دانشگاه شهید بهشتی

دانشکده مهندسی و علوم کامپیوتر

کاهش شروع‌های سرد در پلتفرم‌های بدون سرور

گزارش سمینار کارشناسی ارشد مهندسی کامپیوتر
گرایش نرم‌افزار

نگارش

امیرمحمد کرم‌زاده

استاد راهنما

دکتر علیرضا شاملی

خرداد ۱۴۰۰

چکیده

رایانش ابری، یکی از مباحث داغ حوزه مهندسی نرم‌افزار است که به دلیل فواید بسیار آن، مورد توجه کاربران قرار گرفته است. یکی از مدل‌های اجرایی در رایانش ابری، مدل اجرایی رایانش بدون سرور است که با هدف راحتی کاربران و توسعه‌دهندگان برنامه‌ها و ابزارها برای توسعه‌ی خدمات ابری، برنامه‌های کاربردی در وب یا برنامه‌های کاربردی موبایل طراحی شده است. اگرچه رایانش بدون سرور با ساده‌سازی فرآیند تعامل و مدیریت، بسیار موفق عمل کرده است؛ اما، این مدل اشکلاتی دارد که مانع تمایل کاربرا به سمت آن شده است. رایج‌ترین معضل کاربران با پلتفرمّ‌های بدون سرور تاخیر ناشی از پاسخ برنامه‌هاست که به تاخیر شروع سرد معروف است. این تاخیر می‌تواند در مواردی زمان پاسخ را تا چندین برابر افزایش دهد و معمولا چندین برابر زمان اجرای برنامه است. بنابراین لازم است که با این مشکل به صورت جدی مقابله شود. ما متوجه شدیم عمده تاخیر ناشی از شروع سرد به علت آماده سازی کانتینرهای سرویس فراخوانی شده و بارگذاری کتابخانه‌های برنامه‌نوشته شده در داخل کانتینر هستند. البته بعد از این کارها باید منابع کافی RAM) و (CPU برای اختصاص به کانتینر داشته باشیم. برای مقابله با شروع سرد، دو راهکار کلی داریم. راهکار اول این است که مانع از اتفاق افتادن شروع سرد شویم و راهکار دوم بهبود زمان شروع سرد است. این دو هر کدام مزیت و عیب خود را دارند که در x در مورد آن بحث‌هایی خواهد شد. در x به این نتیجه رسیدیم که احتمالا بتوانیم با ترکیب مدل‌هایی بتوانیم مانع از شروع‌های سرد بشویم. به طور خلاصه پژوهش ما درباره‌ی حداقل سازی شروع سرد و راهکارهای کمینه سازی آن است.

واژگان کلیدی: رایانش ابری، رایانش بدون سرور، شروع‌های سرد، function-as-a-service، FaaS

فهرست مطالب

فهرست شکل‌ها

فهرست جداول

فصل ۱

مقدمه

۱.۱ صورت مسئله

چه روش‌هایی برای کاهش تعداد شروع‌های سرد در پلتفرم‌های بدون‌سرور [۱] با حداقل سربار [۲] و زمان اجرایی وجود دارