپارچه اجرا می‌شود و به شاین امکان را می‌دهد که دیگر نگران هیچ ماشینی به طور خاص نباشید. اگر ظرفیت کافی در زیرساخت شما وجود داشته باشد، این سیستم به راحتی می‌تواند از دست دادن یک یا چند ماشین را برای شما به گونه ای مدیریت کند که کاربران هیچ تغییری در سرویسهای در حال اجرا بر روی این بستر احساس نکنند.

¹ Deployment

² Cloud Native Computing Foundation

³ Data Centers

⁴ Go

⁵ Borg

⁶ Hybrid

⁷ Cluster

این سیستم امکاناتی مانند بررسی سلامت^۱ و تکثیر^۲ برنامه‌ها را به راحتی بر روی مجموعه سرورهای شما فراهم می‌کند. از دیگر قابلیت‌های آن نیز ویژگی‌های مناسب و سطح بالا، مانند کشف سرویسها^۳، توزیع بار^۴ و مدیریت پیکربندی^۵ است که برای ساخت سیستم‌هایی با معماری مایکروسروسی حیاتیست و برای تیم‌های شامکان تولید، تغییر و مقیاس‌پذیری بخش‌های مختلف هر سرویس را بر اساس شرایط مورد نیاز فراهم می‌کند. همچنین این سیستم به صورت خودکار کانتینرهایی که خراب شده‌اند را از بین می‌برند و دوباره تولید می‌کنند و اجازه اینکه ترافیک به این کانتینر خراب فرستاده شود، جلوگیری می‌کند. مزیت دیگری که می‌توانیم برای این سیستم نام ببریم، مدیریت منابع حافظه‌ای است که می‌تواند حافظه برای کانتینرها را از طرق مختلف تامین کند. به طور مثال از طریق حافظه داخلی سرور یا حافظه فراهم کنندگان ابری و یا از طریق Network Storage System فراهم کند. در کنار برخی از مزایایی که برای این پلتفرم گفته شد، قابلیت کاربردی و مهمی که این سیستم دارد این است که می‌توان این پلتفرم را توسعه داد و بخش‌های متفاوتی به آن اضافه کرد. این به این دلیل است که ساختار کوبرنتیز هم ماژولار^۶ است و توسعه دهندگان می‌توانند بخش‌های مختلفی را به این پلتفرم اضافه کنند.

اگر چه بسیاری از نرم افزارها سعی می‌کنند این قابلیت‌ها را در سطح برنامه کاربردی پیاده کنند ولی تجربه نشان داده است که این کار با وجود صرف زمان و انرژی زیاد در اکثر موارد منجر به یک راه حل شکننده و غیر قابل نگهداری می‌شود که برای برنامه‌های کاربردی بعدی باید از نو تکرار شود. کوبرنتیز با انتقال این دغدغه‌ها به لایه مناسب و آزاد کردن برنامه کاربردی از قید و بند آنها به شما کمک می‌کند که وقت و انرژی تیم را در جای مناسب و برای تولید ویژگی‌های خاص برنامه کاربردی خودتان صرف کنید.

۲-۳-۲ اجزا و ساختار پلتفرم و گره رهبر در کوبرنتیز

حال در این بخش به ساختار و اجزای کوبرنتیز می‌پردازیم و بخش‌های مختلف این پلتفرم را توضیح می‌دهیم تا دید بهتری نسبت به این پلتفرم پیدا کنیم.

¹ Health check

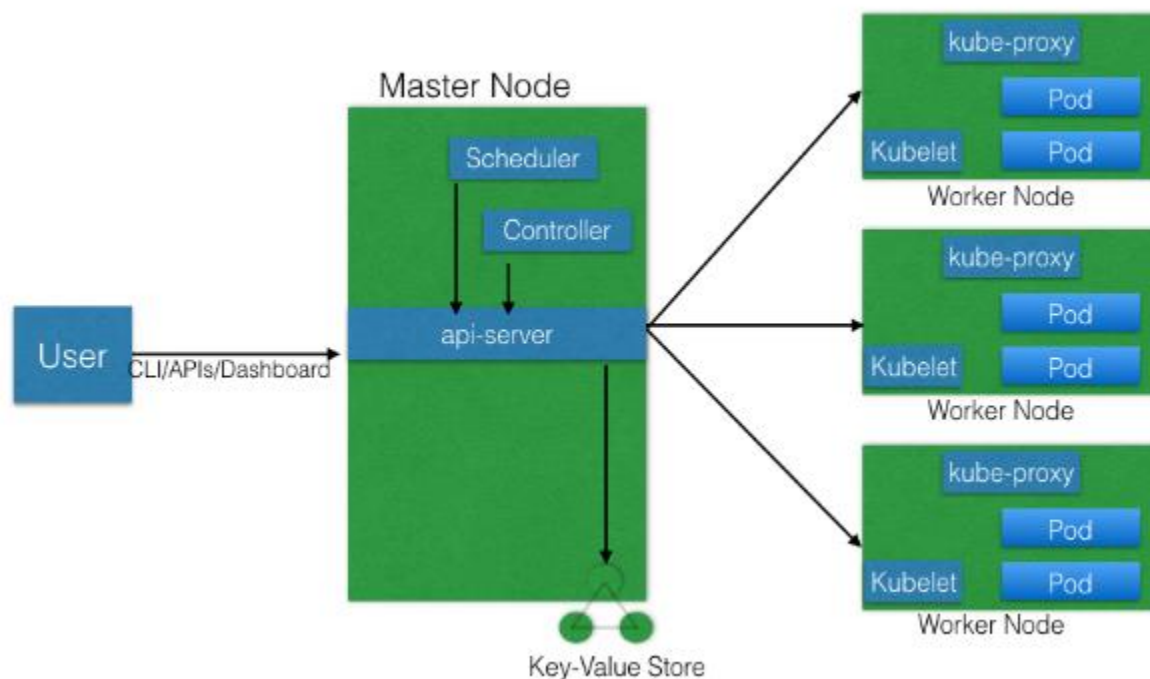
² Replication

³ Service Discovery

⁴ Load Balancing

⁵ Configuration Management

⁶ Modular



شکل ۲-۷: ساختار کلی پلتفرم کوبرنتیز [۱]

همان طور که در شکل بالا مشاهده می‌کنیم، پلتفرم کوبرنتیز شامل چند بخش مهم است که اصلی‌ترین بخش آن گره رهبر^۱ است. این بخش مسئول مدیریت و هماهنگی بخش‌های مختلف کلاستر است و به نوعی همه عملیات‌های کلاستر از طریق این بخش انجام می‌شود. این گره شامل بخش‌های متفاوتی است که در بخش زیر آن‌ها را توضیح خواهیم داد. همه کاربران برای آنکه بتوانند به کلاستر دسترسی پیدا کنند و عملیات‌های مختلف انجام دهند باید از طریق این گره درخواست‌های خود را بفرستند. این درخواست‌ها هم از طریق ترمینال یا داشبورد^۲ و یا API‌های مختلف امکان‌پذیر است.

به دلیل آنکه این گره بسیار نقش اساسی در کلاستر دارد، باید همیشه در دسترس باشد و اگر دچار خطا شود، هزینه زیادی را به شرکت یا سازمان تحمیل خواهد کرد. برای رفع این مشکل، معمولاً چند گره رهبر به کلاستر اضافه می‌کنند و به عبارتی کلاستر در حالت دسترس‌پذیری بالا^۳ قرار می‌دهند و اگر یکی از این گره‌ها دچار مشکل شد، گره‌های رهبر بتوانند سرویس‌دهی را انجام دهند. در مرحله بعدی، سراغ بخش‌های مختلف این گره می‌رویم و توضیح مختصری برای هر کدام ارائه می‌کنیم.

^۱ Master Node

^۲ Dashboard

^۳ High Availability

الف) API server

این نقطه‌ی اصلی مدیریت و ورود به کوبرنتیز است که به کاربر اجازه می‌دهد کوبرنتیز را پیکربندی کند. در واقع kube-apiserver پلی بین اجزای مختلف با هدف نگهداری و حفظ سلامت کلاستر^۱ و انتشار اطلاعات و اجرای دستور عمل‌ها است. api server یک رابط^۲ RESTfull ایجاد می‌کند که به این معنی است که بسیاری از ابزارها و کتابخانه‌ها به راحتی می‌توانند با آن ارتباط برقرار کنند. api server درخواست‌ها را دریافت می‌کند، سپس آنها را تایید و بررسی می‌کند. بعد از آنکه حالت فعلی کلاستر را از یک پایگاه داده توزیع شده به اسم etcd که در بخش بعد توضیح خواهیم داد، خواند و با نتیجه درخواستی که الان آمده است مقایسه کرد، حالت جدید تولید شده از نتیجه درخواست فعلی را در این پایگاه داده ذخیره می‌کند. api server تنها بخشی است که اجازه دارد به این پایگاه داده دسترسی پیدا کند و اطلاعات کلاستر را بخواند یا بنویسد. همچنین در پلتفرم کوبرنتیز این امکان وجود دارد که چند api server وجود داشته باشد و api server اصلی درخواست‌ها را به بقیه api serverها بفرستد و آنها را مدیریت کند.

ب) Scheduler

این بخش مسئولیت این را دارد که آبجکت‌های^۳ جدید مانند پادها را به گره‌ها اختصاص بدهد. برای آنکه scheduler بتواند تصمیم بگیرد که کدام آبجکت را به کدام گره اختصاص دهد، ابتدا باید نیازمندی‌ها و محدودیت‌های هر آبجکت را که در فایل پیکربندیش نوشته می‌شود بررسی کند و همچنین ظرفیت گره‌های کارگر را بررسی کند. scheduler این اطلاعات را باید از پایگاه داده etcd از طریق api server دریافت کند و سپس پس از بررسی این اطلاعات کار اختصاص دادن این آبجکت‌ها به گره‌ها را انجام می‌دهد و حالت فعلی کلاستر را از طریق api server در etcd ذخیره می‌کند. این بخش هم همچنین قابل توسعه است و توسعه‌دهندگان می‌توانند scheduler های خود را به کوبرنتیز اضافه کنند. برای آنکه آبجکت توسط یک scheduler خاص زمانبندی شود باید در فایل پیکربندی آن آبجکت اسم scheduler را گذاشته باشیم در غیر این صورت توسط scheduler پیش فرض زمانبندی می‌شود.

¹ Cluster² Interface³ Object

ج) Controller managers

این کنترلرها مسئولیت این را دارند که وضعیت فعلی کلاستر مطابق وضعیت مطلوب باشد. به طور مثال اگر در فایل پیکربندی نوشته شده است که باید ۳ پاد اجرا شود و در حال حاضر ۱ پاد برای یک آبجکت اجرا می‌شود، باید تعداد پادها را زیاد کند تا وضعیت مطلوب حاصل شود. این کنترلر وضعیت فعلی را از پایگاه داده etcd از طریق api server دریافت می‌کند و به طور مستمر این مقایسه را انجام می‌دهد که کلاستر در وضعیت مطلوب نگه داشته شود.

د) etcd

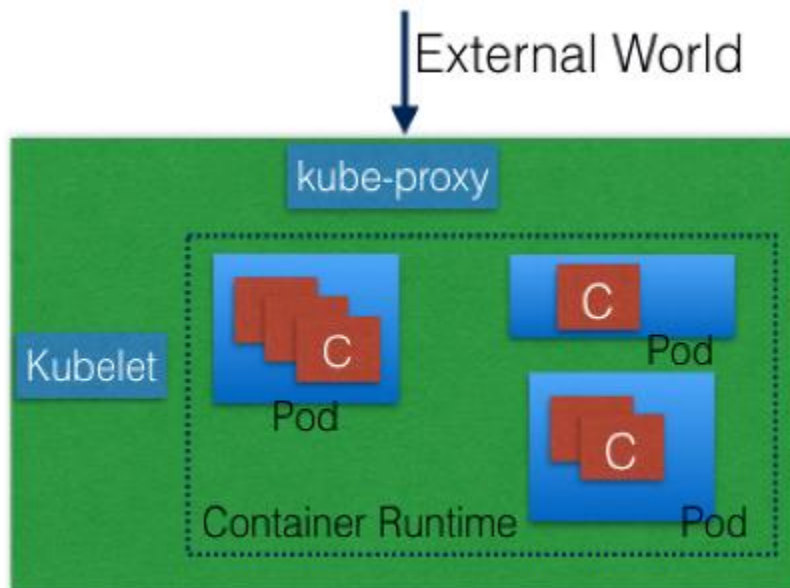
این بخش یک پایگاه داده توزیع شده است که داده‌ها را به صورت key-value نگه‌داری می‌کند. اطلاعات فایل‌های پیکربندی آبجکت‌های مختلف در این پایگاه داده ذخیره می‌شوند که تنها api server می‌تواند به این پایگاه داده دسترسی پیدا کند و اطلاعات را بخواند و بنویسد. این پایگاه داده هم می‌تواند در گره رهبر قرار بگیرد و یا آنکه جدا از این گره باشد و در گره‌های دیگری قرار بگیرد به منظور آنکه اگر گره رهبر دچار مشکل شد یا مورد حمله قرار گرفت، اطلاعات کلاستر از بین نرود. همچنین کوبرنتیز قابلیت‌های snapshot, backup و بازیابی اطلاعات را برای این پایگاه داده قرار داده است تا اطلاعات به راحتی از بین نرود. نکته ای که در این بخش مهم است گفته شود، در مورد آبجکت‌ها است. در کوبرنتیز قسمت‌های مختلف این پلنفرم را آبجکت مینامیم و آبجکت‌های مختلفی برای کارهای مختلف وجود دارد که با نوشتن فایل پیکربندی^۱ این آبجکت‌ها، می‌توانیم از قابلیت‌های مختلف این پلتفرم استفاده کنیم.

۲-۳-۳ معماری و اجزا تشکیل دهنده گره کارگر

بعد از اتمام بخش گره رهبر سراغ گره کارگر^۲ می‌رویم و بخش‌های مختلف این گره را بررسی می‌کنیم.

^۱ Configuration File

^۲ Worker Node



شکل ۲-۸: شمای کلی گره کارگر در پلتفرم کوبرنتیز [۱]

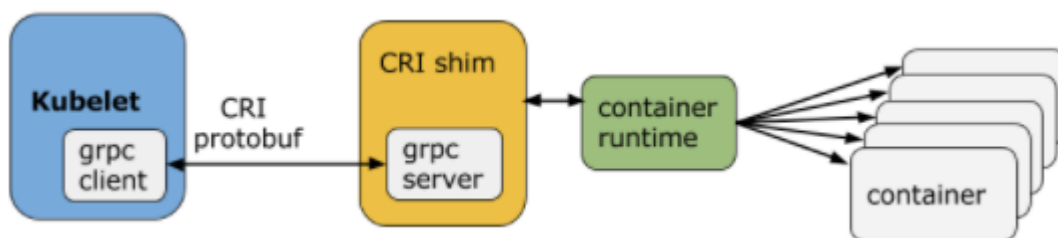
مایکروسرویس‌ها در این گره‌های کارگر اجرا می‌شوند و در آبجکت پاد که کوچک‌ترین آبجکت پلتفرم کوبرنتیز است، قرار دارند که هر پاد می‌تواند چند کانتینری که عملکردشان بسیار مرتبط با هم است قرار بگیرند که اکثراً در هر پاد یک کانتینر اجرا می‌شود. کاربران با دسترسی به این گره‌ها می‌توانند به مایکروسرویس‌ها دسترسی پیدا کنند و عملیات مورد نظر را انجام دهند. در فصل ۳ انواع آبجکت‌های کوبرنتیز را مورد بررسی قرار می‌دهیم. همانطور که در شکل بالا مشاهده می‌کنیم، گره کارگر دارای سه بخش Container Runtime، Kube-proxy و Kubelet است. در بخش زیر توضیح مختصری برای هر کدام ارائه می‌دهیم.

الف) Container Runtime

اگرچه همانطور که گفتیم کوبرنتیز یک پلتفرم برای مدیریت و اجرای کانتینرها است، ولی این پلتفرم خود به طور مستقیم این کانتینرها را نمی‌تواند مدیریت و اجرا کند. برای اینکار باید از یک Container Runtime برای هر گره کارگر استفاده کند تا کوبرنتیز بتواند پادها را در آن گره اجرا و مدیریت کند. Container Runtime‌های متفاوتی مانند Docker، Containerd و CRI-O وجود دارد که در این پروژه از داکر برای اجرای کانتینرها استفاده می‌کنیم.

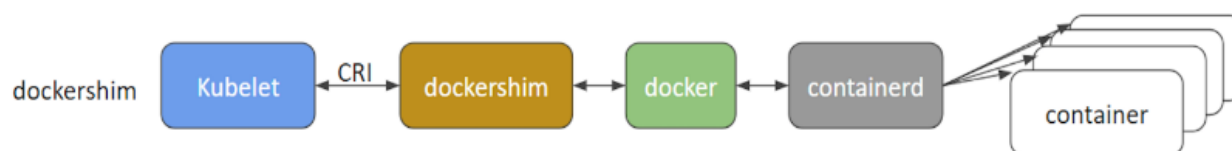
Kubelet(ب)

این عامل در گره کارگر مسئولیت ارتباط با بخش‌های مختلف گره رهبر را دارد و فایل پیکربندی پاد^۱ را از طریق api server دریافت می‌کند سپس به container runtime وصل می‌شود و کانتینرها مشخص شده در فایل پیکربندی را از طریق container runtime اجرا می‌کند. این عامل همچنین سلامت کانتینرهایی که در حال اجرا هستند را مانیتور می‌کند.



شکل ۲-۹: نحوه ارتباط kubelet با container runtime [۱]

همانطور که در شکل بالا مشاهده می‌کنیم، kubelet از طریق Container Runtime Interface توانسته است به Container Runtime متصل شود. این CRI شامل پروتکل‌هایی است که Kubelet از طریق آن‌ها می‌تواند به Container Runtime متصل شود. در اینجا Kubelet به عنوان grpc client عمل می‌کند و به CRI shim که به عنوان grpc server عمل می‌کند، متصل می‌شود و عملیات‌های مربوط به کانتینرها را انجام می‌دهد. CRI دو سرویس اصلی دارد. اولی به عملیات‌های مربوط به ایمج‌ها می‌شود که ImageService نام دارد و دومی عملیات‌های مربوط به کانتینرها و پادها می‌شود که RuntimeService نام دارد. همانطور که قبلاً هم اشاره کردیم، کوبرنتیز با container runtime‌های مختلفی می‌تواند کار کند به شرط آنکه CRI را داشته باشد.



شکل ۲-۱۰: نحوه ارتباط kubelet با محیط اجرای کانتینر برای docker [۱]

به دلیل آنکه از داکر برای اجرای کانتینرهای خود استفاده می‌کنیم، ارتباط kubelet با داکر به صورت شکل بالا است و از طریق CRI مخصوص داکر به داکر متصل می‌شود و داکر هم از طریق containerd کانتینرها را اجرا می‌کند.

^۱ Pod

ج) kube-proxy

این عامل ارتباطاتی در گره کارگر است که IPهای پادها را در IPtables نگه‌داری می‌کند و هنگامی که این IPها تغییر کرد باید جدول را بروزرسانی کند و همچنین درخواست‌هایی که از طرف کاربران به این گره وارد می‌شود را به پادها می‌فرستد. در فصل ۳ بیشتر به نحوه ارتباطات پادها با یکدیگر و محیط بیرون می‌پردازیم.

۲-۴ جمع‌بندی

در این فصل، مزیت‌های کانتینرها نسبت به ماشین‌های مجازی گفته شد و همچنین محیط داکر به منظور اجرا شدن کانتینرها معرفی و توضیح داده شد. در بخش آخر پلتفرم کوبرنتیز و اجزای مختلف آن را توضیح دادیم و نحوه بارگذاری و اجرای داکر بر روی این پلتفرم بیان گردید. دلیل توجه به این موضوع این است که قصد داریم مایکروسرویس‌ها را به صورت ایمج درآوریم و از طریق داکر آن‌ها را اجرا کنیم و با استفاده از پلتفرم کوبرنتیز مدیریت خودکار بر روی این کانتینرها داشته باشیم. عملکرد گره‌های رهبر و کارگر نیز در محیط کوبرنتیز در این رابطه تشریح گردید.

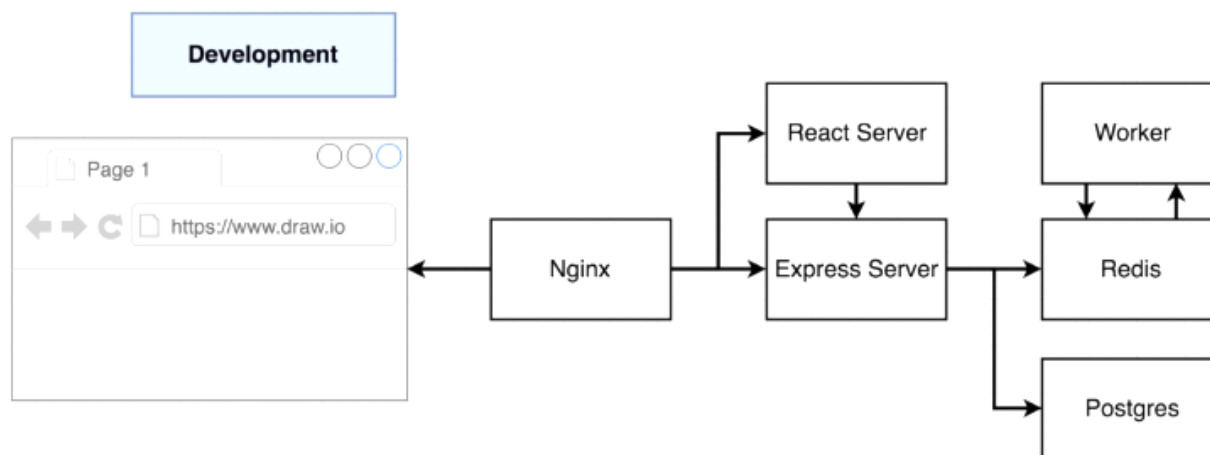
فصل سوم : توضیح مایکروسرویس‌ها و ایمپج کردن آنها در داکر

۳-۱ توضیح مایکروسرویس‌ها و ساختارشان

در این فصل قصد داریم تا سرویس خود را به صورت مایکروسرویس‌ها دریاوریم تا بتوانیم از مزایای این ساختار بهره ببریم و پیچیدگی سیستم خود را کمتر کنیم و همچنین بتوانیم این مایکروسرویس‌ها را در پلتفرم کوبرنتیز بارگذاری کنیم. بعد از اتمام پیاده سازی مایکروسرویس‌ها، باید با استفاده از سیستم داکر ایمپج‌های این مایکروسرویس‌ها را بسازیم و در داکرهاب بارگذاری کنیم تا بتوانیم در گره‌های مختلف از این مایکروسرویس‌ها استفاده کنیم.

برای این پروژه یک سرویس کوچک و ساده را پیاده سازی کرده‌ایم که پیچیدگی سیستم زیاد نشود و بیشتر بتوانیم در ابعاد مدیریت و مقیاس‌پذیری این مایکروسرویس‌ها کار کنیم. برای این منظور سرویس محاسبه فیبوناچی را مد نظر گرفته ایم که این سرویس را به ۶ مایکروسرویس تقسیم بندی کرده‌ایم. یک فرانت اند و بک اند وجود دارد که مایکروسرویس فرانت اند (client) با زبان react است و مایکروسرویس بک اند (server) آن با استفاده از فریم ورکی از node js به نام express نوشته شده است. علاوه بر این دو، یک مایکروسرویس دیگر هم به اسم worker وجود دارد که کار محاسبه کردن فیبوناچی را انجام می‌دهد. دو پایگاه داده مجزا هم داریم که یکی برای ذخیره اعداد وارد شده از طرف کاربر استفاده می‌شود و دیگری برای ذخیره اعداد محاسبه شده از طرف مایکروسرویس worker. هدف از انتخاب دو پایگاه داده مجزا به این منظور بود که مایکروسرویس‌های بیشتری داشته باشیم. مایکروسرویس آخر هم وب سرور^۱ است که درخواست‌های وارد شده از طرف کاربران را به مایکروسرویس‌های مختلف می‌فرستد و مایکروسرویس‌ها فقط از این طریق به دنیای بیرون می‌توانند اتصال یابند. در شکل زیر هم ارتباطات مایکروسرویس‌های مختلف با هم را مشاهده می‌کنیم.

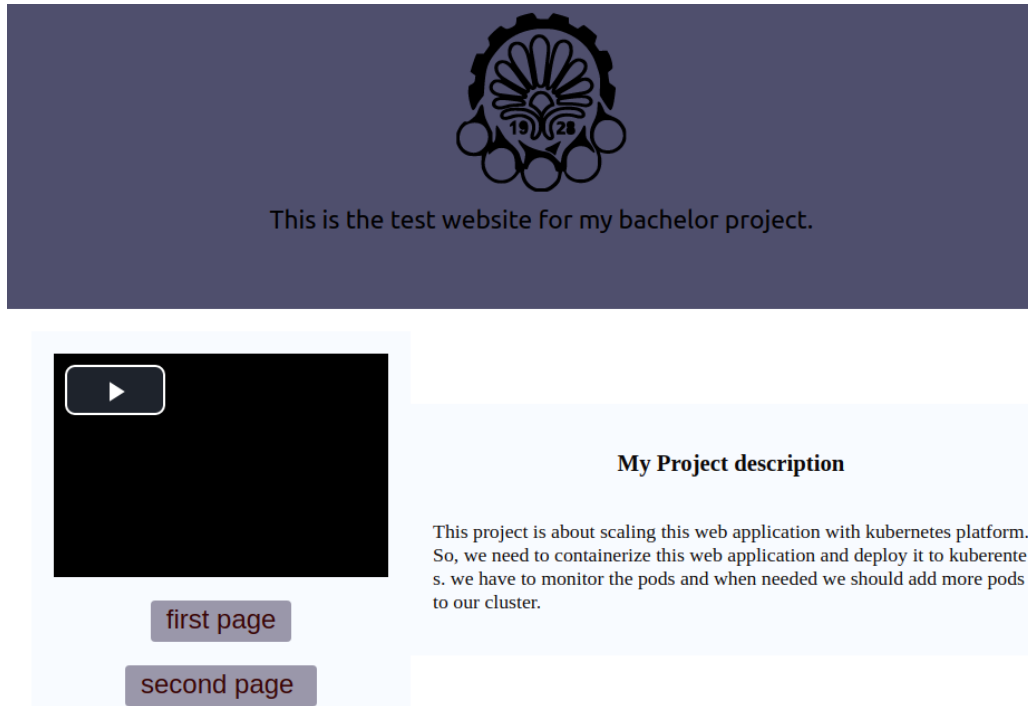
¹ Web Server



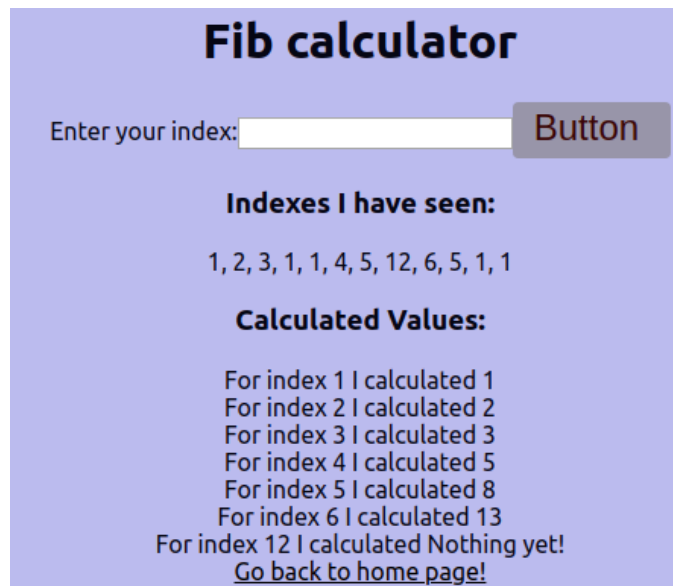
شکل ۳-۱: نحوه ارتباطات مایکروسرویس‌ها با یکدیگر

عملکرد کلی سیستم به این صورت است که کاربر داده را در فرانت اند وارد می‌کند و مایکروسرویس server که مسئول بخش بک‌اند است، داده را هم در پایگاه داده postgres و هم در پایگاه داده redis می‌گذارد. پایگاه داده اولی فقط به این منظور است که به کاربر نشان دهیم تا به حال چه اعدادی را وارد کرده است و دومی برای محاسبه کردن این عدد توسط مایکروسرویس worker است. مایکروسرویس worker، داده را از پایگاه داده redis برمی‌دارد، می‌خواند و محاسبه می‌کند و در آخر نتیجه توسط مایکروسرویس server از پایگاه داده خوانده می‌شود و از طریق مایکروسرویس client که مسئول بخش فرانت اند است، به کاربر نمایش داده می‌شود. در آخر با استفاده از مایکروسرویس nginx مسیریابی را انجام می‌دهیم و هر درخواست را مطابق با مایکروسرویسی که درخواست شده است به مایکروسرویس مد نظر منتقل می‌کند.

شمای فرانت اند که مایکروسرویس client است به این شکل زیر است که دارای دو صفحه است. در صفحه دوم سرویس فیبوناچی مشخص است که شمای این سرویس هم بعد از شکل فرانت اند قرار گرفته است. همان طور که در شکل دوم صفحه بعدی معلوم است، نتایج مایکروسرویس‌های مختلف خود را مشاهده می‌کنیم. در قسمت اول نتایج مایکروسرویس پایگاه داده postgres است که اعدادی که تا به حال کاربر وارد کرده است نشان داده می‌شود که در این بخش مایکروسرویس‌های client، server و postgres دخیل بوده اند. در قسمت دوم نتایج اعداد وارد شده توسط مایکروسرویس worker محاسبه می‌شود و نمایش داده می‌شود.



شکل ۳-۲: شمای صفحه اول مایکروسرویس client

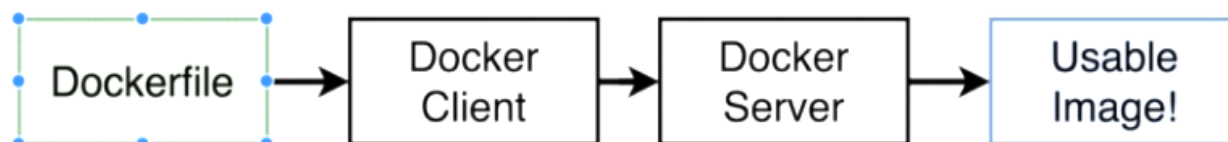


شکل ۳-۳: نتایج مایکروسرویس‌های server, client, worker, redis و postgres

۳-۲ ایمیج کردن مایکروسرویس‌ها

قبل از آنکه مایکروسرویس‌ها را ایمیج کنیم، ابتدا توضیحاتی در مورد مراحل ایمیج درست کردن را ارائه می‌دهیم.

برای آنکه بتوانیم ایمیج خود را بسازیم، باید یک فایل پیکربندی به اسم Dockerfile را درست کنیم که مشخص می‌کند کانتینر چه برنامه‌هایی را شامل می‌شود و چه دستوری را موقع اجرا شدن باید استفاده کند. سپس این فایل پیکربندی را از طریق ترمینال به داکر کلاینت^۱ می‌دهیم و داکر کلاینت هم این فایل را به داکر سرور^۲ می‌دهد. داکر سرور همه خط‌های این فایل را می‌خواند و ایمیج مد نظر را می‌سازد. در شکل زیر مراحل ساخت ایمیج را مشاهده می‌کنیم.



شکل ۳-۴: مراحل ساخت ایمیج (Image) [۲]

۳-۲-۱ نوشتن Dockerfile و توضیح مراحل ایمیج کردن

در این بخش سراغ دستوراتی که باید در این فایل پیکربندی بنویسیم می‌رویم و توضیحاتی درباره هر کدام می‌دهیم. این فایل پیکربندی دارای سه بخش است:

الف) مشخص کردن base image

ب) اجرا کردن دستوراتی برای نصب برنامه‌هایی که در کانتینر می‌خواهیم استفاده کنیم

ج) مشخص کردن دستوری که هنگامی که کانتینر اجرا می‌شود باید آن دستور را اجرا کند

به منظور آنکه این فایل پیکربندی را بهتر توضیح بدهیم، مراحل را از روی ایمیج کردن یکی از مایکروسرویس‌هایی که در این پروژه استفاده کرده‌ایم به نام پایگاه داده redis می‌رویم. ایمیج پایگاه داده redis در داکرهاب^۳ موجود است و در پروژه از این ایمیج استفاده می‌کنیم ولی برای درک بهتر این پروسه، dockerfile این مایکروسرویس را توضیح می‌دهیم.

^۱ Docker Client

^۲ Docker Server

^۳ Docker Hub

```

1  # استفاده از یک ایمج پایه
2  From alpine
3
4  # دانلود و نصب برنامه ها و وابستگی ها
5
6  Run apk add --update redis
7
8  # دستوری که کانتینر هنگام اجرا شدن باید اجرا کند
9
10 CMD ["redis-server"]

```

شکل ۳-۵: Dockerfile مایکروسرویس redis

داکر سرور با خواندن این فایل پیکربندی دستوراتی که برای درست کردن یک ایمج لازم است را میگیرد و ایمج مد نظر را میسازد. دستورات دیگری هم برای Dockerfile وجود دارد ولی این دستورات، دستورات کلی و مهم برای ساختن یک ایمج هستند.

دستور اول به این منظور است که برای اینکه ایمج خود را درست کنیم ابتدا به یک ایمج پایه نیاز داریم که بتوانیم برنامه‌ها و وابستگی‌های مختلف را دانلود و نصب کنیم. این ایمج پایه شامل برنامه‌های مختلفی است که با استفاده از آن می‌توانیم ایمج خود را بسازیم و بدون این ایمج پایه امکان درست کردن ایمج مد نظر خود را نداریم. این دستور مانند این است که می‌خواهیم نرم افزاری را در کامپیوتر خود نصب کنیم. برای این کار ابتدا یک سیستم عامل نیاز داریم تا بتوانیم نرم افزار خود را دانلود و نصب کنیم و سپس اجرا کنیم. پس این ایمج پایه مشابه سیستم عامل در کامپیوتر عمل می‌کند.

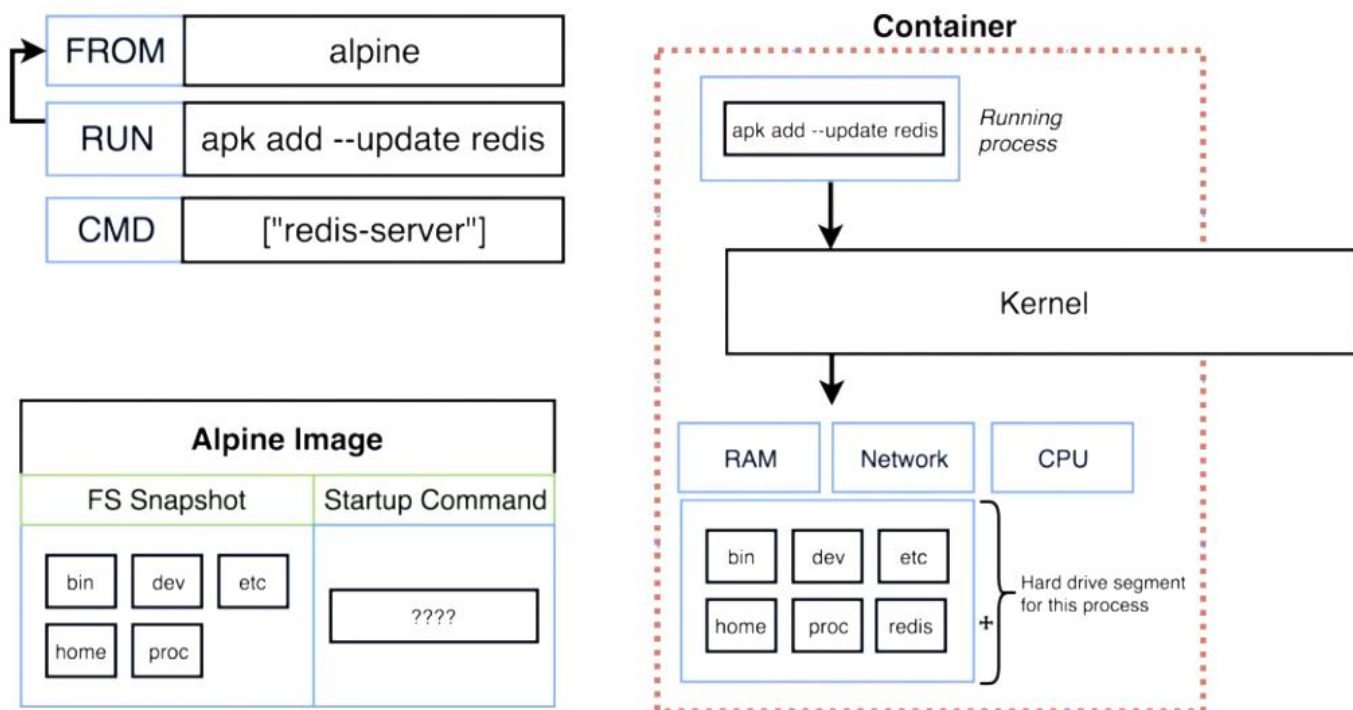
با استفاده از دستور دوم برنامه ای را که می‌خواهیم دانلود و نصب می‌کنیم. در اینجا مایکروسرویس redis را می‌خواهیم دانلود و نصب کنیم که به استفاده از مدیریت کننده بسته^۱ apk که در ایمج پایه alpine نصب شده است این کار امکان پذیر است. در آخر هم با استفاده از دستور CMD مشخص می‌کنیم هنگامی که کانتینر اجرا شد برنامه redis را درون این کانتینر اجرا کند.

بعد از این توضیحات، سراغ ساختن ایمج مد نظر می‌رویم که می‌توانیم با استفاده از دستور زیر این کار را انجام دهیم.

"docker build. "

¹ Package Manager

توجه داشته باشیم که نقطه بعد کلمه build حتما آورده شود تا مراحل ساختن ایمج کردن شروع شود. این نقطه به این معنی است که در همین مسیری که الان در آن هستیم و dockerfile وجود دارد اشاره می‌کند. در گام بعدی، توضیحی در مورد مراحل ساخته شدن ایمج توسط داکر سرور می‌دهیم.



شکل ۳-۶: کانتینرهای میانی برای ساخته شدن نهایی ایمج مد نظر [۲]

هر ایمجی یک فایل سیستم و یک دستور هنگامی که کانتینر شروع به اجرا شد، دارد که در شکل بالا این ساختار را در شکل پایین سمت چپ مشاهده می‌کنیم. در مرحله اول داکر سرور ایمج پایه را دانلود می‌کند و دستور "apk add --update redis" را در کانتینر جدید ساخته شده اجرا می‌کند که این کانتینر میانی اول برنامه‌ها را دانلود و نصب می‌کند. بعد از این عملیات ایمج جدید تولید شده و این کانتینر میانی اول حذف می‌شود. ایمج جدید شامل برنامه‌های ایمج پایه و برنامه redis است. در مرحله آخر این کانتینر میانی دوم با دستور "redis-server" اجرا می‌شود و بعد از ذخیره کردن دستور جدید و برنامه‌های جدید، ایمج مرحله پایانی ساخته می‌شود و کانتینر میانی دوم هم حذف می‌شود. در شکل زیر این مراحل را با جزئیات بیشتری می‌توانیم مشاهده کنیم که Id کانتینرهای میانی و دستورات آمده است.


```
Step 1/3 : From alpine
latest: Pulling from library/alpine
df20fa9351a1: Pull complete

Digest: sha256:185518070891758909c9f839cf4ca393ee977ac378609f700f60a771a2dfe321
Status: Downloaded newer image for alpine:latest
--> a24bb4013296
Step 2/3 : Run apk add --update redis
--> Running in 65161b3cf15f
fetch http://dl-cdn.alpinelinux.org/alpine/v3.12/main/x86_64/APKINDEX.tar.gz
fetch http://dl-cdn.alpinelinux.org/alpine/v3.12/community/x86_64/APKINDEX.tar.gz
(1/1) Installing redis (5.0.9-r0)
Executing redis-5.0.9-r0.pre-install
Executing redis-5.0.9-r0.post-install
Executing busybox-1.31.1-r16.trigger
OK: 7 MiB in 15 packages
Removing intermediate container 65161b3cf15f
--> 3a01024b7b35
Step 3/3 : CMD ["redis-server"]
--> Running in 2781ef0a4bc4
Removing intermediate container 2781ef0a4bc4
--> 5ac635d6b859
Successfully built 5ac635d6b859
```

شکل ۳-۷: مراحل ساخته شدن ایمج نهایی

همانطور که در شکل بالا مشاهده می‌کنیم، اسم ایمج یک عدد است که در اینجا 5ac635d6b859 است که به یاد داشتن این اسم هنگامی که می‌خواهیم این ایمج را با دستور `docker run` اجرا کنیم سخت و مشکل است. برای رفع این مشکل می‌توانیم از قابلیت نامگذاری داکر استفاده کنیم و یک اسم برای این ایمج خود بگذاریم. برای این کار باید مطابق دستور زیر عمل کنیم.

`Docker build -t mpouaykh/redis:v2 .`

ابتدا دستور `-t` را می‌گذاریم و سپس `Docker ID` خود را مشخص می‌کنیم و بعد اسم مایکروسرویس خود و نسخه ای که مد نظر است را مشخص می‌کنیم. همچنین با استفاده از دستور `push` می‌توانیم این ایمج را در داکرهاب بارگذاری کنیم.

`Docker push mpouyakh/redis:v2`

بعد از توضیحاتی که در مورد نحوه ایمج ساختن ارائه کردیم، سراغ نوشتن `Dockerfile` برای مایکروسرویس‌ها می‌رویم. `Dockerfile` برای این مایکروسرویس‌ها کمی متفاوت است ولی ساختار اصلی همان سه بخش اصلی است.

در این بخش می‌خواهیم مایکروسرویس‌های client که با استفاده از react نوشته است و مایکروسرویس‌های server و worker که با استفاده از node js نوشته شده‌اند را ایمج کنیم. برای نوشتن dockerfile این مایکروسرویس باید تغییراتی نسبت به dockerfile قسمت قبل بدهیم. در شکل زیر dockerfile برای این مایکروسرویس‌ها را بررسی می‌کنیم.

```
1 FROM node:alpine
2 WORKDIR "/app"
3 COPY ./package.json ./
4 RUN npm install
5 COPY . .
6 CMD ["npm", "run", "start"]
```

شکل ۳-۸: dockerfile مایکروسرویس‌های server, worker و client

همان طور که مشاهده می‌کنیم، این dockerfile کمی متفاوت‌تر از dockerfile قبلی است. ایمج پایه node است به دلیل آنکه مایکروسرویس‌ها با node js و react js نوشته شده‌اند و برای آنکه مدیریت‌کننده بسته npm را داشته باشیم باید از این ایمج پایه استفاده کنیم. نسخه alpine برای این منظور است که فقط برنامه‌های پایه مانند node و npm در این ایمج وجود دارد و برنامه‌های مختلف و جانبی دیگر وجود ندارد. با استفاده از npm می‌توانیم برنامه‌ها و وابستگی‌هایی^۱ که سرویس‌های نیاز دارند را در کانتینر دانلود و نصب کنیم. با استفاده از ایمج پایه قبلی، داکر سرور دچار مشکل میشد به دلیل آنکه در ایمج پایه قبلی npm نصب نبوده است. در خط دوم مشخص می‌کنیم که برنامه‌ها^۲ و وابستگی‌هایی که دانلود می‌شوند، در کدام مسیر در داخل کانتینر قرار بگیرند. در اینجا مسیر /app را انتخاب کردیم. در خط سوم فایل package.json شامل وابستگی‌هایی که برنامه دارد و باید آن وابستگی‌ها را در کانتینر نصب کنیم تا برنامه درست کار کند و همچنین یک بخش دیگر برای دستوراتی که موقع اجرای کانتینر می‌توانیم استفاده کنیم را شامل می‌شود. در شکل زیر package.json مربوط به مایکروسرویس server را می‌توانیم مشاهده کنیم. با دستور COPY این فایل را در مسیری که در خط ۲ مشخص کردیم کپی می‌کنیم.

"/." اسلش نقطه به این معنی است که در همان مسیری که dockerfile وجود دارد یا مسیری که در خط ۲ مشخص کردیم.

¹ Dependencies

² Programs

در خط ۴ وابستگی‌هایی که در فایل package.json مشخص کردیم را دانلود و نصب می‌کند و در خط ۵ همه فایل‌های مسیر فعلی را در مسیر خط ۲ کپی می‌کند. در آخر هم با استفاده از دستور خط ۶ مایکروسرویس را اجرا می‌کند.

نکته ای در این dockerfile وجود دارد این است که دوبار از دستور copy استفاده کردیم، یکی در خط ۳ و ۵. می‌توانستیم فقط از دستور خط ۵ قبل از خط چهار استفاده کنیم. در این صورت هر بار که تغییری در کد میدادیم، همه ی برنامه‌ها و وابستگی‌ها دوباره دانلود و نصب میشدند. حال برای آنکه دوباره کاری انجام نشود، هر سری که کد خود را تغییر می‌دهیم، باید package.json را جدا کنیم و قبل از دستور RUN قرار بدهیم و بقیه فایل‌های دیگر را بعد از دستور RUN قرار می‌دهیم. این کار باعث می‌شود تا هر بار کد را تغییر می‌دهیم، لازم نباشد همه ی وابستگی‌ها دوباره دانلود و نصب شوند.

```

1  {
2    "dependencies": {
3      "express": "4.16.3",
4      "pg": "7.4.3",
5      "redis": "2.8.0",
6      "cors": "2.8.4",
7      "nodemon": "1.18.3",
8      "body-parser": "*"
9    },
10   ▶ Debug
11   "scripts": {
12     "dev": "nodemon",
13     "start": "node index.js"
14   }

```

شکل ۳-۹: فایل package.json برای مایکروسرویس server

با استفاده از این فایل می‌توانیم وابستگی‌ها و برنامه‌هایی که مایکروسرویس نیاز دارد را مشخص کنیم و با استفاده از دستور "RUN npm install" این‌ها را نصب کنیم. همانطور که در شکل معلوم است برنامه‌های redis و postgres در این بخش آمده اند که مایکروسرویس server با استفاده از این برنامه‌ها به سرور redis و postgres وصل می‌شود. وابستگی دیگر، فریم‌ورک^۱ express است که برای اجرا شدن کدها نیاز داریم. قسمت script دستوراتی که می‌توانیم در کانتینر خود اجرا کنیم آورده می‌شود. دستور اول زمانی استفاده می‌شود که می‌خواهیم تغییرات زیادی در کد خود بدهیم و نمی‌خواهیم هر بار برای تغییرات جدید ایمپج جدید درست

^۱ Framework

کنیم. وقتی از این دستور استفاده می‌شود تغییرات به صورت آنی در کانتینر ظاهر می‌شود به شرطی که یک سری تنظیمات را در فایل docker-compose اعمال کنیم. قسمت دوم که "start" است برای اجرا شدن برنامه node js استفاده می‌شود که در dockerfile مایکروسرویس server از این بخش استفاده کردیم. Package.json برای مایکروسرویس‌های worker و client هم به همین صورت است فقط با کمی تفاوت در وابستگی‌ها. ایمج‌های خود را با دستور "docker images" می‌توانیم در شکل زیر برای ۳ مایکروسرویس خود مشاهده کنیم.

mpouyakh/worker	v1	529086a3ff9f
mpouyakh/server	v1	184b2c81e948
mpouyakh/my-app	v1	6f5bbd466af4

شکل ۳-۱۰ : ایمج‌های ساخته شده برای مایکروسرویس‌های worker, server و client که در اینجا به اسم my-app است هر کدام از ایمج‌ها ID و اسم مخصوص به خود را گرفته اند که می‌توانیم این مایکروسرویس‌ها را بر اساس نامشان اجرا کنیم.

۳-۳ جمع‌بندی

در این فصل، با مایکروسرویس‌هایی که قرار است در پلتفرم کوبرنتیز بارگذاری کنیم و با استفاده از این پلتفرم آنها را مدیریت و اجرا کنیم، آشنا شدیم. اجزا و نحوه ارتباطات این مایکروسرویس‌ها با یکدیگر را شناختیم. در آخر هم نحوه ایمج کردن این مایکروسرویس‌ها با استفاده از سیستم داکر توضیح داده شد. بدون این فصل امکان استفاده از پلتفرم کوبرنتیز ممکن نبود و این فصل بخش مهمی برای آماده سازی استفاده از پلتفرم کوبرنتیز به شمار می‌رود.

فصل چهارم : راه اندازی پلتفرم کورنتیز

بعد از آنکه میکروسرویس ها را نوشتیم و ایمپج کردیم، سراغ راه اندازی کلاستر خود می رویم و پلتفرم کورنتیز را به صورت محلی^۱ راه اندازی می کنیم. بعد از راه اندازی، باید آبجکتهای مختلف در کورنتیز را بسازیم تا بتوانیم این میکروسرویس ها را به صورت خودکار مدیریت کنیم. در این فصل آبجکتهای مختلف پلتفرم کورنتیز را معرفی و توضیح می دهیم. در آخر فایل های پیکربندی این آبجکت ها را توضیح می دهیم و در پلتفرم کورنتیز بارگذاری می کنیم تا این پلتفرم بتواند میکروسرویس ها را شناسایی کند و آنها را مدیریت کند.

۴-۱ راه اندازی کلاستر کورنتیز

در این پروژه از یک گره رهبر و دو گره کارگر استفاده می کنیم. یکی از گره ها یک ماشین فیزیکی مجزا با ۳ هسته cpu و ۴ گیگ رم^۲ است. گره کارگر بعدی یک ماشین مجازی در ماشین فیزیکی است با ۲ هسته cpu و ۴ گیگ رم. سیستم عامل ۳ گره Ubuntu 19.10 است. برای اتصال این گره ها به یکدیگر، از kubeadm استفاده می کنیم.

برای راه اندازی کلاستر خود باید مراحل زیر را اجرا کنیم.

الف) باید داکر بر روی همه گره ها نصب شود با استفاده از دستور زیر این کار را انجام می دهیم.

```
sudo apt-get update \
&& sudo apt-get install -qy docker.io
```

ب) repository های مربوط به پکیج های کورنتیز را با استفاده از دستورات زیر به سیستم خود اضافه می کنیم.

```
sudo apt-get update \
&& sudo apt-get install -y apt-transport-https \
&& curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -
```

```
echo "deb http://apt.kubernetes.io/ kubernetes-xenial main" \
| sudo tee -a /etc/apt/sources.list.d/kubernetes.list \
&& sudo apt-get update
```

^۱ Local

^۲ RAM

ج) سپس باید بسته های نرم افزاری مختلف مربوط به کلاستر کوبرنتیز را نصب کنیم.

```
sudo apt-get update \
&& sudo apt-get install -yq \
kubelet \
kubeadm \
kubernetes-cni
```

kubernetes-cni مربوط به بخش های شبکه ای در کوبرنتیز است. cni مخفف container Network Interface است که مسئول ارتباطات کانتینرها با network driverها است. container runtime مسئولیت تخصیص دادن IP به کانتینرها را به دوش cni میگذارد و cni به network driver متصل می شود و یک IP برای کانتینر میگیرد.

د) باید بسته های نرم افزاری خود را در حالت on hold قرار بدهیم که هنگام بروزرسانی جدید بسته های نرم افزاری، کلاستر بهم نریزد.

```
sudo apt-mark hold kubelet kubeadm kubectl
```

ه) swap را باید در سیستم خود خاموش کنیم چون ممکن است swap memory باعث خطاهای غیر قابل پیش بینی شود. برای این منظور می توانیم از دستور sudo swapoff -a یا اینکه در مسیر /etc/fstab بخش های مربوط به swap را کامنت^۱ کنیم.

ر) باید در مسیر /etc/hosts نامها و IPهای گره های مختلف را برای هر گره وارد کنیم تا این گره ها بتوانند همدیگر را ping کنند.

ف) در مرحله بعد باید دستور زیر را در گره رهبر وارد کنیم.

```
kubeadm init --pod-network-cidr=10.244.0.0/16 --apiserver-advertise-address=192.168.43.159
```

در بخش اول این دستور subnet پادهای خود را مشخص می کنیم و در بخش دوم api server را با استفاده از IP ماشین خود به بقیه گره ها تبلیغ می کنیم تا بتوانند به گره رهبر متصل شوند. بعد از این مرحله، باید دستورات زیر را وارد کنیم تا بتوانیم به کلاستر دسترسی پیدا کنیم.

```
mkdir -p $HOME/.kube
```

^۱ Comment(#)

```
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
```

```
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

با استفاده از flannel نتورکینگ بین پادها را تنظیم می کنیم و با استفاده از دستورات زیر در کلاستر خود این pod network را راه اندازی می کنیم.

```
export ARCH=amd64
```

```
curl -sSL "https://github.com/coreos/flannel/blob/master/Documentation/kube-
```

```
flannel.yml?raw=true" | sed "s/amd64/${ARCH}/g" | kubectl create -f -
```

م) دستور زیر را در گره های کارگر وارد می کنیم تا بتوانیم به گره رهبر متصل بشویم.

```
kubeadm join 192.168.43.159:6443 --token 0gvohk.nvz098qjy0ymupao \
```

```
--discovery-token-ca-cert-hash
```

```
sha256:0561d8a5a0c56a3564303c36075275cb66724bc72a31af3b6dc94d561b2f52c6
```

بعد از انجام این مراحل می توانیم وضعیت گره های خود را با استفاده از دستور "kubectl get nodes" باید به صورت زیر باشند.

NAME	STATUS	ROLES	AGE	VERSION
mpouyakh	Ready	<none>	14d	v1.18.3
mpouyakh-m	Ready	master	112d	v1.18.3
worker-node1	Ready	<none>	14d	v1.18.3

شکل ۴-۱: وضعیت گره ها بعد از راه اندازی کلاستر

همچنین می توانیم با استفاده از داشبورد کوبرنتیز که یک web UI است به کلاستر خود دسترسی پیدا کنیم و راحت تر بتوانیم کلاستر خود را مدیریت کنیم.

برای راه اندازی این داشبورد مراحل زیر را باید انجام دهیم.

الف) ابتدا باید دستور زیر را اجرا کنیم تا قسمت های مختلف این داشبورد در کلاستر نصب شوند.

```
kubectl apply -f
```

```
https://raw.githubusercontent.com/kubernetes/dashboard/v2.0.0/aio/deploy/recommended.yaml
```

ب) با استفاده از آبجکت های service account و ClusterRoleBinding که در فصل بعدی بیشتر در مورد این آبجکت توضیح خواهیم داد یک دسترسی admin به کاربر جدید می دهیم که بتواند به همه منابع و namespaces دسترسی پیدا کند.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: admin-user
  namespace: kubernetes-dashboard
```

شکل ۴-۲: آبجکت Service Account برای کاربر جدید

با استفاده از این آبجکت کوبرنتیز کاربر خود را مشخص می‌کنیم و همچنین namespace این آبجکت هم معین می‌شود.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: admin-user
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: admin-user
  namespace: kubernetes-dashboard
```

شکل ۴-۳: اتصال کاربر جدید به نقش ادمین کلاستر

با استفاده از این آبجکت کوبرنتیز می‌توانیم به کاربر جدید دسترسی ادمین را بدهیم. به دلیل آنکه نقش ادمین قبلاً تعریف شده است دیگر لازم نیست ClusterRole را تعریف کنیم و تنها با استفاده از این فایل کاربر جدید نقش ادمین می‌گیرد.

برای ورود به این داشبورد باید یک token دریافت کنیم که با استفاده از دستور زیر می‌توانیم این token را تولید کنیم.

```
kubectl -n kubernetes-dashboard describe secret $(kubectl -n kubernetes-dashboard get secret |
grep admin-user | awk '{print $1}')
```

سپس با استفاده از این token وارد داشبورد کوبرنتیز می‌شویم.

Kubernetes Dashboard

☐ Kubeconfig
Please select the kubeconfig file that you have created to configure access to the cluster. To find out more about how to configure and use kubeconfig file, please refer to the [Configure Access to Multiple Clusters](#) section.

☒ Token
Every Service Account has a Secret with valid Bearer Token that can be used to log in to Dashboard. To find out more about how to configure and use Bearer Tokens, please refer to the [Authentication](#) section.

Enter token *

Sign in

شکل ۴-۴: وارد کردن token مربوط به داشبورد کوبرنتیز برای وارد شدن

۴-۲ آجکت های کوبرنتیز

۴-۲-۱ RBAC^۱

در کلاستر دو نوع کاربر داریم: یک کاربر حقیقی (User Accounts) که می تواند به کلاستر دسترسی پیدا کند و منابع را تغییر یا ایجاد کند و یک کاربر دیگر (Service Accounts) که برنامه ها و پروسه های درون کلاستر هستند که آنها هم می توانند تغییراتی در کلاستر ایجاد کنند. به منظور آنکه این دسترسی ها را برای این کاربرها یا برنامه ها محدود کنیم از این قابلیت کوبرنتیز استفاده می کنیم. Service Account یک آجکت کوبرنتیز است برای برنامه ها و پروسس ها^۲ که معمولاً توسط api server به صورت خودکار ایجاد می شود. به طور مثال در فصل قبلی برای آنکه داشبورد کوبرنتیز را راه بیندازیم یک service account نوشتیم که مشخص می کند که نام این مایکروسرویس چه باشد و در کدام namespace قرار بگیرد.

برای آنکه بتوانیم با api-server ارتباط برقرار کنیم باید احراز هویت شویم. این احراز هویت برای کاربران حقیقی جدید به صورت client-certificate است. این نوع احراز هویت بر اساس کلیدهای عمومی و خصوصی انجام می شود و باید certificate authority ای وجود داشته باشد که درخواست احراز هویت کاربر جدید را امضا کند تا اجازه پیدا کند به کلاستر دسترسی پیدا کند. برای مثال می خواهیم یک کاربر جدید را اضافه کنیم. ابتدا باید با استفاده از ابزار openssl یک کلید خصوصی و certificate بگیرد و سپس یک certificate signing request با

¹ Role-Based Access Control

² Processes

همین ابزار بسازد و این درخواست را به صورت فایل Yaml درآورد. بعد از آنکه این درخواست توسط CA تایید شد، کاربر جدید به کلاستر اضافه می شود.

برای احراز هویت درخواست های میکروسرویس ها هم باید از service account tokens استفاده کنیم که برای هر میکروسرویس یک token به صورت آبجکت secret که باعث می شوند token های رمزگذاری^۱ شوند و اطلاعات امن تر باشد، به این میکروسرویس اختصاص می یابد. با استفاده از این token ها می توانند با api server ارتباط برقرار بکنند.

بعد از مرحله احراز هویت می توانیم با استفاده از آبجکت های Role و ClusterRole دسترسی های کاربران و میکروسرویس ها را محدود کنیم و فقط بتوانند به بخشی از منابع دسترسی پیدا کنند و بعضی از عملیات ها را انجام دهند. آبجکت Role مشخص می کند که در یک namespace مشخص چه عملیاتی می توانند انجام دهند و به چه منابعی می توانند دسترسی پیدا کنند، ولی آبجکت ClusterRole مشخص می کند که در سطح کلاستر چه عملیاتی می تواند انجام دهد و به چه منابعی می تواند دسترسی پیدا کند.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  creationTimestamp: "2020-06-15T20:24:46Z"
  labels:
    app: nginx-ingress
    chart: nginx-ingress-1.36.2
    heritage: Helm
    release: pouya
  name: pouya-nginx-ingress
  namespace: default
  resourceVersion: "8922157"
  selfLink: /apis/rbac.authorization.k8s.io/v1/namespaces/default/roles/pouya-nginx-ingress
  uid: 3993cc79-f48b-43da-9e0c-5a788de8e73f
rules:
- apiGroups:
```

شکل ۴-۵: آبجکت Role برای میکروسرویس nginx

همانطور که مشاهده می کنیم، آبجکت Role برای میکروسرویس وب سرور نوشته شده است و مشخص شده است که این میکروسرویس به کدام namespace دسترسی داشته باشد. در شکل زیر هم عملیات هایی را که برای برخی از منابع می تواند انجام دهد مشاهده می کنیم.

¹ Encode

```
resources:
- configmaps
- pods
- secrets
- endpoints
verbs:
- get
- list
- watch
apiGroups:
- ""
resources:
- services
verbs:
- get
- list
- update
- watch
```

شکل ۴-۶: عملیات‌هایی^۱ که مایکروسرویس بر روی برخی از منابع می‌تواند انجام دهد

آبجکت Role به تنهایی نمی‌تواند کاری انجام دهد. برای آنکه این آبجکت کار کند باید از آبجکت RoleBinding استفاده کنیم که service account یا کاربران را به namespace ای که مشخص کردیم اتصال بدهد و محدودیت‌ها را اعمال بکند.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  creationTimestamp: "2020-06-15T20:24:46Z"
  labels:
    app: nginx-ingress
    chart: nginx-ingress-1.36.2
    heritage: Helm
    release: pouya
  name: pouya-nginx-ingress
  namespace: default
  resourceVersion: "8922158"
  selfLink: /apis/rbac.authorization.k8s.io/v1/namespaces/default/rolebindings/pouya-nginx-ingress
  uid: f675eeb8-b3f4-466f-87ee-7668fecca908
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: pouya-nginx-ingress
subjects:
- kind: ServiceAccount
  name: pouya-nginx-ingress
  namespace: default
```

شکل ۴-۷: آبجکت RoleBinding برای وصل کردن کاربران یا service account به آبجکت Role

¹ Verbs

آبجکت های ClusterRole و ClusterRoleBinding به همین شکل است فقط با این تفاوت که این آبجکت ها در سطح کلاستر عمل می کنند.

۴-۲-۲ Pod ها، Replica set ها و Deployment ها

ساده ترین واحدی که در کوبرنیتیز وجود دارد Pod ها هستند. در واقع کانتینر به خود ماشین (سرور) تعلق ندارد، بلکه یک یا چند کانتینر در یک Pod قرار دارند. در دید کلی یک Pod شامل یک یا چند کانتینر است که به عنوان یک اپلیکیشن یا میکروسرویس شناخته می شوند. Pod ها شامل کانتینرهایی هستند که با یکدیگر کار می کنند و یک چرخه ی حیات^۱ دارند. همچنین همیشه آن ها باید در یک گره قرار بگیرند. این کانتینرها به یک حافظه داخلی یا خارجی مشخص دسترسی دارند و در یک فضای ارتباطی^۲ قرار دارند.

Pod ها به عنوان یک واحد، مدیریت می شوند و فضا، منابع و IP را با هم به اشتراک می گذارند. معمولا Pod ها شامل یک یا چند کانتینر هستند که هدفشان این است که یک وظیفه را به خوبی انجام دهند. سرویس و برنامه ها زمانی کار می کنند که Pod ها در حال اجرا باشند. به عنوان مثال در یک Pod ممکن است یک کانتینر میکروسرویس اصلی را اجرا کند و کانتینر دیگر در پایگاه داده تغییرات را اعمال و همچنین زمانی که منابع خارجی متصل می شوند، آن را شناسایی می کند.

هنگامی که با کوبرنیتیز کار می کنیم به جای مدیریت مستقیم و تکی Pod ها، گروه هایی از Pod ها را مدیریت می کنیم که شامل Pod های کپی شده هستند. از قالب آماده ی Pod ها استفاده می کنیم و Pod ها را به صورت افقی در گره های کلاستر توزیع می کنیم (کپی می کنیم) که اینکار توسط Deployment و Replication Set انجام می شود.

Deployment یک شیء سطح بالاست که به منظور تسهیل مدیریت چرخه ی حیات برای Pod های کپی شده در کلاستر طراحی شده است. پیکربندی Deployment را به راحتی می توان تغییر داد و تعداد Replica Set ها را تنظیم کرد. به دلیل وجود این ویژگی ها Deployment ابزار بسیار پرکاربردی است که در کوبرنیتیز با آن اغلب کار می کنید.

Deployment ابزاری برای تعریف Pod است که کپی، توزیع یکسان، افزایش و کاهش Pod های در حال اجرا را بر عهده دارد. این یک راه آسان برای Load Balancing، افزایش و کاهش Pod ها در صورت افزایش حجم کاری

¹ Life Cycle

² Network Namespace

در کوبرنتیز است. Deployment می داند که چگونه Pod های مورد نیاز را ایجاد کند، زیرا در فایل پیکربندی خود یک template کپی شده از یک Pod دارد.

Deployment مطمئن می شود که تعداد Pod هایی که در گره باید قرار بگیرد با تعداد Pod هایی که در تنظیمات خودش است منطبق باشد. اگر گره یا Pod های در آن از بین برود، جهت جبران، Controller شروع به ساختن Pod های جایگزین می کند. اگر تعداد کپی ها در پیکربندی Controller تغییر کند، Controller بلافاصله شروع به اعمال تغییرات می کند (تعداد Pod های کپی شده را تغییر می دهد) مانند اضافه یا حذف کردن Pod ها. Deployment همچنین می تواند Pod ها را نیز به نسخه ی جدید خود بروزرسانی کند یا اگر نسخه جدید عملکرد خوبی نداشت به نسخه قدیمی برگرداند.

Replication Set برخلاف Deployment قابلیت Rollback و Rolling Update، چرخه ی بروزرسانی Pod ها را ندارد، در عوض Deployment در سطح بالاتر این ویژگی ها را فراهم می کند. پس نیازی به کنترل replication set نداریم و خود Deployment آن را ایجاد و مدیریت می کند.

حال آبجکت deployment یکی از مایکروسرویس ها به اسم server را با هم بررسی می کنیم.

```

1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: server-deployment
5  spec:
6    replicas: 3
7    selector:
8      matchLabels:
9        component: server
10   template:
11     metadata:
12       labels:
13         component: server
14     spec:
15       containers:
16         - name: server
17           image: mpouyakh/multi-server
18           ports:
19             - containerPort: 5000
20           env:
21             - name: REDIS_HOST
22               value: redis-cluster-ip-service
23             - name: REDIS_PORT
24               value: '6379'
25             - name: PGUSER
26               value: postgres
27             - name: PGHOST
28               value: postgres-cluster-ip-service
29             - name: PGPORT
30               value: '5432'
31             - name: PGDATABASE
32               value: postgres
33             - name: PGPASSWORD
34               valueFrom:
35                 secretKeyRef:
36                   name: pgpassword
37                   key: PGPASSWORD

```

شکل ۴-۸: آبجکت deployment برای مایکروسرویس server

در این آبجکت ابتدا مشخص می‌کنیم که به کدام api server در api متصل شویم و این آبجکت را بسازیم. در خط دوم مشخص می‌کنیم که نوع آبجکت چه است، این نوع می‌تواند service که در بخش بعدی توضیح خواهیم داد و مربوط به شناسایی پادها در کلاستر است، یا می‌تواند pod باشد و یا آبجکت‌های دیگر. در قسمت spec اولی، مشخص می‌کنیم که تعداد پادهایی که می‌خواهیم در کلاستر اجرا شود چه تعداد باشد و همچنین label برای این مایکروسرویس مشخص می‌کند تا بتوان به جای اسم یا ID این آبجکت از این label برای گروه بندی پادها در کلاستر استفاده شود. پادها با استفاده از قسمت template تولید می‌شوند. در این قسمت ایمج مربوط به مایکروسرویس مشخص شده است که پاد از این ایمج درست می‌شود. همچنین مایکروسرویس بر

روی پورت ۵۰۰۰ در داخل کانتینر اجرا می شود. در بخش آخر یک سری Environment variable هایی را باید ست کنیم تا بتوانیم به پایگاه داده های redis و postgres وصل شویم. این variable ها در فایل keys.js که در کانتینر این فایل وجود دارد، آمده است تا بتوانیم به این پایگاه داده ها متصل شویم. همان طور که در فصل قبل مشاهده کردیم میکروسرویس server به این دو پایگاه داده متصل است. همچنین میکروسرویس worker هم به پایگاه داده redis متصل است که بخش اتصال به redis را در فایل پیکربندی آبجکت deployment خود دارد. پورت این میکروسرویس های پایگاه داده مشخص شده است و قسمت value مشخص می کند که این پایگاه داده ها در کدام پادها اجرا می شوند و میکروسرویس بتواند به میکروسرویس های پایگاه داده متصل شود. بخش نتورکینگ پادها در بخش بعدی بیشتر توضیح داده خواهد شد. برای اینکه بتوانیم به پایگاه داده postgres متصل شویم باید یک رمز ایجاد کنیم و با استفاده از این رمز به این پایگاه داده متصل شویم. برای آنکه این پسورد در معرض دید نباشد از آبجکت secret استفاده می کنیم. دستور ایجاد این آبجکت و گذاشتن پسورد در این آبجکت به شکل زیر است.

```
Kubectl create secret generic pgpassword --form-literal PGPASSWORD=12345678
```

بخش --form-literal به این منظور است که این آبجکت را با استفاده از دستور ترمینال اجرا می کنیم و نه به صورت نوشتن فایل و بعد بارگذاری آن در کوبرنتیز.

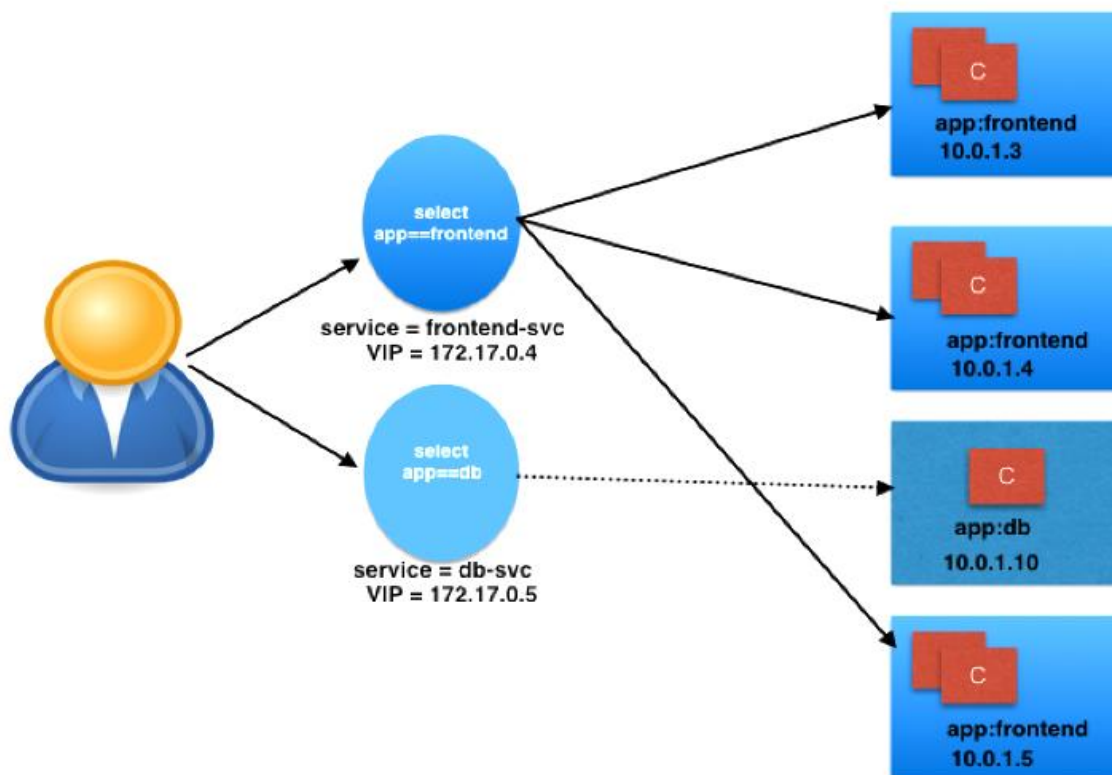
۴-۲-۳ Service ها

Service جزئی است که شبکه سازی بین پادها را در کوبرنتیز ایجاد می کند و ترافیک را به Pod مورد نظر هدایت می کند. یک سرویس شامل گروه هایی منطقی از Pod ها هستند که با هم یک عملکرد را دارند. به عنوان مثال: Web Service, Mail Service.

Service به شما اجازه می دهد تا مسیر مناسب برای رسیدن به کانتینر مناسب فراهم گردد. در عین حال Service به شما اجازه ی گسترش یا جایگزینی را در Backend در صورت نیاز می دهد. یک Service بدون در نظر گرفتن تغییرات در Pod ها مسیریابی آن را انجام می دهد. با اجرا و عملیاتی سازی Service، شما به راحتی می توانید ساختار Container ها را تغییر دهید.

همانطور که میدانیم وقتی پادها ایجاد می شوند، یک IP به آنها تخصیص میابد و اگر بخواهیم بر اساس این IP ها به پادها دسترسی یابیم، دچار مشکل می شویم چون پادها همیشه یک آدرس مشخص ندارند و ممکن است از بین بروند و یک آدرس جدید بگیرند. برای حل این مشکل از آبجکت Service استفاده می کنیم که با استفاده از

label هایی که در deployment استفاده شد، این پادها را گروه بندی می کند و ترافیک را به این پادها میفرستد. با این تکنیک دیگر لازم نیست نگران تغییر آدرس های پادها باشیم.



شکل ۴-۹: نحوه عملکرد آبجکت service در پیدا کردن پادها [۱]

همانطور که در شکل بالا مشاهده می کنیم، هر کدام از پادها یک آدرس مخصوص به خود دارند. ولی آبجکت service این پادها را بر اساس label مدیریت می کند و ترافیک را به این پادها میفرستد. همچنین هر یک از این آبجکت های service یک آدرس مخصوص (clusterIP) به خود میگیرند که بر اساس این آدرس، پادهای دیگر می توانند به پادهای دیگر از طریق این آبجکت دسترسی پیدا کنند. این نوع آبجکت ClusterIP نام دارد که فقط به صورت محلی قابل دسترسی است و ترافیکی از بیرون کلاستر به این پادها نمی توان فرستاد. آدرس های VIP همان آدرس های ClusterIP برای دو اپلیکیشن db و frontend هستند.

این clusterIP ها و آدرس های پادها در جدولی به نام iptables قرار دارند که توسط kube-proxy که در هر گره وجود دارد، مدیریت می شود و این آدرس ها را اضافه یا حذف می کند.

آبجکت service برای میکروسرویس server را در شکل زیر مشاهده می کنیم.


```

1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: server-cluster-ip-service
5  spec:
6    type: ClusterIP
7    selector:
8      component: server
9    ports:
10     - port: 5000
11       targetPort: 5000
12

```

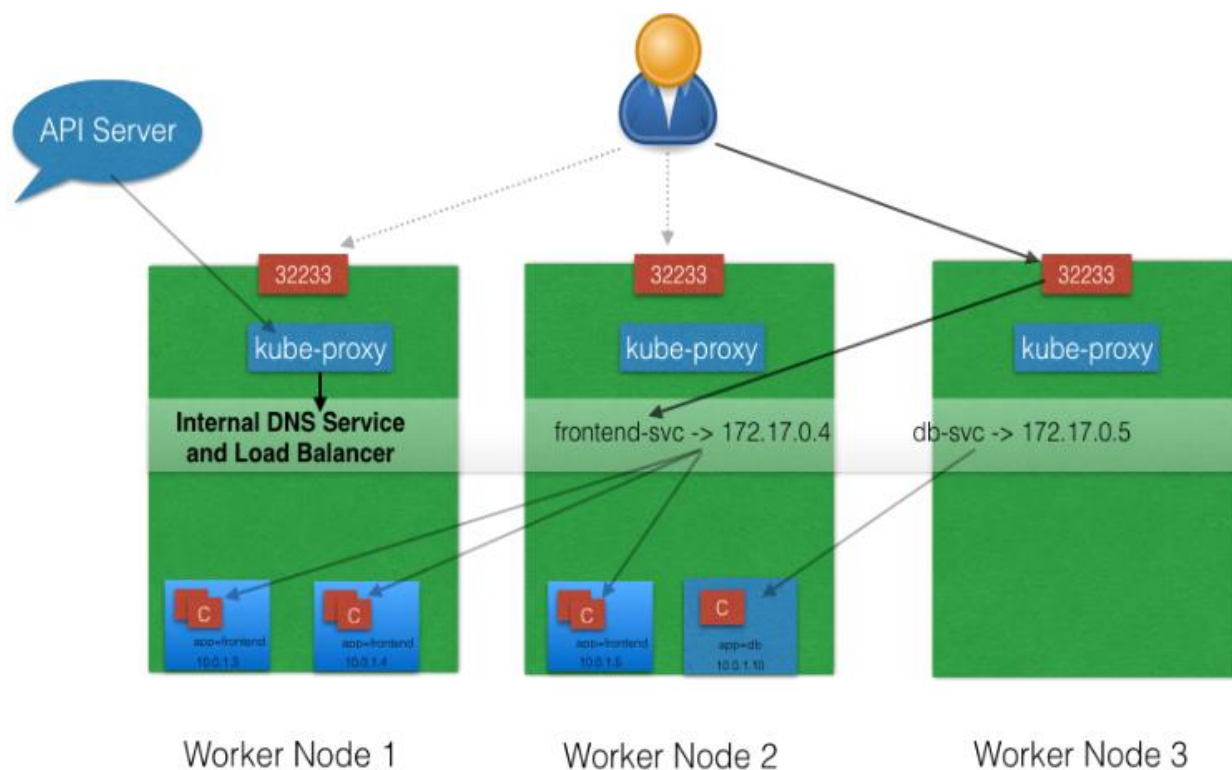
شکل ۴-۱۰: آبجکت service برای میکروسرویس server

در این شکل می توان ساختار آبجکت service را مشاهده کرد. در خط ۶ ام نوع این service را مشخص کردیم که این نوع می تواند مقادیر همچون NodePort و LoadBalancer باشد. در بخش بعدی بیشتر در مورد این دو مورد صحبت خواهیم کرد. در خط ۸، میبینیم همان label ای که در آبجکت deployment برای این میکروسرویس مشخص کرده ایم، آمده است و آبجکت service این پادها را بر اساس label مدیریت می کند و نه بر اساس آدرس پادها. پورتهای^۱ که این میکروسرویس را می خواهیم در معرض کاربران برای دسترسی قرار دهیم، در خط ۱۱ آمده است و targetPort نام دارد. Port در خط ۱۰ برای این منظور است که اگر پادهای دیگر قصد این را داشتند به این پاد دسترسی داشته باشند از طریق این پورت دسترسی پیدا کنند. این پورت می تواند از targetport متفاوت باشد. این آبجکت Service برای بقیه میکروسرویس های این پروژه به همین صورت است و تفاوت آنچنانی نمی کند.

۴-۲-۳-۱ Nodeport و LoadBalancer

به منظور آنکه بتوانیم خارج از کلاستر به پادهای داخل کلاستر دسترسی پیدا کنیم، می توانیم از NodePort استفاده کنیم. با انتخاب پورتهای از بازه ۳۰۰۰ تا ۳۲۷۶۷، می توانیم از طریق هریک از گره های داخل کلاستر به پادهای خود دسترسی پیدا کنیم. در شکل زیر به صورت ملموس تر می توانیم مشاهده کنیم که کاربر با پورت ۳۲۲۳۳ توانسته از طریق یکی از گره ها به اپلیکیشن مد نظر خود دسترسی پیدا کند. این پورت به ClusterIP اپلیکیشن متصل شده است. وقتی کاربر با پورت مشخص به اپلیکیشن می خواهد دسترسی پیدا کند، این دسترسی از طریق service این اپلیکیشن است و آبجکت service است که ترافیک را به این پادها می فرستد.

¹ Port



شکل ۴-۱۱: نحوه وصل شدن کاربر از خارج کلاستر به اپلیکیشن با استفاده از NodePort [۱]

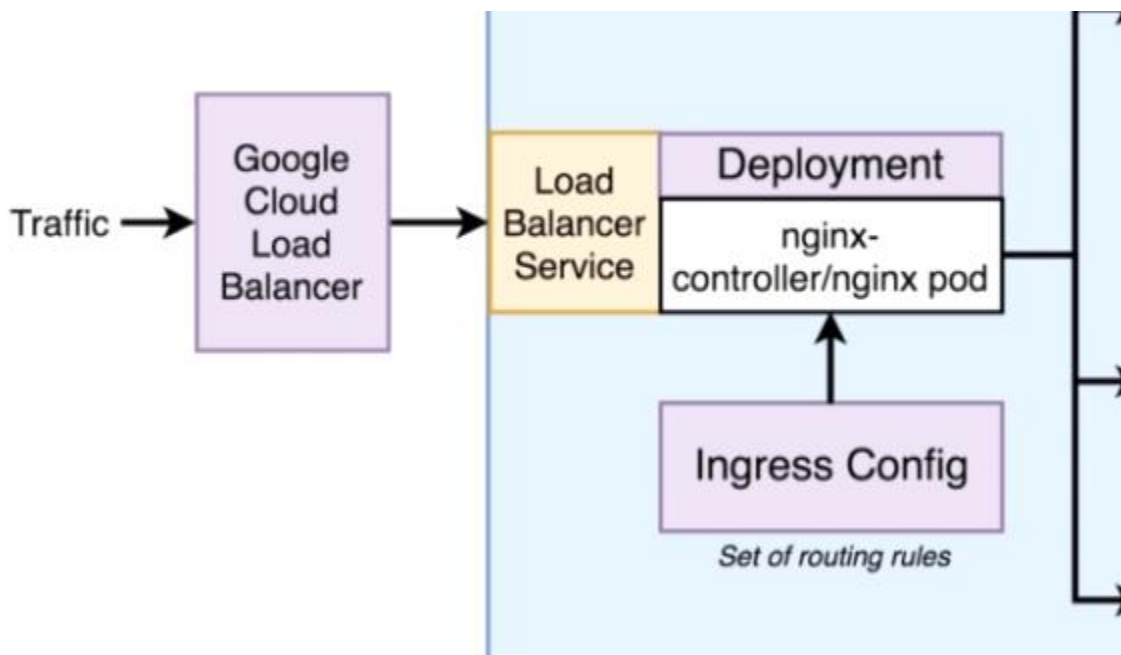
این نوع دسترسی از خارج کلاستر خیلی مطلوب نیست و بیشتر برای تست استفاده می‌شود. روش مناسب‌تری که می‌توانیم برای آنکه از خارج کلاستر به پادها دسترسی پیدا کنیم، استفاده کنیم LoadBalancer است. این Loadbalancer به صورت خودکار، ClusterIP و NodePort را برای اپلیکیشن راه‌اندازی می‌کند و ترافیک‌های خارجی را از طریق یک پورت مشخص به اپلیکیشن می‌فرستد. دو مشکل در این روش وجود دارد: الف) برای استفاده از این LoadBalancer باید حتماً کلاستر خود را در یکی از فراهم‌کننده‌های ابر مانند Google یا AWS^۱ راه‌اندازی کرده باشیم تا بتوانیم از این قابلیت استفاده کنیم. ب) به دلیل آنکه این LoadBalancerها بخش‌های سخت‌افزاری هستند و یک کارت شبکه جدا و IP جدا دارند، هزینه زیادی را برای تحمیل می‌کنند اگر بخواهیم برای هر یک از اپلیکیشن‌های خود از این LoadBalancer استفاده کنیم. برای حل این دو مشکل می‌توانیم از نوع ۴ ام Service استفاده کنیم که Ingress نام دارد.

¹ Amazon Web Services

Ingress ۲-۳-۲-۴

با استفاده از این آبجکت، می‌توانیم اپلیکیشن‌های خود را به محیط بیرون دسترس پذیر کنیم و هنگامی که درخواست‌ها وارد کلاستر می‌شوند از طریق Ingress به اپلیکیشن‌های مختلف می‌فرستد. حال با Ingress لازم نیست برای هر اپلیکیشن خود یک LoadBalancer اختصاص دهیم و تنها می‌توانیم این LoadBalancer را به Ingress متصل کنیم و این آبجکت کار LoadBalancing را در داخل کلاستر انجام دهد.

با استفاده از آبجکت Ingress می‌توانیم فقط Rule‌های مسیریابی را مشخص کنیم و بگوییم هر درخواست به کدام اپلیکیشن باید فرستاده شود. ولی برای آنکه بتوان این مسیریابی را انجام داد باید از یک کنترلر مانند deployment، به نام Ingress-controller استفاده کنیم که آبجکت Ingress را دریافت می‌کند و یک پاد درست کند و کار LoadBalancing را انجام می‌دهد. پروژه Ingress-Nginx یک پروژه آماده است که این Ingress-controller را برای ما فراهم می‌کند.



شکل ۴-۱۲: نحوه کارکرد سرویس LoadBalancer با Ingress [۲]

نحوه کارکرد کلی این دو نوع service با هم را در شکل بالا مشاهده می‌کنیم. LoadBalancer فراهم کننده ابر به یک سرویس LoadBalancer در داخل کلاستر متصل می‌شود و این Service هم ترافیک را به پاد Nginx می‌فرستد که با استفاده از Nginx controller مسیرهایی که این پاد باید بفرستد را مشخص کرده‌ایم.

همان طور که در شکل بالا مشخص است یک فایل پیکربندی مربوط به آبجکت Ingress است که به کنترلر می‌دهیم تا این مسیرها را در پاد پیاده سازی کند و Nginx pod بتواند درخواست‌ها را به میکروسرویس‌های مختلف بفرستد.

فایل پیکربندی Ingress برای این پروژه را در شکل زیر مشاهده می‌کنیم.

```

1  apiVersion: extensions/v1beta1
2  kind: Ingress
3  metadata:
4    name: ingress-service
5    annotations:
6      kubernetes.io/ingress.class: nginx
7      nginx.ingress.kubernetes.io/rewrite-target: /$1
8  spec:
9    rules:
10     - host: mpouyakh.com
11       http:
12         paths:
13           - path: /?(.*)
14             backend:
15               serviceName: client-cluster-ip-service
16               servicePort: 3000
17           - path: /api/?(.*)
18             backend:
19               serviceName: server-cluster-ip-service
20               servicePort: 5000

```

شکل ۴-۱۳: فایل پیکربندی آبجکت Ingress

مانند همه فایل‌های پیکربندی باید api مورد نظر و نوع آبجکت را مشخص کنیم. در خط ۶ به کوبرنتیز اعلام می‌کنیم که می‌خواهیم از پروژه nginx برای مسیریابی در کلاستر خود استفاده کنیم. در خط ۷ مشخص می‌کنیم که بعد از آنکه مسیر مشخص شد، " /api " به "/" تغییر پیدا کند چون در server مسیرهایی که مشخص کرده‌ایم /api ندارند. در بخش spec هم اپلیکیشن‌هایی که می‌خواهیم درخواست‌ها را بفرستیم، معین کرده‌ایم. در بخش host یک domain name به صورت محلی تعریف کرده‌ایم که باید بعداً آدرس ClusterIP برای Ingress را در /etc/hosts برای این domain name بگذاریم. همان طور که معلوم است دو مسیر "/" و "/api" وجود دارد که اولی به ClusterIP میکروسرویس client و دومی به ClusterIP میکروسرویس server اشاره می‌کند. در نتیجه درخواست‌ها به این گونه به این دو میکروسرویس فرستاده می‌شود.

برای نصب پروژه Ingress-Nginx می‌توانیم از مدیریت‌کننده بسته Helm استفاده کنیم. این مدیریت‌کننده بسته به کلاستر دسترسی پیدا می‌کند و برنامه‌های مختلف را در کلاستر نصب می‌کند.

helm install pouya stable/nginx-ingress

با استفاده از این دستور، Helm همه بخش‌های مختلف Nginx را در کلاستر راه اندازی می‌کند.

۴-۲-۴ Persistent Volumes

کانتینرها وقتی اجرا می‌شوند، یک قسمتی از حافظه ماشین به این کانتینر اختصاص می‌یابد. زمانی که این کانتینر در حال اجرا شدن است، داده‌هایی که تولید می‌شوند در فایل سیستم مخصوص این کانتینر قرار می‌گیرد. حال اگر کانتینر از بین رفت یا دوباره شروع به کار کرد آیا این داده‌هایی که توسط کانتینر تولید شده بود باقی می‌ماند؟ جواب خیر است. همه داده‌های کانتینر پاک می‌شود و همه چیز از نو شروع می‌شود. همچنین حافظه‌ای که در داخل پاد است هم به همین صورت است و با از بین رفتن پاد همه داده‌ها پاک می‌شود. برای آنکه بتوانیم این داده‌ها را نگهداری کنیم تا از بین نروند، باید از حافظه‌های مخصوص که خارج از این کانتینرها و پادها قرار دارند استفاده کنیم. نام این آبجکت در کوبرنتیز persistent volume نام دارد که حافظه مورد نیاز برای یک کانتینر را فراهم می‌کند.

برای آنکه بتوانیم این حافظه‌های پایدار را بدست بیاوریم، باید از یک آبجکت دیگر به اسم Persistent Volume Claim استفاده کنیم که با استفاده از این آبجکت میزان حجم حافظه‌ای را که نیاز داریم از کوبرنتیز درخواست می‌کنیم و کوبرنتیز بر اساس منابعی که در اختیار دارد، این حافظه را به پاد مورد نظر تخصیص می‌دهد.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: d-p-v-c
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 3Gi
```

شکل ۴-۱۴: آبجکت PersistentVolumeClaim

با استفاده از این درخواست ۳ گیگ را از کوبرنتیز درخواست می‌کنیم تا به پاد مد نظر اختصاص دهد. دسترسی برای این بخش حافظه می‌تواند به سه بخش تقسیم شود.

الف) ReadWriteOnce : این دسترسی فقط اجازه می‌دهد در لحظه یک کاربر به این حافظه دسترسی داشته باشد و عملیات نوشتن و خواندن را انجام دهد.

ب) ReadOnlyMany : با استفاده از این دسترسی چند کاربر همزمان می‌توانند از این حافظه فقط بخوانند.

ج) ReadWriteMany: این دسترسی اجازه می دهد که چند کاربر در لحظه بتوانند در این حافظه عملیات خواندن و نوشتن را انجام دهند.

پس کوبرنتیز حافظه ای که نوع دسترسی آن را مشخص می کنیم برای پاد مد نظر فراهم می کند. نکته ای دیگری که در این فایل پیکربندی وجود دارد، قسمت StorageClass است که تعیین می کند که آیا فراهم کردن این حافظه به صورت static انجام می شود یا dynamic. Static به این صورت است که خود ادمین باید آبجکت Persistent Volume را بسازد و به api server بدهد. به دلیل آنکه در این قسمت manual آمده است، این به این منظور است که باید توسط خود ادمین این آبجکت تولید شود. برای اینکه به صورت خودکار این آبجکت Persistent volume را بسازیم، می توانیم از NFS-provisioning استفاده کنیم.

۴-۲-۴-۱ Dynamic NFS Provisioning

NFS-client provisioner یک فراهم کننده حافظه به صورت خودکار است. که این فراهم کننده را در کلاستر خود با استفاده از مدیریت کننده بسته Helm که بخش های مختلف مانند RBAC، Deployment و StorageClass را در کوبرنتیز به طور خودکار ایجاد می کند. این فراهم کننده یک پاد در داخل کلاستر است که به آبجکت Storage Class متصل می شود. این فراهم کننده درخواست هایی که برای حافظه به آبجکت Storage Class فرستاده می شود، را می گیرد و از طریق NFS server این حافظه را برای پاد مد نظر فراهم می کند.

به منظور آنکه NFS server را راه اندازی کنیم باید مراحل زیر را انجام دهیم.

الف) `sudo apt install nfs-kernel-server` برای گره مستر و `sudo apt install nfs-common` برای گره های کارگر

ب) `mkdir /svr/nfs/kubedata && sudo chown nobody: /svr/nfs/kubedata`

ج) `Sudo systemctl enable nfs-server && sudo systemctl start nfs-server`

د) در فایل `/etc/exports` خط زیر را وارد می کنیم.

`/srv/nfs/kubedata *(rw,sync,no_subtree_check,no_root_squash,no_all_squash,insecure)`

ه) `sudo exportfs -rav`

برای اینکه NFS-server کار می کند باید از گره کارگر دستور زیر وارد کنیم.

Mount -t nfs 192.168.43.159:/svr/nfs/kubedata/mnt

آدرسی که می بینیم آدرس گره مستر است که nfs-server در آن نصب شده است.

بعد از آنکه این مراحل را انجام دادیم و پاد nfs-provisioning در کلاستر اجرا می شود که با دستور kubectl pod describe وضعیت NFS-server و این پاد را مشاهده می کنیم.

```
Ready:          True
Restart Count:  49
Environment:
  PROVISIONER_NAME: cluster.local/nfs-client-provisioner-1586887262
  NFS_SERVER:      192.168.43.159
  NFS_PATH:        /srv/nfs/kubedata
Mounts:
  /persistentvolumes from nfs-client-root (rw)
```

شکل ۴-۱۵: وضعیت پاد nfs-client-provisioner

همان طور که می بینیم این پاد به NFS-server متصل است و مسیر /persistentvolumes در داخل کانتینر به سرور /svr/nfs/kubedata متصل شده است و درخواست های حافظه ای که از storageclass میگیرد را در این مسیر برای پادهای مختلف ایجاد می کند.

نکته آخر، برای آنکه از این فراهم کننده حافظه استفاده کنیم باید در آبجکت persistent Volume Claim قسمت StorageClass به جای manual، nfs-client قرار بدهیم تا این فراهم کننده بتواند به صورت خودکار برای حافظه مورد نیاز را تامین کند. همچنین در آبجکت فایل Deployment باید به آبجکت Persistent volume claim مد نظر خود در قسمت spec اشاره کنیم تا این حافظه برای پاد مد نظر درست شود.

```
template:
  metadata:
    labels:
      component: postgres
  spec:
    volumes:
      - name: postgres-storage
        persistentVolumeClaim:
          claimName: d-p-v-c
    containers:
      - name: postgres
        image: postgres
```

شکل ۴-۱۶: مشخص کردن PVC برای پاد ای که می خواهد حافظه پایدار داشته باشد

در اینجا برای مایکروسرویس پایگاه داده postgres از persistent volume استفاده کرده ایم تا داده ها از بین نروند. برای آنکه بتوانیم از این حافظه استفاده کنیم باید در آبجکت فایل deployment مایکروسرویس postgres اسم درخواست حافظه ای که قبلا درست کردیم را بیآوریم.

۳-۴ جمع‌بندی

در این فصل کلاستر سه گره ای خود را به صورت محلی راه‌اندازی کردیم و سپس توضیحاتی در مورد آبجکت‌های مختلف کوبرنتیز ارائه دادیم و در آخر فایل‌های پیکربندی نوشته شده مربوط به آبجکت‌های پلتفرم کوبرنتیز را بررسی کردیم. این پلتفرم با استفاده از این آبجکت‌ها می‌تواند مایکروسرویس‌ها را به صورت خودکار مدیریت کند.

فصل پنجم: روش‌های مقیاس‌پذیری خودکار در پلتفرم کوبرنتیز

در این فصل انواع روش‌های مقیاس‌پذیری خودکار^۱ در کوبرنتیز را معرفی می‌کنیم و توضیح می‌دهیم. ولی روشی که برای مقیاس‌پذیری خودکار در این پروژه استفاده می‌کنیم روش سوم که مقیاس‌پذیری خودکار مایکروسرویس‌ها است. روش اول به دلیل آنکه منابع سخت افزاری 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 آنها را با هم استفاده کنیم، در نتیجه نمی‌توانیم از روش دوم در این پروژه بهره ببریم.

۵-۱ مقیاس‌پذیری خودکار تعداد گره‌ها (cluster auto-scaler)

یکی از ابزارهای کوبرنتیز است که به این اجازه را می‌دهد سایز کلاستر خود را که شامل تعدادی گره است، افزایش یا کاهش دهیم. این مقیاس‌پذیر کننده خودکار^۲ در لایه Infrastructure کار می‌کند و هر موقع پادها دچار کمبود منابع شدند و گره‌ای وجود نداشت که این پاد تخصیص یابد، آنگاه این ابزار یک گره به کلاستر اضافه می‌کند و هنگامی که منابع از یک حدی به بعد مورد استفاده قرار نگرفتند، گره‌ها کم می‌شوند تا در مصرف منابع سخت افزاری صرفه جویی شود. این ابزار کلاستر را انعطاف‌پذیر و مقیاس‌پذیر می‌کند. به منظور اینکه این ابزار درست کار کند اقداماتی باید صورت پذیرد.

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

ب) درخواست‌های محاسباتی و حافظه‌ای برای هر پاد باید مشخص باشد به این منظور که این ابزار بتواند مقدار استفاده^۳ 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.

¹ Auto Scaling

² Auto Scaler

³ Utilization

ج) باید یک حدی را مشخص کنیم که تعداد گره‌ها از این مقدار پایین نیایند و باعث نشود که سرویس‌های مهم و اساسی کم یا از بین بروند و معیار دسترس‌پذیری بالا^۱ سرویس‌ها خدشه دار شوند. در کوبرنتیز با استفاده از PodDisruptionBudget می‌توانیم پادها را به گونه‌ای 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 گره‌ای را خاموش کند که این مقدار پاد کمتر از حد مجاز شود، این اجازه را ندهد.

د) درخواست‌های محاسباتی و حافظه‌ای^۲ که برای پادها 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 درخواست‌های نزدیک به واقعیت باشد باید یا از ابزار VPA (مقیاس‌پذیری خودکار تخصیص منابع) استفاده کنیم یا خودمان بر اساس عملکرد مایکروسرویس در مواقع مختلف و زیر بارهای مختلف تشخیص دهیم که چه میزان منابع برای این مایکروسرویس لازم است.

با توجه به اینکه در این پروژه از سه گره استفاده می‌کنیم و این سه گره را به صورت Local (محلی) با استفاده از ابزار Kubeadm راه‌اندازی کردیم و از فراهم آورهای ابری (cloud providers) مانند AWS , Google Cloud و Azure استفاده نکرده‌ایم، امکان استفاده از این ابزار وجود ندارد.

۵-۲ مقیاس‌پذیری خودکار تخصیص منابع (VPA)

این مقیاس‌پذیر کننده خودکار با توجه به منابعی که یک پاد مصرف می‌کند، درخواست‌های محاسباتی و حافظه‌ای مناسبی را می‌تواند هم پیشنهاد بدهد یا خودش آن درخواست‌ها را برای پاد تنظیم کند. بنابراین زمانی که مصرف یک پاد زیاد باشد به آن نسبت درخواست‌ها را افزایش می‌دهد و بالعکس زمانی که مصرف کم باشد درخواست‌ها را کم می‌کند. این کار باعث می‌شود که منابع بهینه مدیریت شوند.

¹ High Availability

² CPU and Memory Requests

بخش مهم این مقیاس‌پذیر کننده خودکار پیشنهاد دهنده^۱ است که با استفاده از Metrics-server که در بخش بعدی توضیح داده خواهد شد، میزان مصرفی محاسباتی و حافظه‌ای را می‌گیرد و بر اساس آن بهترین پیشنهاد خود را می‌دهد.

برای نصب این مقیاس‌پذیر کننده خودکار 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
```

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"
```

شکل ۵-۱: فایل Yaml برای ساختن مقیاس‌پذیر کننده خودکار [۱۷]

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

¹ Recommender

```

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

```

شکل ۵-۲: خروجی مقیاس‌پذیر کننده خودکار تخصیص منابع (VPA) [۱۷]

این مقیاس‌پذیر کننده خودکار همانطور که در شکل بالا آمده است، مقادیر حد پایین، حد مطلوب و حد حداکثر را برای پیشنهاد داده است. اگر updateMode فعال شود مقدار مطلوب را در قسمت درخواست‌های محاسباتی و حافظه‌ای اعمال می‌کند.

```

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

```

شکل ۵-۳: اعمال مقیاس‌پذیری تخصیص منابع برای پاد مد نظر [۱۷]

در اینجا بعد از آنکه این مقیاس‌پذیر کننده خودکار پاد را دوباره میسازد، این درخواست‌ها را در پاد مد نظر اعمال می‌کند و دیگر لازم نیست که بخواهیم خودمان به طور دستی این درخواست‌ها را برای پادها مشخص کنیم.

مشکلی که در این مقیاس‌پذیر کننده خودکار وجود دارد این است که چون هنوز در حالت آزمایشی قرار دارد با مقیاس‌پذیر کننده خودکار مایکروسرویس‌ها هنوز سازگار نیست و نمی‌توان از هر دو همزمان هنگامی که از معیارهای محاسباتی و حافظه‌ای برای مقیاس‌پذیری استفاده می‌کنیم، استفاده کرد و فقط این دو زمانی با هم کار می‌کنند که از custom metrics استفاده کنیم که معیارهایی هستند که خودمان آنها را تعریف می‌کنیم و با استفاده از آنها مایکروسرویس‌ها را مقیاس‌پذیر می‌کنیم.

چون کلاستر کوچک است و خودمان با استفاده از میزان مصرفی پادها در زمان‌های مختلف می‌توانیم درخواست‌های محاسباتی و حافظه‌ای مناسب را برای پادهای خود مشخص کنیم. به علاوه به دلیل آنکه

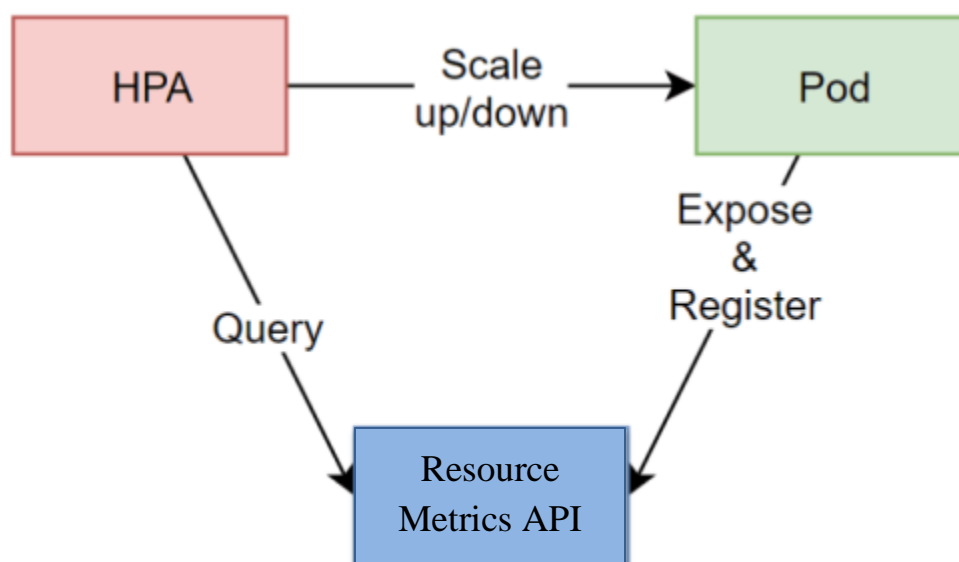
میخواهیم از معیارهای محاسباتی و حافظه‌ای استفاده کنیم، در نتیجه از این مقیاس‌پذیر کننده خودکار در پروژه استفاده نخواهیم کرد.

۳-۵ مقیاس‌پذیری خودکار مایکروسرویس‌ها (HPA)

۱-۳-۵ معیارهای محاسباتی و حافظه‌ای

بخش سوم و بخش مهم این پروژه در این قسمت است که می‌خواهیم مایکروسرویس‌ها را بر اساس معیارهای مختلف مقیاس‌پذیر کنیم به منظور آنکه مایکروسرویس‌ها را پاسخگو^۱ نگه داریم و از تاخیر زیاد و بار زیاد بر روی یک مایکروسرویس جلوگیری شود.

برای اینکه بتوانیم پادهای خود را بر اساس معیارهای محاسباتی و حافظه‌ای مقیاس‌پذیر کنیم، می‌توانیم از metrics server که در کنار api server کوبرنیتیز نصب می‌شود، استفاده کنیم. این ابزار اطلاعات را از گره‌ها با استفاده از cAdvisor که یک بخشی از Kubelet است، جمع‌آوری می‌کند و از طریق Resource Metrics API (metrics.k8s.io) در دسترس HPA قرار می‌دهد که بتواند میزان مصرفی پادها رصد کند و هر موقع میزان مصرفی محاسباتی یا حافظه‌ای از حد مشخص شده رد شد، این مقیاس‌پذیر کننده خودکار شروع به افزایش پادها کند.



شکل ۴-۵: نحوه عملکرد مقیاس‌پذیر کننده خودکار [۱۹]

¹ Responsive

این شکل یک شمای کلی از کارکرد HPA بیان می‌کند که مقیاس‌پذیر کننده خودکار با استفاده از Resource Metrics API اطلاعات محساباتی و حافظه‌ای برای پادها را بدست می‌آورد و کار مقیاس‌پذیری 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 پادهای فعلی را در تقسیم مقدار الان بر مقدار مطلوب ضرب می‌کند. برای آنکه بیشتر مشخص شود. به طور مثال اگر مقدار مطلوب برای یک پاد ۱۰ درخواست در ثانیه باشد و مقدار حال حاضر ۲۰ درخواست بر ثانیه است باید پادها را دو برابر کند و اگر از این مقدار کمتر شد پادها را کاهش می‌دهد.

در این مقیاس‌پذیر کننده خودکار یک مفهومی به نام "Cooldown Period" وجود دارد به این معنی که یک زمانی طول میکشد که این مقیاس‌پذیر کننده خودکار پادها را افزایش دهد یا کاهش دهد. این به این منظور است که شاید پیک‌های گذرا رخ دهد و لازم نباشد که مقیاس‌پذیری 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 مقیاس‌پذیر کننده خودکار از کم شدن بار یا زیاد شدن بار اطمینان حاصل کند و بیهوده کار مقیاس‌پذیری را انجام ندهد. مقدار پیش فرض برای cooldown period ۵ دقیقه است.

در مرحله بعدی به سراغ راه‌اندازی این خود کار ساز با استفاده از metrics server می‌رویم.

۵-۳-۱-۱ نصب metrics server

برای راه‌اندازی metrics-server از مدیریت‌کننده بسته Helm استفاده می‌کنیم که این پکیج منجیر Third-party software است که به کلاستر دسترسی پیدا می‌کند و سرویس‌هایی که نیاز داریم را نصب می‌کند و کار را برای ما بسیار آسان می‌کند و لازم نیست فایل‌های پیکربندی و آبجکت‌های مختلف زیادی را در کلاستر خود بارگذاری و راه‌اندازی کنیم.

ابتدا برای نصب این بسته نرم‌افزاری 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.

hostNetwork: true (الف)

(ب)

```

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'

```


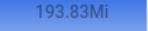
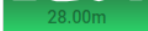




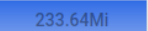

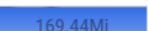
شکل ۵-۵: اضافه کردن پارامترهای اضافی به فایل Yaml سرویس Metrics server

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

بعد از این تغییرات نوبت به نصب این بسته نرم‌افزاری 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
```

می‌توانیم با استفاده از داشبورد کوبرنتیز مقدار مصرفی محاسباتی و حافظه‌ای 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 شکل‌های داشبورد و command line آمده است.

Pods							
Name	Namespac	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Usage (bytes)
✓ api-gateway-5df59d8bc8-ntkh4	default	app: api-gateway pod-template-hash: 5df59d8bc8	mpouyakh	Running	0	 24.00m	 193.83Mi
✓ position-tracker-54fb5494bf-sc4nd	default	app: position-tracker pod-template-hash: 54fb5494bf	mpouyakh	Running	0	 28.00m	 239.49Mi
✓ webapp-785d5b86bf-8j8j8	default	app: webapp pod-template-hash: 785d5b86bf	mpouyakh	Running	0	 1.00m	 6.47Mi
✓ queue-ff85789bd-m2l6x	default	app: queue pod-template-hash: ff85789bd	mpouyakh	Running	0	 84.00m	 233.64Mi
✓ position-simulator-7f4d479d95-54xvv	default	app: position-simulator pod-template-hash: 7f4d479d95	mpouyakh	Running	0	 2.00m	 169.44Mi

شکل ۵-۶: نمای کلی از نمایش معیارهای محاسباتی و حافظه‌ای در داشبورد کوبرنیتی

```
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-svpm	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

شکل ۵-۷: نمایش معیارهای محاسباتی و حافظه‌ای در ترمینال

همانطور که در شکل‌های 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 ۱۰۰۰ m. با انجام این بخش، سراغ مرحله بعد که کار با HPA است می‌رویم و برای یکی از مایکروسرویس‌ها کار مقیاس‌پذیری را انجام می‌دهیم.

برای اینکه مقیاس‌پذیر کننده خودکار در این بخش کار کند، باید برای پادهایی که می‌خواهیم مقیاس‌پذیری خودکار را انجام دهیم درخواست‌های محاسباتی و حافظه‌ای مناسبی را برای پادهای خود تعیین کنیم. به دلیل آنکه در این بخش از مقیاس‌پذیر کردی بر اساس میزان مصرفی cpu و memory استفاده می‌کنیم، نمی‌توانیم همزمان از VPA استفاده کنیم. باید خودمان بر اساس سابقه مصرفی پادهای مورد نظر و منابعی که در کل در اختیار داریم، یک درخواست نزدیک به واقعیت تعیین کنیم.

برای تست این مقیاس‌پذیری از client-deployment استفاده می‌کنیم که بخش فرانت اند وب اپلیکیشن است. سپس با بررسی مقدار مصرفی این پاد و منابعی که در اختیار داریم، درخواست‌ها و محدودیت‌های زیر را برای این پاد در نظر گرفته ایم.

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

شکل ۵-۸: تعیین درخواست‌های محاسباتی و حافظه‌ای برای مایکروسرویس مد نظر

۵-۳-۱-۲ ساختن مقیاس‌پذیر کننده خودکار برای معیارهای محاسباتی و حافظه‌ای

سپس فایل Yaml این مقیاس‌پذیر کننده خودکار 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
```

شکل ۵-۹: فایل Yaml برای ساختن مقیاس‌پذیر کننده خودکار

همانطور که مشخص است در خط اول ابتدا API مد نظر را مشخص کردیم که با استفاده از این API به این مقیاس‌پذیر کننده خودکار 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 معیارها را مشخص می‌کنیم. در این فایل دو معیار cpu و memory را تعیین کردیم که مقدار مطلوب برای cpu مساوی دو برابر (۲۰۰٪) درخواستی که در صفحه قبل مشخص کردیم که برابر ۱۴۰m می‌شود، و برای memory، ۱۷۰ درصد مقدار درخواستی که تعیین کردیم، برابر ۲۵۵ MB است.

بعد از ساختن این آبجکت در کوبرنیتیز، مقیاس‌پذیر کننده خودکار به این شکل درخواست خواهد آمد.

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

شکل ۵-۱۰: نتیجه مقیاس‌پذیر کننده خودکار برای Metrics server

در خط اول این شکل مشاهده می‌کنیم که در قسمت Targets دو معیار آورده شده‌اند 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، کوبرنیتیز تعداد پادها را افزایش خواهد داد و سعی می‌کند مقدار فعلی را به زیر مقدار تعیین شده برسد. همچنین بعد از آنکه بار کم شد این مقیاس‌پذیر کننده خودکار بعد از ۵ دقیقه که زمانی است که مقیاس‌پذیر کننده خودکار صبر می‌کند تا تعداد پادها را کاهش دهد، تعداد پادها را کم می‌کند.

۵-۳-۲ تولید بار (Load Generator)

در این مرحله، سراغ تولید بار می‌رویم که برای این کار، از ابزار siege استفاده می‌کنیم. برای اینکه بتوانیم تشخیص دهیم که توانایی وب اپلیکیشن یا میکروسرویس چقدر است و چه میزان توانایی پاسخگویی دارد باید بر روی آن بار تولید کنیم. با استفاده از این ابزار درخواست‌های HTTP را به میزانی که مد نظر است در یک بازه زمانی مشخص شده برای پاد می‌فرستد. بعد از نصب این بسته نرم‌افزاری در سیستم خود آدرس پاد خود را به این تولیدکننده بار (load generator) می‌دهیم و به طور مثال ۲۰ کاربر همزمان درخواست‌های خود را برای این میکروسرویس 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 در دستور بالا آمده است که بر روی پورت ۳۰۰۰ می‌دهد. نتایج این تولیدکننده بار برای مثال به صورت زیر است.

Transactions:	15 hits
Availability:	100.00 %
Elapsed time:	5.83 secs
Data transferred:	0.58 MB
Response time:	0.37 secs
Transaction rate:	2.57 trans/sec
Throughput:	0.10 MB/sec
Concurrency:	0.94
Successful transactions:	15
Failed transactions:	0
Longest transaction:	1.85
Shortest transaction:	0.16

شکل ۵-۱۱: نتیجه تولیدکننده بار siege [۲۱]

در قسمت اول نشان می‌دهد که چه میزان در کل درخواست انجام شده است. در قسمت دوم میزان دسترس پذیر بودن مایکروسرویس را نشان می‌دهد که چه میزان از این درخواست‌ها^۱ جواب ۲۰۰ گرفته شده است. بخش مهمی که باید در این بخش در رابطه با این تولیدکننده بار ذکر کرد، میانگین زمان پاسخ^۲ کل درخواست‌ها است که در قسمت پنجم شکل آمده است و بخش Concurrency تعداد کاربری که همزمان به مایکروسرویس متصل هستند را نشان می‌دهد. بخش آخر به طولانی‌ترین و کوتاه‌ترین درخواست‌های از کل درخواست‌هایی که فرستاده شده است که در دو قسمت آخر آمده است، اشاره می‌کند.

همچنین این تولیدکننده بار این امکان را به داده است که بتوانیم با استفاده از فایل پیکربندی این تولیدکننده بار که در این مسیر است (~/siege/siege.conf)، قابلیت‌های مختلف این ابزار را استفاده کنیم. به طور مثال چند نمونه از قابلیت‌های مهم این ابزار را در قسمت زیر اشاره می‌کنیم.

اگر در این فایل پیکربندی بخش parser را فعال کنیم یعنی مساوی true قرار دهیم، این تولیدکننده بار فقط یک درخواست HTTP به همان مسیری که مشخص کردیم می‌دهد و دیگر بقیه بخش‌های این وب‌پیج^۳ را که شامل فایل‌های Javascript و CSS و عکس‌ها است، درخواست نمی‌دهد. همچنین می‌توانیم محدودیت تعداد کاربر همزمان را که به طور پیش فرض ۲۵۵ است را تغییر دهیم و افزایش دهیم. در بخش دیگر این فایل

¹ Requests

² Response time

³ Webpage

پیکربندی می‌توانیم با استفاده از فعال کردن قسمت verbose، نتیجه درخواست‌ها را در command line خود مشاهده کنیم. این تولیدکننده بار بین درخواست‌ها 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 عملکرد مایکروسرویس خود را بهتر تست کنیم و توان مایکروسرویس خود را ارزیابی کنیم این تاخیر را صفر در نظر گرفتیم.

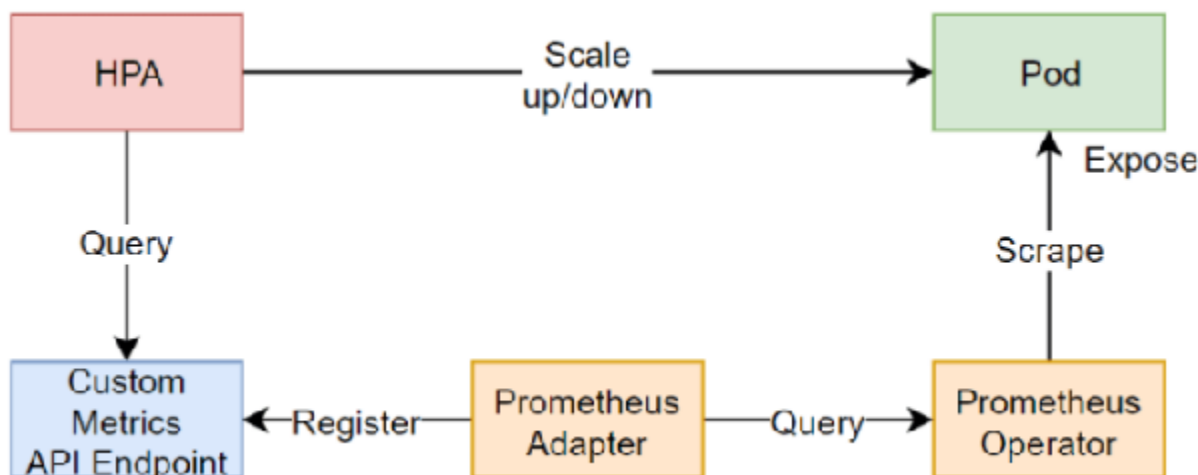
۳-۳-۵ custom metrics

در این بخش قصد این را داریم که اپلیکیشن خود را بر اساس معیارهای بیشتری بتوانیم مقیاس‌پذیر کنیم. در بخش قبل با استفاده از metrics server تنها می‌توانستیم معیارهای محاسباتی و حافظه‌ای پادها را استخراج کنیم و از طریق Resource metrics API در دسترس HPA قرار دهیم. ولی اگر بخواهیم مایکروسرویس‌ها را بر اساس معیارهای دیگری مقیاس‌پذیر کنیم باید سراغ راه‌های 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 معیارهایی که Prometheus به عنوان معیار جمع‌آوری^۱ انجام می‌دهد را از طریق Custom Metrics API در اختیار HPA قرار بدهد تا بتواند بر اساس معیارها کار مقیاس‌پذیری خودکار را انجام دهد. همچنین Prometheus adapter یک فایل پیکربندی به عنوان ورودی دریافت می‌کند که در آن فایل مشخص شده است که چه معیارهایی را می‌خواهیم و چگونه داده‌های آن معیارها را به گونه‌ای که مورد مطلوب هست تحویل بدهد.

مورد بعدی که وجود دارد خود Prometheus، معیارهایی را از گره‌ها و پادها جمع‌آوری می‌کند. ولی بعضی از معیارها مخصوص خود اپلیکیشن هستند و باید در اپلیکیشن تنظیم شوند و سپس آن معیارها را در اختیار Prometheus قرار دهند و Prometheus بتواند داده‌های مربوط به این معیارها را از اپلیکیشن جمع‌آوری کند.

¹ Metrics Collector



شکل ۵-۱۲: نحوه ارتباطات اجزا مختلف برای استخراج Custom metrics [۱۹]

در این شکل به خوبی نشان داده شده است که چگونه با استفاده از Prometheus adapter می‌خواهیم کار مقیاس‌پذیری 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^۱ می‌تواند نشان دهد. سپس Prometheus adapter این معیارها را به گونه‌ای که می‌خواهیم درمیاورد و از طریق API مشخص شده در شکل برای HPA در دسترس قرار می‌دهد.

همانطور که در صفحه قبل اشاره کردیم، برای اینکه بعضی از معیارها را بتوانیم تعریف کنیم و بر اساس آنها مقیاس‌پذیری را انجام دهیم باید کد اپلیکیشن خود را به گونه‌ای تغییر دهیم که بتواند این معیارها را در اختیار Prometheus بگذارد. ولی با استفاده از Linkerd که یک Service Mesh برای کوبرنیتی است، می‌خواهیم بدون آنکه تغییری در مایکروسرویس‌های بدهیم، از بعضی از معیارهای مفید برای مقیاس‌پذیری خودکار استفاده کنیم.

۵-۳-۳-۱ نصب و ساختار linkerd

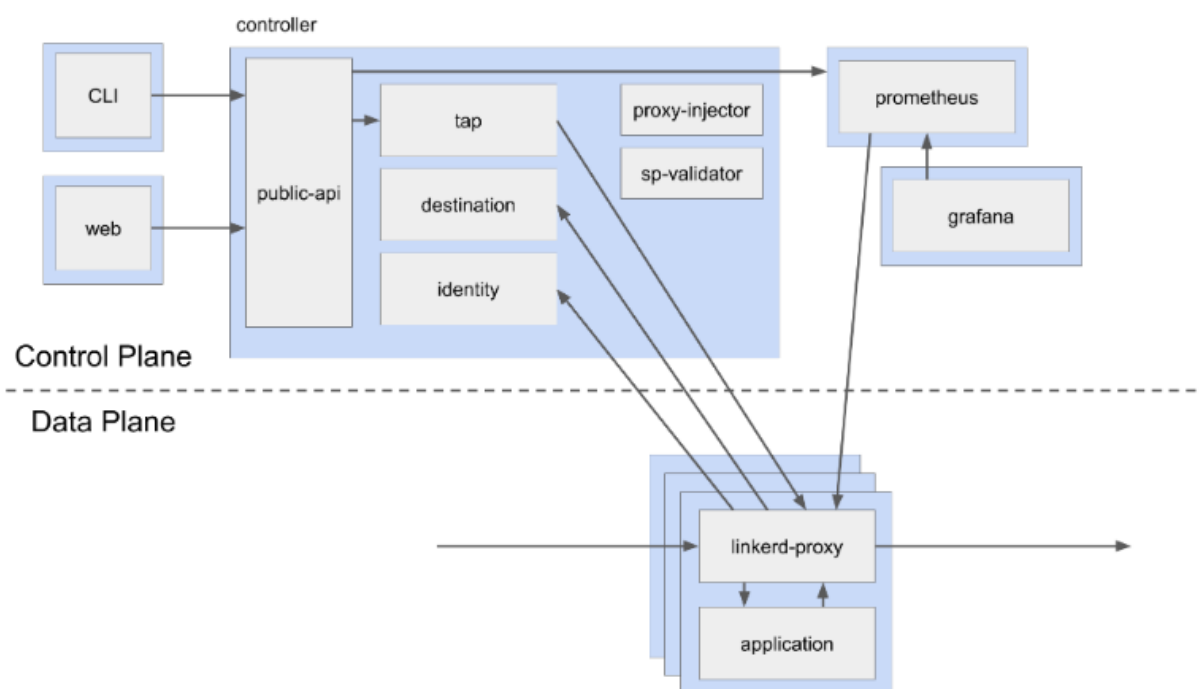
ابتدا توضیح مختصری از اینکه این Service Mesh چگونه کار می‌کند داده می‌شود سپس سراغ استفاده از آن می‌رویم.

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 مایکروسرویس قرار می‌گیرد است و مسئول ارتباطات این پروکسی‌ها است.

^۱ Time series

مسئولیت‌های دیگر این بخش مدیریت، جمع‌آوری اطلاعات از پروکسی‌ها، فراهم کردن ارتباطات بر اساس تکنولوژی^۱ 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 پروکسی‌هایی^۲ است که در کنار مایکروسرویس‌های قرار می‌گیرد و درخواست‌هایی که به مایکروسرویس‌ها می‌شود را دریافت می‌کند و به آنها می‌فرستد سپس بر اساس جواب‌هایی که می‌گیرد، تصمیماتی را می‌تواند بگیرد و آنها را انجام دهد. به طور مثال اگر جواب نگرفت دوباره درخواست را بفرستد و نتیجه جواب‌ها را ذخیره کند یا زمان تاخیر هر مایکروسرویس را محاسبه کند و کارهای دیگر. در شکل زیر شمای کلی این سیستم را مشاهده می‌کنیم.



شکل ۵-۱۳: ساختار کلی سیستم Linkerd [۱۲]

همانطور که در شکل آمده است Control Plane بخش‌های مختلفی دارد که یکی از بخش‌های مهم آن که با آن کار داریم بخش Prometheus است که اطلاعات را از پروکسی‌ها دریافت می‌کند و در اختیار کاربر می‌گذارد. بخش دیگری که استفاده می‌کنیم سیستمی به نام grafana است که ابزار قوی‌ای برای تصویر سازی داده‌ها است و با استفاده از این ابزار می‌توانیم داده‌های جمع‌آوری شده از طریق Prometheus را به طرز مفید و زیبایی

¹ Transport Layer Security

² Proxy

مشاهده کنیم. پس linkerd این دو ابزار مهم را هم برای نصب می‌کند و لازم نیست تا جداگانه این دو ابزار را نصب و راه‌اندازی کنیم.

در گام بعدی، 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 عملیات‌های 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 دسترسی پیدا کرد و با استفاده از دستور زیر همه‌ی پیش‌نیازهای کلاستر را بررسی کرد و همه‌ی پیش‌نیازها آماده بود سراغ مرحله آخر که اضافه کردن پروکسی linkerd به مایکروسرویس‌ها 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 مایکروسرویس خود اضافه کنیم.

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

در شکل زیر اضافه شدن این پروکسی را در مایکروسرویس 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
```

شکل ۵-۱۴: اضافه شدن پروکسی linkerd به پاد مد نظر

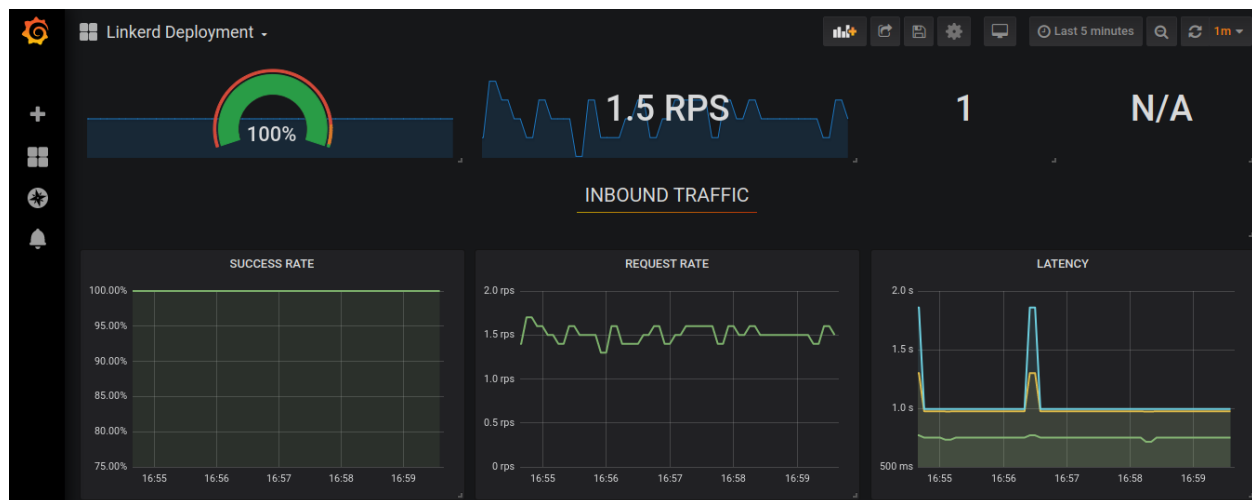
Linkerd همچنین یک web UI دارد که از طریق آن می‌توانیم میکروسرویس‌های اضافه شده را رصد کنیم. در شکل زیر این قابلیت linkerd را هم مشاهده کنیم.

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

Deployment ↑	↑ Meshed	↑ Success Rate	↑ RPS	↑ P50 Latency	↑ P95 Latency	↑ P99 Latency	Grafana
position-tracker	1/1	100.00% ●	1.33	792 ms	1.65 s	1.93 s	⚙️
pouya-nginx-ingress-controller	2/2	100.00% ●	0.4	2 ms	4 ms	4 ms	⚙️
api-gateway	1/1	--	--	--	--	--	⚙️

شکل ۵-۱۵: نمای داشبورد Linkerd

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



شکل ۵-۱۶: نمای داشبورد 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 ۹۹P است، این به این منظور است که تنها یک درصد درخواست‌ها تاخیرشان پایین این تاخیر نوشته شده است و ۹۵P هم به همین شکل یعنی فقط ۵ درصد درخواست‌ها تاخیرشان از این میزان کمتر از این تاخیر ذکر شده است.

۵-۳-۲ معیارهای تاخیر در پاسخ (Response latency) و تعداد درخواست در یک ثانیه (RPS)

حال می‌خواهیم بر اساس یک سری معیارهای مفیدتر دیگری کار مقیاس‌پذیری را انجام دهیم. درست است که معیار محاسباتی و حافظه‌ای تا حدودی اطلاعات خوبی را درباره اینکه چه مقدار بار بر روی یک میکروسرویس 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 میکروسرویس نیست. ممکن است این بالا رفتن CPU به دلیل این است که بعضی از بارهای Memory bound و IO bound هستند. همچنین CPUها مختلف هستند و هر کدام ممکن است مقدار مصرفی که نشان می‌دهند متفاوت باشد و این معیار نسبی است و خیلی دقیق نیست. حال معیاری که دقیق‌تر عملکرد میکروسرویس ما را نشان می‌دهد و می‌توانیم مقیاس‌پذیری خودکار را با استفاده از این معیار انجام دهیم، معیار تاخیر در پاسخ^۱ یک میکروسرویس است که نشان می‌دهد میکروسرویس در چه مدت زمانی جواب کاربر را می‌دهد. شرکت‌ها و کمپانی‌های بزرگ همه در صدد کم کردن میزان این تاخیر هستند تا بتوانند رضایت مشتری را جذب کنند. پس این معیار بسیار اساسی و مهم است.

معیار دیگری که می‌تواند مفید باشد تعداد درخواست‌هایی که در یک ثانیه^۲ به میکروسرویس وارد می‌شود، است و بر اساس چه میزان بار بر روی یک میکروسرویس، این تاخیر در پاسخ را دریافت می‌کنیم. همچنین بر اساس این معیار هم می‌توانیم کار مقیاس‌پذیری را هم انجام دهیم.

بعد از آنکه این دو معیار معرفی شد، سراغ استخراج این دو معیار با استفاده از Prometheus adapter می‌رویم که همان طور در بخش‌های قبل گفته شد مسئول گرفتن اطلاعات از Prometheus است و مرتب کردن اطلاعات به گونه که در فایل پیکربندی مشخص کردیم سپس با در دسترس قرار دادن اطلاعات از طریق API مربوطه برای HPA، کار مقیاس‌پذیری را انجام می‌دهیم.

فایل پیکربندی به صورت زیر است :

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

شکل ۵-۱۷: معرفی سرویس Prometheus به Prometheus adapter

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

¹ Response Latency

² RPS (Request Per Second)

```

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

```

شکل ۵-۱۸: بخش مشخص کردن معیارهای مد نظر در فایل پیکربندی Prometheus adapter

در این بخش قوانین‌های (Rules) خود را مینویسیم و معیاری که می‌خواهیم را مشخص می‌کنیم. همچنین این اطلاعات به چه صورت به 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 با استفاده از تابع histogram_quantile می‌توانیم تاخیر ۵۰p را همانطور که در بخش قبل اشاره کردیم، می‌توانیم محاسبه کنیم. با استفاده از تابع sum هم این داده‌ها را در بازه زمانی ۵ دقیقه جمع می‌کنیم و به تابع histogram_quantile به عنوان ورودی می‌دهیم که تاخیر را محاسبه کند. برای تاخیرهای 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>>
    )

```

شکل ۵-۱۹: تعیین معیار درخواست در ثانیه و مشخص کردن نحوه دریافت اطلاعات این معیار

بعد از آنکه این فایل پیکربندی را نوشتیم، سراغ نصب Prometheus adapter با استفاده از مدیریت‌کننده بسته 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 بخش‌های مختلف linkerd نصب شده است، نصب می‌کنیم و فایل پیکربندی را در هنگام نصب به این adapter می‌دهیم.

حال نگاهی بیندازیم به مقادیری که در custom metrics api که HPA از این طریق می‌تواند به معیارها دسترسی پیدا کند، وجود دارد. با استفاده از دستور زیر می‌توانیم این مقادیر را نگاه کنیم.

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

همان‌طور که در شکل زیر معلوم است در API می‌توانیم اسم این معیارها را ببینیم و همچنین مقادیرشان را مشاهده کنیم که در شکل زیر بعضی از این معیارها 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"
  ]
}
```

شکل ۵-۲۰: مقادیر api Custom metrics

بعد از اینکه توانستیم این معیارها را دسترس پذیر کنیم، سراغ نوشتن فایل Yaml برای مقیاس‌پذیر کننده خودکار 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

```

شکل ۵-۲۱: فایل Yaml برای ساختن مقیاس‌پذیر کننده خودکار

برای این که بتوانیم از custom metrics استفاده کنیم، از apiversion در شکل بالا استفاده می‌کنیم و این API مشخص شده در apiversion به API custom metrics متصل می‌شود و اطلاعات معیارها را دریافت می‌کند. سپس میکروسرویسی را که می‌خواهیم مقیاس‌پذیر کنیم را مشخص می‌کنیم و API ای که از طریق آن در دسترس هست را می‌نویسیم که همان apps/v1 است. در مرحله بعد بازه تعدادی که این میکروسرویس می‌تواند مقیاس‌پذیر شود را مشخص می‌کنیم که این تعداد از یک تا ۱۰ است. با توجه به اینکه در منابع محاسباتی و حافظه‌ای محدودیت داریم، امکان آنکه تعداد بسیار بالا بگذاریم وجود ندارد و تا ۳۰ پاد سیستم به خوبی کار می‌کند و بیشتر از آن، منابع حافظه‌ای و محاسباتی توان تولید پادهای جدید را نخواهند داشت و کوبرنیتی هم اجازه تولید پادهای جدید را به گره‌ها نخواهد داد. در مرحله آخر معیارهایی که می‌خواهیم بر اساس آنها مقیاس‌پذیری را انجام دهیم را معین می‌کنیم و سپس حدی را که اگر از آن گذشت کار مقیاس‌پذیری 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 میکروسرویشان مناسب است و از لحاظ رقابتی چه میزان تاخیر مناسب است. در اینجا برای تست تا ۵۰۰ میلی ثانیه را زمان مناسبی برای تاخیر در پاسخ میکروسرویس خود در نظر گرفته ایم و اگر از این حد گذشت، مقیاس‌پذیر کننده خودکار 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 پادهای اضافه شده تقسیم کند تا عملکرد بهتری از نظر پاسخ دهی داشته باشد.

۵-۳-۳ نحوه انجام تست‌ها برای custom metrics

در این بخش می‌خواهیم تست‌هایی را انجام دهیم و بررسی کنیم که آیا با افزایش تعداد پادها توسط مقیاس‌پذیر کننده خودکار، زمان پاسخ مایکروسرویس کمتر خواهد شد و همچنین آیا تعداد بیشتری درخواست را این مایکروسرویس می‌تواند پاسخ بدهد. برای این تست چند سناریو^۱ مشخص را بر اساس محدودیت‌های نرم افزاری و سخت افزاری که داریم، مشخص کرده‌ایم. با استفاده از تولیدکننده 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 پادها را برای مقیاس‌پذیر کننده خودکار محدود می‌کنیم تا بهتر بتوانیم نسبت افزایش پاد و کم شدن تاخیر و سایر معیارهای بدست آمده را بررسی کنیم. تعداد ماکزیمم پادها را ۱، ۳، ۶ و ۱۰ در نظر گرفتیم که مقیاس‌پذیر کننده خودکار بر اساس این تعداد پادها محدود خواهد شد. به این معنی که برای هر یک از آن چهار تعداد مشخص کاربر ما، مقیاس‌پذیر کننده خودکار خود را به ۱ پاد، ۳ پاد، ۶ پاد و ۱۰ پاد محدود می‌کنیم و تا سقف همین تعداد امکان تولید پادها را دارد و نه بیشتر. در آخر، نتایج هر بخش را جداگانه دریافت می‌کنیم و بر اساس نتایجی که از تولیدکننده بار برای این ۱۶ باری که این تولیدکننده بار را اجرا کردیم گرفته ایم، 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 مایکروسرویس خود و تعداد درخواست در ثانیه را می‌توانیم دریافت کنیم و بر اساس سیستم Grafana، گراف‌های این معیارها را می‌توان برای زمان‌های مختلف مشاهده کرد. سپس برای هر اجرای تولیدکننده بار بر اساس تعداد کاربرهای متفاوت و محدودیت‌هایی که برای پادها گذاشته ایم، دو گراف از تاخیر پاسخ و تعداد درخواست بر ثانیه از linkerd دریافت می‌کنیم. به علاوه تولیدکننده بار هم نتایج مختلفی را مانند میزان دسترس‌پذیری به می‌دهد که در بخش Load generator این نتایج را معرفی کردیم. بنابراین بر اساس نتایج تولیدکننده بار و این دو گراف بدست آمده از linkerd و Grafana، تحلیل‌های خود را انجام می‌دهیم.

۵-۴ جمع‌بندی

در این فصل سه روش اصلی برای مقیاس‌پذیری خودکار در کوبرنتیز را توضیح دادیم و پیاده‌سازی روش سوم را که بخش اصلی این پروژه هست را به طور دقیق و مفصل شرح دادیم. همچنین اینکه چگونه با استفاده از معیارهای مختلف مایکروسرویس‌ها را در پلتفرم کوبرنتیز مقایس پذیر کنیم، شرح داده شد.

¹ Senario

فصل ششم: نتایج مقیاس‌پذیری خودکار مایکروسرویس‌ها

در این فصل نتایج بدست آمده از مقیاس‌پذیری بر اساس معیارهای مختلف را به نمایش خواهیم گذاشت. در ابتدا نتایج برای معیارهای محاسباتی و حافظه‌ای را نشان می‌دهیم. سپس نتایج مقیاس‌پذیری خودکار مایکروسرویس‌ها بر اساس معیارهای زمان پاسخ و تعداد درخواست در ثانیه را به صورت گراف‌ها و جداول مختلف نشان خواهیم داد و تحلیل و بررسی خواهیم کرد.

۱-۶ نتایج مقیاس‌پذیر کننده خودکار برای معیارهای محاسباتی و حافظه‌ای

نتایج عملکرد این مقیاس‌پذیر کننده خودکار برای معیارهای محاسباتی و حافظه‌ای را در عکس‌های زیر مشاهده می‌کنیم. برای اینکه میزان مصرفی محاسباتی مایکروسرویس زیاد شود از ابراز siege که قبلاً توضیح داده بودیم استفاده 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

شکل ۱-۶: خروجی مقیاس‌پذیر کننده خودکار

در خط اول شکل بالا مشاهده می‌کنیم که بار بر روی پاد زیاد شده است و تقریباً ۳,۵ برابر حدی که برای مقیاس‌پذیر کننده خودکار خود تعیین کرده‌ایم شده است. ولی معیار از لحاظ حافظه‌ای کمتر از حد مجاز است و این مقیاس‌پذیری 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

شکل ۱-۷: مقیاس‌پذیری مایکروسرویس مد نظر

در مرحله بعدی می بینیم که بار زیادتر شده است و مقیاس‌پذیر کننده خودکار تعداد بیشتری از این پاد را می‌سازد و هنوز مقدار مصرفی پادها بیشتر از حد تعیین شده است که یک پاد مجاز است مصرف کند. پس انتظار می‌رود که مقیاس‌پذیر کننده خودکار پادهای بیشتری تولید کند.

وقتی ماکزیمم تعداد مجاز پادها تولید شد، می‌بینیم که درصد مصرفی هر پاد کمتر پایین‌تر آمده است و بار بر روی این پادها 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

شکل ۸-۱: تقسیم بار بر روی مایکروسرویس‌ها و کم شدن مقدار مصرفی هر پاد

بعد از اینکه بار کمتر شد، این مقیاس‌پذیر کننده خودکار شروع به کمتر کردن تعداد پادها بر اساس مرور زمان می‌کند. همانطور که گفته شد حدوداً ۵ دقیقه طول میکشد تا تعداد پادها را کم کند و همین طور که بار کمتر می‌شود، این مقیاس‌پذیر کننده خودکار تعداد پادها را هم کمتر می‌کند تا منابع سخت افزاری بهینه مصرف شوند و اتلاف منابع نداشته باشیم.

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

شکل ۹-۱: کم کردن مایکروسرویس‌ها توسط مقیاس‌پذیر کننده خودکار بعد از کم شدن بار

برای معیار حافظه‌ای هم به همین صورت عمل می‌کند و وقتی پاد از حدی که مشخص کردیم بیشتر حافظه مصرف کرد، تعداد پادها 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 مایکروسرویس از تولیدکننده بار siege استفاده نمی‌کنیم و در عوض از ابزار stress استفاده می‌کنیم که با استفاده از این ابزار می‌توانیم میزان مصرفی حافظه برای یک مایکروسرویس را افزایش دهیم. برای اینکار ابتدا باید وارد کانتینر شویم که با استفاده از دستور زیر این کار را انجام می‌دهیم.

```
Kubectrl exec -it pouya-nginx-ingress-controller-59f4fdb9dd-sffsd -- 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 مشخص می کنیم که دو thread هر کدام به میزان ۲۰۰ مگابایت شروع به مصرف حافظه بکنند. هنگامی که میزان مصرفی از حد تعیین شده رد شد، مقیاس پذیر کننده خودکار پادها مثل معیار محاسباتی افزایش می دهد و بعد از تمام شدن بار، پادها را کاهش می دهد.

۶-۲ نتایج مقیاس پذیر کننده خودکار برای custom metrics

در این بخش، نتایج این مقیاس پذیر کننده خودکار را برای یک مایکروسرویس خود مورد بررسی قرار می دهیم. این مایکروسرویس همان وب سرور است به نام pouya-nginx-ingress-controller که درخواستها را بین مایکروسرویس های مختلف پخش می کند. این مایکروسرویس نقش مهمی را در سیستم ایفا می کند و اگر درست کار نکند، کاربران نمی توانند به بقیه مایکروسرویس های دسترسی پیدا کنند. پس باید اطمینان حاصل کنیم که تاخیر این مایکروسرویس در حد قابل قبولی است و کار خود را به موقع انجام می دهد. وضعیت این مایکروسرویس را قبل از اعمال بار نگاه می کنیم 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

شکل ۱-۱۰: شمای کلی مقیاس پذیر کننده خودکار

در مرحله بعدی بار را با استفاده از تولیدکننده بار افزایش می دهیم تا نتیجه مقیاس پذیری خودکار را برای مایکروسرویس ها مشاهده کنیم.

ابتدا یک توضیحی در مورد اعداد این مقیاس پذیر کننده خودکار بدهیم که عدد ۴۰ در روبروی مایکروسرویس به این معنی است که اگر از ۴۰ درخواست بر ثانیه بیشتر شد، این مقیاس پذیر کننده خودکار پادها را افزایش دهد. عدد ۲۰۴m هم یعنی ۲۰۴ mili 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 می کند. 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

شکل ۱-۱۱: نتیجه مقیاس پذیر کننده خودکار بعد از اعمال بار

بعد از اعمال این بار میبینیم که میزان پاسخ دهی بیشتر از حد مجاز شده است و باید این مقیاس پذیر کننده خودکار تعداد پادها 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

شکل ۱-۱۲: نتیجه مقیاس پذیر کننده خودکار بعد از مقیاس پذیری مایکروسرویس

همان طور که میبینیم این مقیاس پذیر کننده خودکار در خط دوم شکل بالا تعداد پادها را افزایش داده و میزان تاخیر کمتر از حد مجاز شده است. در شکل عملکرد هر پاد را جداگانه مشاهده می کنیم.

pouya-nginx-ingress-controller-59f4fdb9dd-9jxsr	1/1	100.00% ●	0.42	4 ms	190 ms	198 ms
pouya-nginx-ingress-controller-59f4fdb9dd-djdw1	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

شکل ۱-۱۳: مربوط به نتایج Linkerd

میزان تاخیرهای ۹۵P, ۹۹P و ۵۰P را برای این پادها 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.

۶-۲-۱ نتایج تست ها بر روی مایکروسرویس مد نظر

بعد از آنکه نحوه عملکرد این مقیاس پذیر کننده خودکار را مشاهده کردیم، سراغ انجام سناریوهای مربوط به اعمال بار بر روی مایکروسرویس خود می رویم که به ترتیب این بارها را با تعداد کاربر ۳۰۰، ۴۰۰، ۵۰۰ و ۷۰۰ تست می کنیم و ماکزیمم پادها را هم به ۱، ۳، ۶ و ۱۰ محدود می کنیم. زمان انجام تست ها برای همه ی اجراها ۴ دقیقه است.

الف) بار با ۳۰۰ کاربر:

بعد از اعمال بار با ۳۰۰ کاربر با استفاده از تولیدکننده بار برای ماکزیمم پادهای مختلف، نتایج گوناگونی را استخراج کردیم که به صورت جدول در شکل زیر آمده است.

جدول ۶-۱: نتایج تولیدکننده بار برای ۳۰۰ کاربر

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

ب) بار با ۴۰۰ کاربر:

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

جدول ۶-۲: نتایج تولیدکننده بار برای ۴۰۰ کاربر

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

(ج) بار با ۵۰۰ کاربر:

جدول ۳-۶: نتایج تولیدکننده بار برای ۵۰۰ کاربر

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

(د) بار با ۷۰۰ کاربر:

جدول ۴-۶: نتایج تولیدکننده بار برای ۷۰۰ کاربر

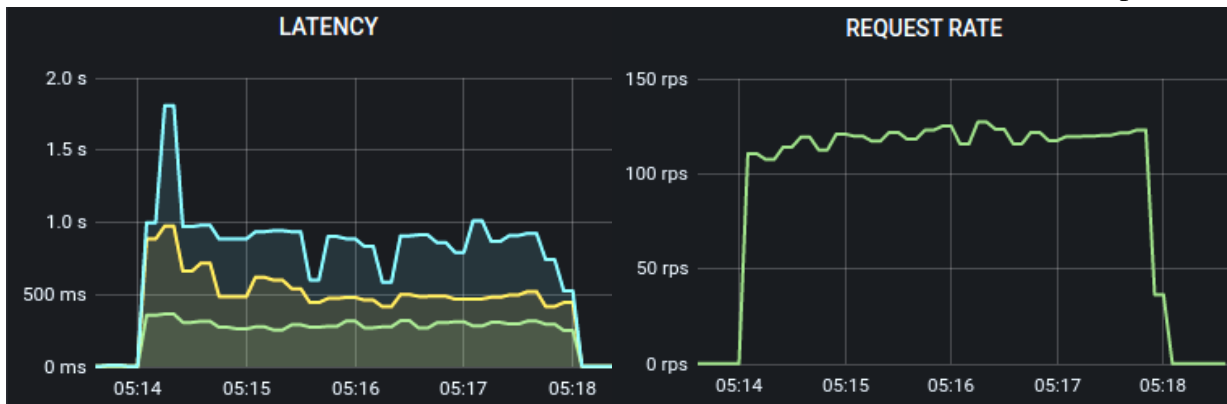
	Max pods = 1	Max pods = 3	Max pods = 6	Max pods = 10
Transactions	26705 hits	42071 hits	57885 hits	57457 hits
Availability	96.74%	98.03%	99.01%	98.95%
Response Time	5.97 secs	3.87 secs	2.80 secs	2.84 secs
Transaction Rate	111.45 trans/sec	175.32 trans/sec	241.54 trans/sec	239.59 trans/sec
Concurrency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

بعد از آنکه نتایج تولیدکننده بار را مشاهده کردیم سراغ نتایجی که Linkerd نشان داده است می‌رویم. سیستم Linkerd نتایج زمان پاسخ و تعداد درخواست در ثانیه را برای مایکروسرویس را با استفاده از Grafana به صورت شکل‌های زیر به نمایش می‌گذارد که در شکل‌های پایین این نتایج برای تعداد کاربر و ماکزیمم پادها نشان داده

می‌شود. دلیلی که میزان زمان پاسخ برای تولیدکننده بار و Linkerd متفاوت است، این است که زمان پاسخ در نتیجه تولیدکننده بار، مساوی جمع زمان پاسخ وب سرور و پاسخ مایکروسرویزی که وب سرور آن درخواست را به آن مایکروسرویس داده است. در صورتی که Linkerd تنها زمان پاسخ خود وب سرور را می‌دهد که همان مایکروسرویس nginx است.

الف) تعداد کاربر ۳۰۰:

Max pods = 1



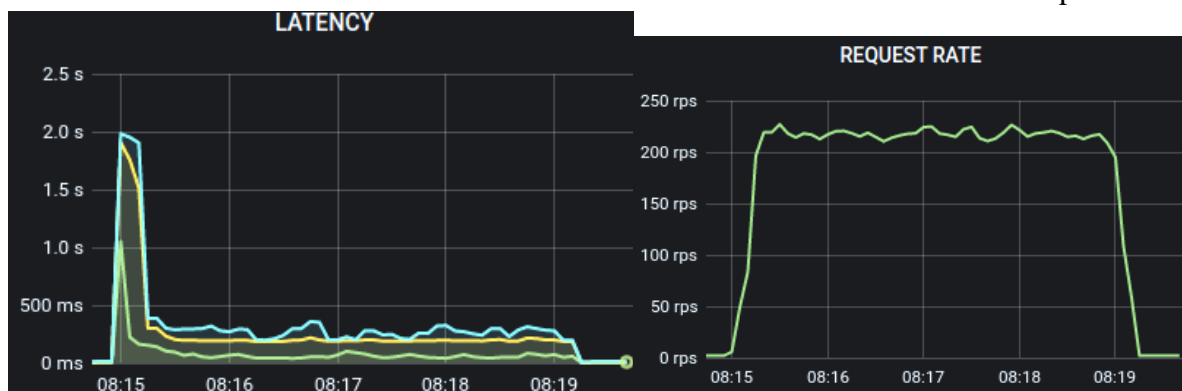
Max pods = 3



Max pods = 6

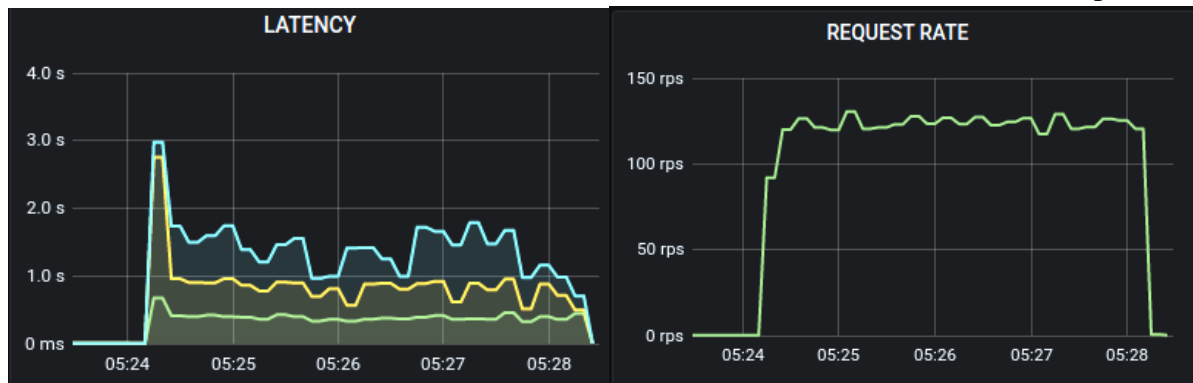


Max pods = 10

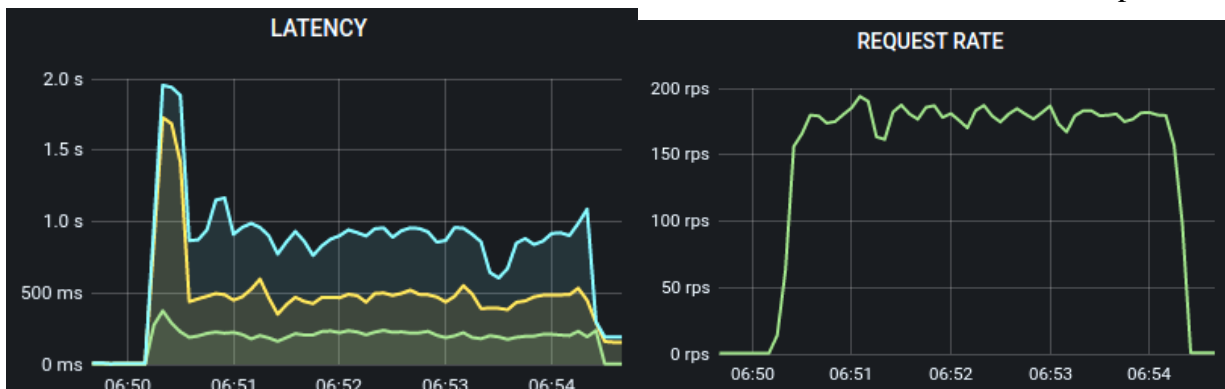


ب) تعداد کاربر ۴۰۰:

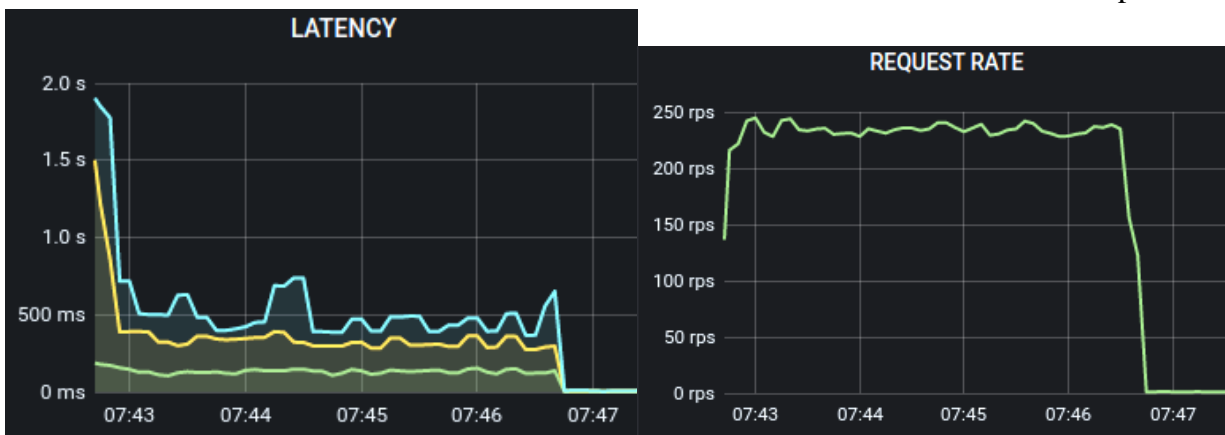
Max pods = 1



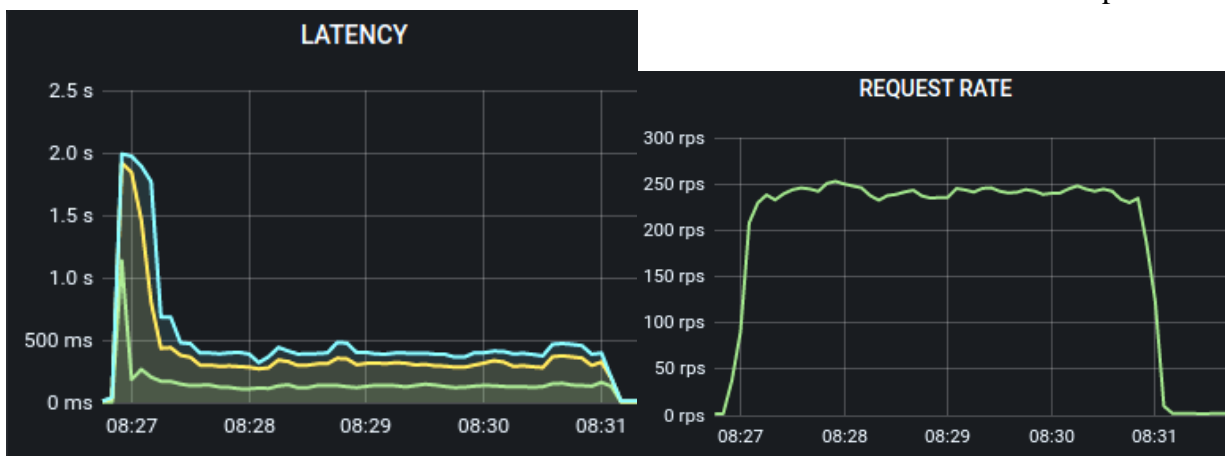
Max pods = 3



Max pods = 6

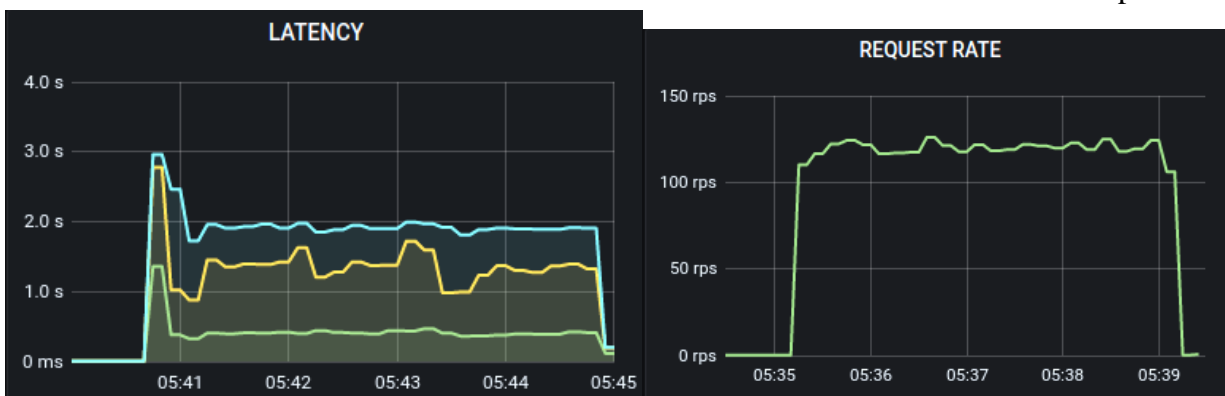


Max pods = 10



ج) تعداد کاربر ۵۰۰:

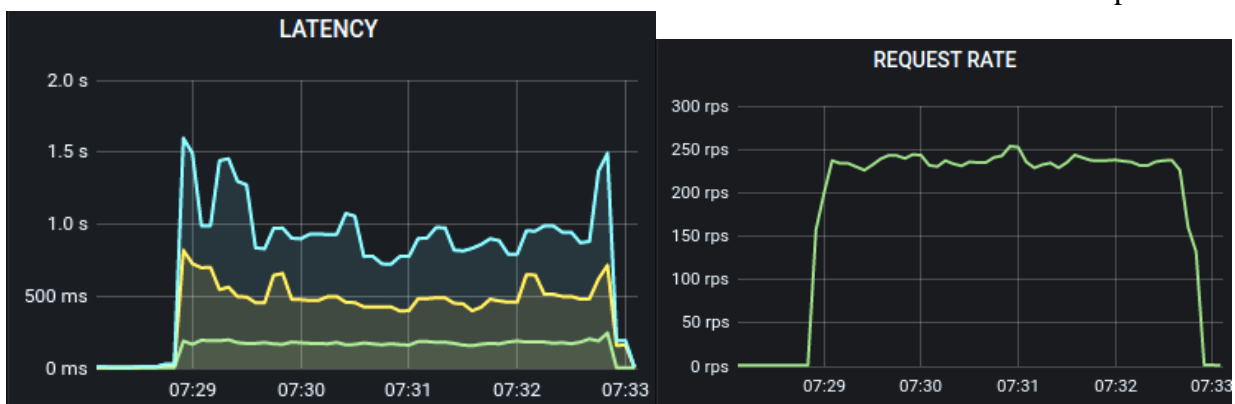
Max pods = 1



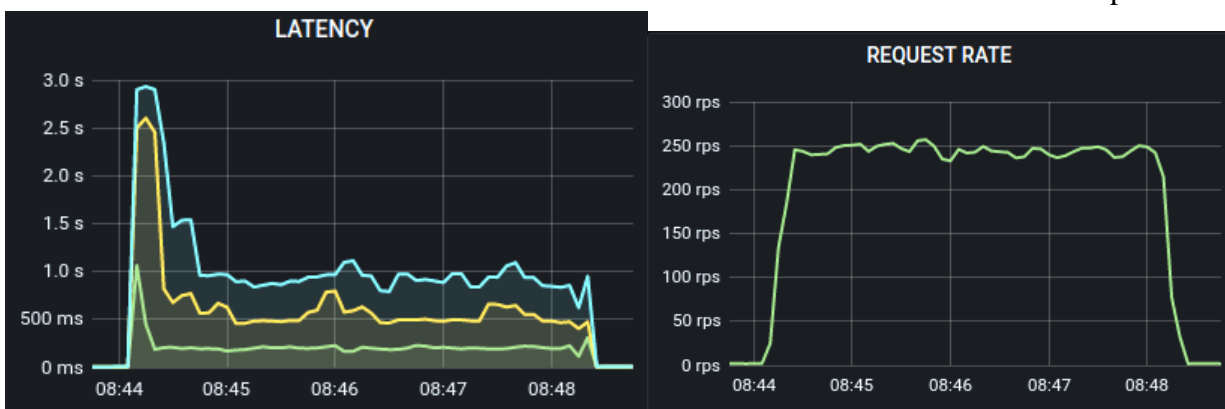
Max pods = 3



Max pods = 6

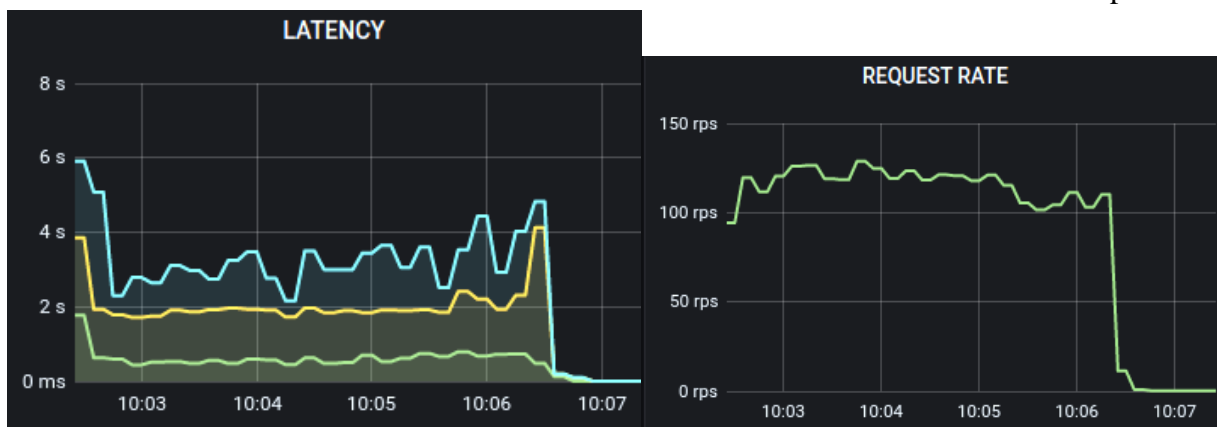


Max pods = 10

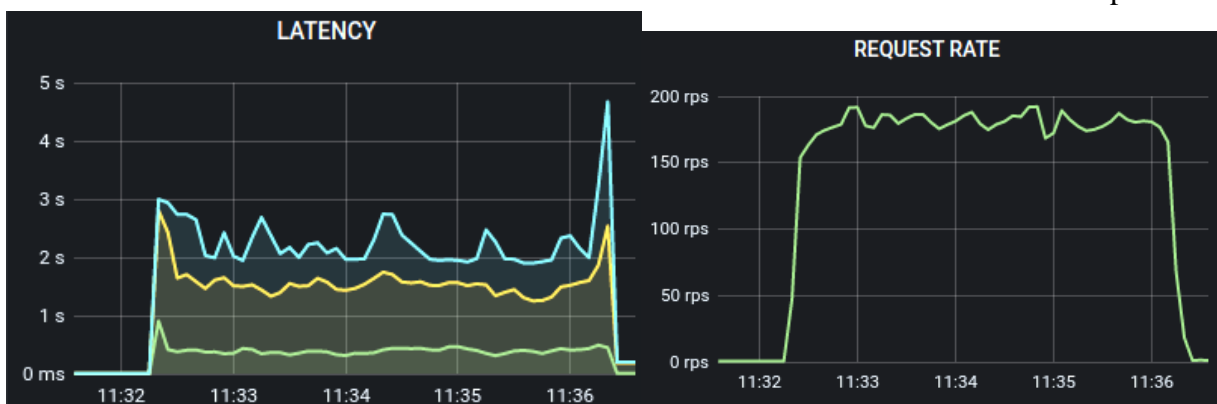


د) تعداد کاربر ۷۰۰:

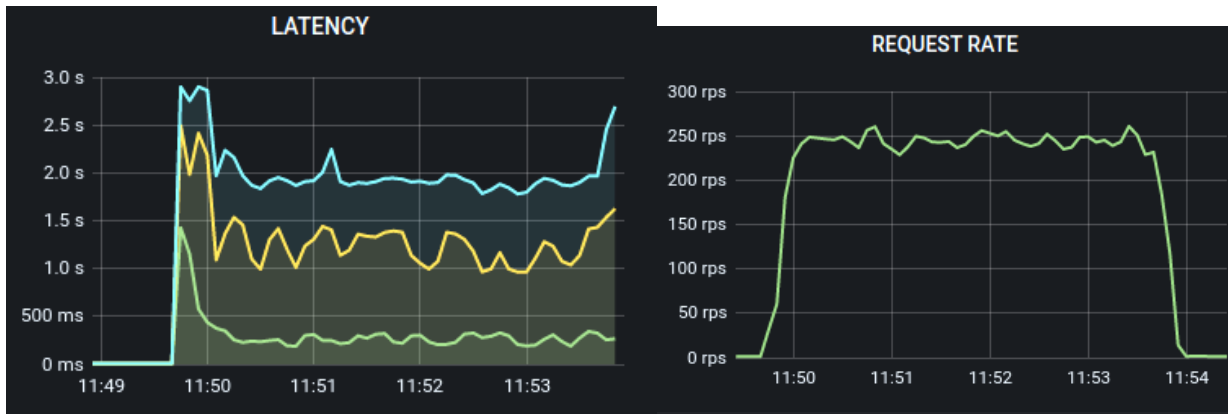
Max pods = 1



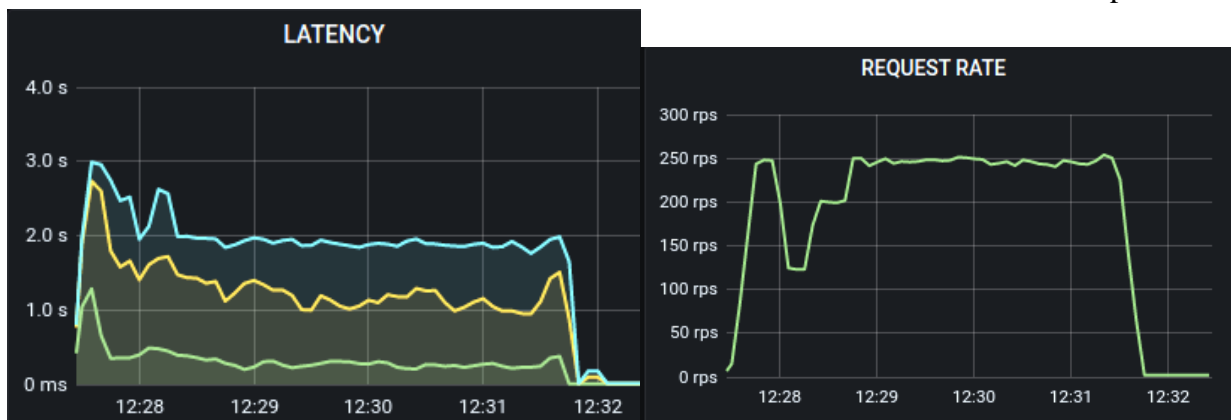
Max pods = 3



Max pods = 6



Max pods = 10



۲-۲-۶ تحلیل نتایج

بعد از آنکه نتایج را بدست آوردیم و گراف‌های 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 پادها، مایکروسرویس عملکرد بهتری دارد و زمان پاسخ کمتری نسبت به داشتن یک پاد نشان می‌دهد. ولی با توجه به اینکه منابع سخت افزاری محدود است این عملکرد تا یک جایی می‌تواند بهبود یابد و از یک حدی بیشتر دیگر افزایش پادها کمکی به بهبود زمان پاسخ نخواهد کرد و این به این منظور است که هر پاد از لحاظ نرم افزاری تعداد محدودی درخواست در ثانیه را می‌تواند جواب بدهد و هنگامی که بیش از حد توان خود بار دریافت می‌کند، خطا در

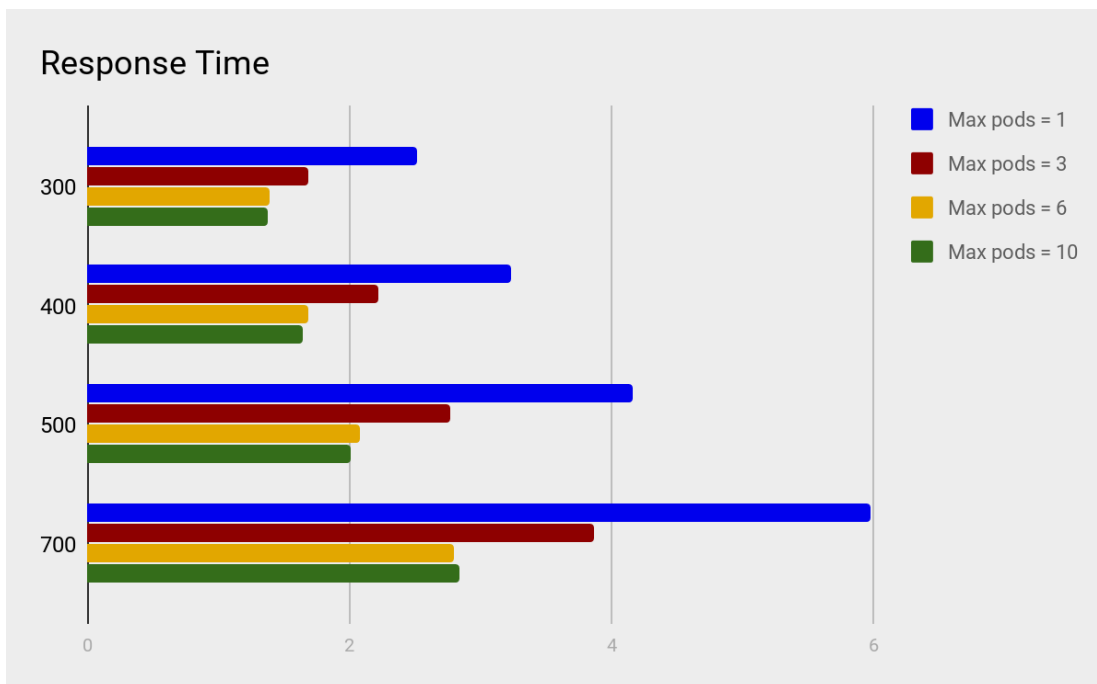
جواب هم بالا می‌رود پس با تقسیم بار بر روی پادها می‌توانیم این مشکل را حل کنیم و باعث شود که هر پاد تعداد کمتری درخواست را پاسخ دهد. در آن طرف قضیه منابع سخت افزاری محدود است و هر پادی که تولید می‌شود یک بخشی از حافظه و CPU را اشغال می‌کند و اگر این منابع کم بیاید، سیستم کند می‌شود و دیگر قادر نخواهد بود به سرعت به درخواست‌ها جواب بدهد. پس اگر پادها را از یک حدی بیشتر کنیم، درست است که بار خیلی کمتر بر روی هر پاد می‌افتد ولی به دلیل آنکه منابع سخت افزاری هم محدودتر می‌شود، پادها هم سرعتشان کمتر می‌شود. پس باید یک نقطه بهینه در سیستم خود پیدا کنیم که چه میزان پاد برای درخواست‌های مناسب است و برای تعداد درخواست‌هایی که در نظر گرفتیم، تعداد ۶ پاد عملکرد سیستم را بیشتر بهبود بخشیده و بیشتر از آن خیلی در نتایج فرقی ایجاد نمی‌کند.

همچنین وقتی تعداد پادها را افزایش می‌دهیم و زمان پاسخ مایکروسرویس کمتر می‌شود به تبع آن میزان تعداد درخواست بیشتری در ثانیه را می‌تواند جواب دهد که این نتایج در گراف‌ها و جداول قسمت نتایج آمده است. به علاوه جدولی برای مقایسه تعداد بارهای مختلف و تاثیر افزایش پادها بر روی معیارهای زمان پاسخ و میزان تعداد درخواست در ثانیه، در صفحه بعد آمده است. 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 پادها تا یک حدی، عملکرد نرم افزاری بهتر می‌شود و پادهای بیشتر می‌توانند پاسخگو باشند و تعداد بیشتری درخواست در ثانیه را جواب بدهند ولی بعد از یک حدی از تعداد پادها به دلیل محدودیت‌های سخت افزاری، مایکروسرویس نمی‌تواند تعداد بیشتری درخواست در ثانیه را جواب بدهد به دلیل آنکه دچار کمبود در منابع سخت افزاری می‌شویم و همین باعث می‌شود که تعداد بیشتر پادها به افزایش تعداد درخواست در ثانیه کمکی نکند.

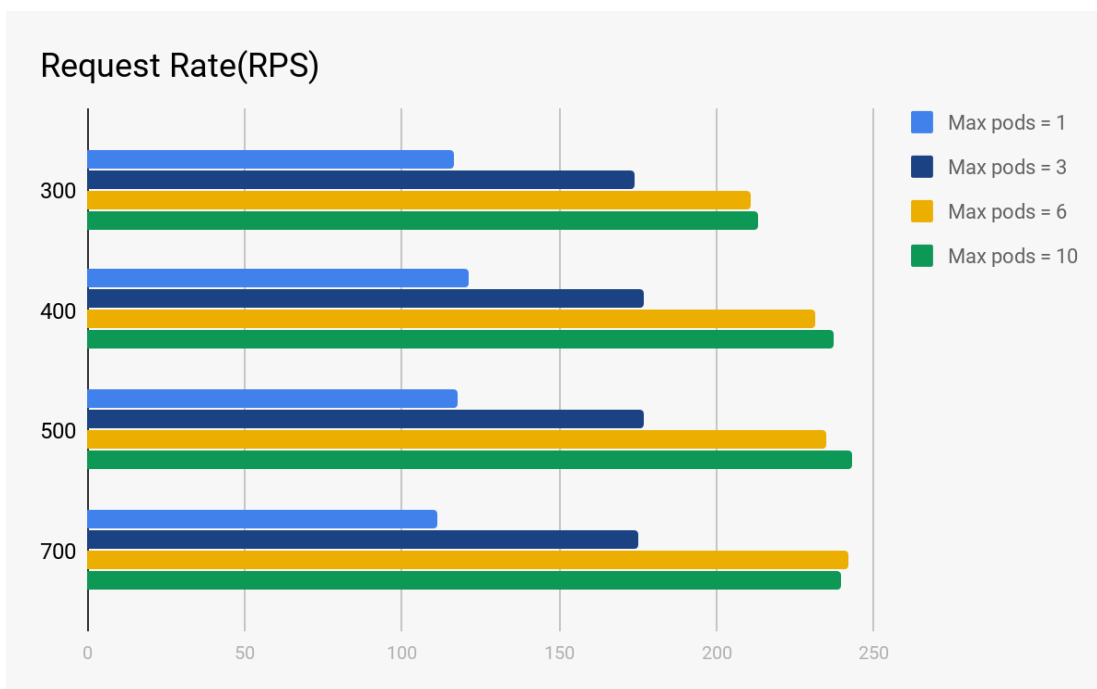
همان طور که از نتایج مشخص است، با افزایش پادها میزان دسترس‌پذیری سیستم هم افزایش پیدا کرده است. درست است که این افزایش برای بارهای پایین‌تر قابل توجه نیست ولی برای بار ۵۰۰ و ۷۰۰ این افزایش تا حدودی ملموس است و به دلیل آنکه بیشتر از ۷۰۰ سیستم دچار مشکل میشد تولیدکننده بار قادر به ادامه کار نبود و امکان افزایش بار بیش از این وجود نداشت. در حالت کلی وقتی بار بر روی پادها تقسیم می‌شود و مایکروسرویس سریع‌تر بتواند درخواست‌ها را پاسخ بدهد و تعداد بیشتری درخواست پاسخ داده شود، به تبع آن دسترس‌پذیری هم افزایش میابد.

قسمت آخر از نتایج که می‌تواند قابل توجه باشد این است که با افزایش پادها حداکثر زمان پاسخ هم کم شده است. زمان پاسخی که در نتیجه تولیدکننده بار آمده است متوسط همه زمان پاسخ‌ها است و بعضی از درخواست‌ها به دلیل آنکه بر روی مایکروسرویس بار زیادی افتاده است، طول میکشد پاسخ داده شود. ولی

هنگامی که تعداد پادها را افزایش می‌دهیم و بار بین پادها تقسیم می‌شود، مشاهده می‌کنیم که حداکثر زمان پاسخ هم کاهش می‌ابد و باعث بهبود عملکرد مایکروسرویس می‌شود.



شکل ۶-۱۴: مقایسه بارهای مختلف و ماکزیمم پادها برای زمان پاسخ



شکل ۶-۱۵: مقایسه بارهای مختلف و ماکزیمم پادها برای تعداد درخواست در ثانیه

۳-۶ جمع‌بندی

در این فصل نتایج مقیاس‌پذیری خودکار برای معیارهای محاسباتی و حافظه‌ای را مشاهده کردیم. همچنین نحوه عملکرد مقیاس‌پذیر کننده خودکار برای این معیارها هم نشان داده شد. در قسمت دوم مقیاس‌پذیری بر اساس معیارهای زمان پاسخ و تعداد درخواست در ثانیه انجام شد. به دلیل آنکه این دو معیار در قسمت دوم اهمیت بیشتری دارند و عملکرد مایکروسرویس‌ها را بهتر و کاربردی‌تر نشان می‌دهند، سناریوهای مختلفی برای مقیاس‌پذیری خودکار مایکروسرویس مد نظر در نظر گرفتیم و نتایج آن را به صورت گراف و جدول به نمایش گذاشتیم. همچنین این نتایج مورد تحلیل و بررسی قرار گرفت.

فصل هفتم: جمع‌بندی و پیشنهادها

۷-۱ جمع‌بندی

در این پروژه، سیستم مقیاس‌پذیری مبتنی بر بستر کوبرنتیز راه‌اندازی و مورد ارزیابی قرار گرفت. کلیه اجزا لازم در کنار کوبرنتیز بعنوان یک بستر مجازی‌سازی شامل اجزا مانیتورینگ معیارهای کیفیت میکروسرویس و مدیریت تعداد پادها راه‌اندازی شدند تا مقیاس‌پذیری بصورت خودکار صورت گیرد. خروجی این پروژه پیاده‌سازی یک سیستم برای دستیابی به مقیاس‌پذیری خودکار میکروسرویس‌ها با استفاده از معیارهای کیفیت میکروسرویس می‌باشد. برای این منظور از پلتفرم کوبرنتیز استفاده شد و قابلیت و امکانات این پلتفرم و اجزا جانبی مورد نیاز مورد بررسی قرار گرفت. مراحل مختلف راه‌اندازی این سیستم و اجزا مورد نیاز برای مانیتورینگ کیفیت سرویس و افزایش/کاهش تعداد پادهای میزبان میکروسرویس‌ها تشریح گردید و سازگاری اجزا مختلف این زیست‌بوم تست و عملکرد آن تحت شرایط مختلف بار کاری مورد ارزیابی قرار گرفته و نتایج حاصله مستندسازی گردیده است.

در این گزارش، ابتدا مفاهیم، اجزا و ساختار داکر و کوبرنتیز را بررسی کردیم. سپس با ایمج کردن میکروسرویس‌ها و راه‌اندازی محیط کوبرنتیز، توانستیم فایل پیکربندی آبجکت‌های مختلف را بنویسیم و کانتینرهای خود که همان میکروسرویس‌ها می‌شوند را مدیریت کنیم. در نهایت با استفاده از پلتفرم کوبرنتیز و داکر، توانستیم میکروسرویس‌ها را بر اساس معیارهای مختلف مقیاس‌پذیر کنیم. برای آنکه این مقیاس‌پذیری خودکار را تست کنیم، از تولیدکننده بار استفاده کردیم و میکروسرویس خود را زیر بارهای مختلف امتحان کردیم. در نتیجه این مقیاس‌پذیری خودکار، توانستیم دسترس‌پذیری و زمان پاسخ میکروسرویس‌ها را کمتر کنیم و به تعداد درخواست بیشتری در ثانیه به کاربران جواب بدهیم که بهبود این معیارها برای ما بسیار حایز اهمیت است. طی این گزارش، هر یک از مراحل پیاده‌سازی و راه‌اندازی برای دستیابی به مقیاس‌پذیری خودکار میکروسرویس‌ها را در فصل‌های جداگانه شرح دادیم.

۲-۷ پیشنهادها

الف) همان طور که قبلا هم ذکر کرده بودیم، از ۳ گره با منابع سخت افزاری محدود، کلاستر خود را راه‌اندازی کرده‌ایم. منابع سخت افزاری محدود، باعث می‌شود که نتوانیم بار زیادی را بر روی مایکروسرویس‌ها اعمال کنیم. در نتیجه می‌توانیم کلاستر خود را به جای آنکه به صورت محلی راه‌اندازی کنیم، بر روی ابر راه‌اندازی کنیم تا بتوانیم منابع سخت افزاری بیشتری در اختیار داشته باشیم و بار نزدیک به واقعیت را بروی مایکروسرویس‌ها اعمال کنیم. در این پروژه حداکثر بار ۷۰۰ کاربر بود که امروزه بار زیادی به حساب نمی‌آید، به همین دلیل افزایش دسترس‌پذیری در این پروژه برای بارهای ۳۰۰ و ۵۰۰ خیلی ملموس نیست ولی برای بار ۷۰۰ ملموس‌تر است. پس با بار بیشتر و با مقیاس‌پذیری خودکار مایکروسرویس‌ها، بهتر و ملموس‌تر می‌توانیم بهتر شدن دسترس‌پذیری را مشاهده کنیم.

ب) با استفاده از راه‌اندازی کلاستر خود در محیط ابر، می‌توانیم روش اول در فصل ۵ که cluster auto-scaler نام داشت را تست بکنیم. اگر گره‌های ثابت باشند، و تعداد پادها زیاد شود به طوری که دیگر منابع سخت افزاری لازم برای ایجاد پاد جدید نداشته باشیم، کوبرنتیز اجازه ساخته شدن پادهای جدید را نمی‌دهد. ولی اگر بتوانیم گره‌های جدید اضافه کنیم، به تبع آن پادهای بیشتری می‌توانیم در کلاستر خود بسازیم. و همچنین وقتی پادهای کمتر شد، گره اضافی را خاموش می‌کنیم.

ج) می‌توانیم از بقیه ارکستریتورها مانند Cloudify و Openshift برای مقیاس‌پذیری استفاده کنیم و عملکرد مقیاس‌پذیر کننده خودکار آنها را با ارکستریتور کوبرنتیز مقایسه کنیم.

منابع و مراجع

- [1] edx. (2020, March 10). *Introduction to Kubernetes* [Online]. Available: <https://courses.edx.org/courses/course-v1:LinuxFoundationX+LFS158x+2T2019/course/>
- [2] S. Grider. (2020, February 24). *Docker and Kubernetes: The Complete Guide* [Online]. Available: <https://www.udemy.com/course/docker-and-kubernetes-the-complete-guide/>
- [3] B. Lewis. (2020, July 2). *5 important things you need to know about Docker* [Online]. Available: <https://www.besttechie.com/5-important-things-you-need-to-know-about-docker/>
- [4] D. Weibel. (2020, May 20). How to autoscale apps on Kubernetes with custom metrics [Online]. Available: <https://learnk8s.io/autoscaling-apps-kubernetes>
- [5] Z. Antolovic. (2020, June 17). *Web App Performance Testing With Siege*[Online]. Available: <https://www.sitepoint.com/web-app-performance-testing-siege-plan-test-learn/>
- [6] Prometheus. (2020, June 27). *HISTOGRAMS AND SUMMARIES* [Online]. Available: <https://prometheus.io/docs/practices/histograms/>
- [7] Prometheus. (2020, June 27). *Functions* [Online]. Available: <https://prometheus.io/docs/prometheus/latest/querying/functions/>
- [8] H. Haidar. (2020, June 24). *Kubernetes Autoscaling in Production: Best Practices for Cluster Autoscaler, HPA and VPA* [Online]. Available: <https://www.replex.io/blog/kubernetes-in-production-best-practices-for-cluster-autoscaler-hpa-and-vpa>
- [9] R. Chesterwood. (2020, May 10). *Kubernetes Hands-on - Deploy Microservices to the AWS Cloud* [Online]. Available: <https://www.udemy.com/course/kubernetes-microservices/>
- [10] edx. (2020, January 15). *Introduction to Cloud Infrastructure Technologies* [Online]. Available: <https://courses.edx.org/courses/course-v1:LinuxFoundationX+LFS151.x+2T2020/course/>

- [11] linkerd. (2020, June 3). *Getting Started* [Online]. Available: <https://linkerd.io/2/getting-started/>
- [12] linkerd. (2020, June 7). *The Service Mesh: What Every Software Engineer Needs to Know About the World's Most Over-Hyped Technology* [Online]. Available: <https://servicemesh.io/>
- [13] Just me and Opensource. (2020, May 18). *Prometheus monitoring for Kubernetes Cluster and Grafana visualization* [Online]. Available: <https://www.youtube.com/watch?v=CmPdyvgmw-A&t=1054s>
- [14] Just me and Opensource. (2020, May 4). *Using Horizontal Pod Autoscaler in Kubernetes* [Online]. Available: https://www.youtube.com/watch?v=uxuyPru3_Lc&t=1325s
- [15] Just me and Opensource. (2020, May 4). *Pod auto-scaling based on memory utilization* [Online]. Available: <https://www.youtube.com/watch?v=KS5MzK4EDg8&t=770s>
- [16] Just me and Opensource. (2020, May 19). *Dynamically provision NFS persistent volumes in Kubernetes* [Online]. Available: <https://www.youtube.com/watch?v=AavnQzWDTEk>
- [17] Google Cloud. (2020, June 21). *Configuring vertical Pod autoscaling* [Online]. Available: <https://cloud.google.com/kubernetes-engine/docs/how-to/vertical-pod-autoscaling>
- [18] T. Rampelberg. (2020, June 7). *Scale Your Service on What Matters: Autoscaling on Latency* [Online]. Available: <https://www.youtube.com/watch?v=gSiGFH4ZnS8>
- [19] Jace. (2020, May 11). *Horizontal Pod Autoscale with Custom Prometheus Metrics* [Online]. Available: <https://dev.to/mjace/horizontal-pod-autoscale-with-custom-prometheus-metrics-5gem>
- [20] Kubernetes. (2020, April 10). *Horizontal Pod Autoscaler Walkthrough* [Online]. Available: <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/>
- [21] W. Hetherington. (2020, May 30). *Load Testing your Site with Siege* [Online]. Available: <https://drupalize.me/blog/201507/load-testing-your-site-siege>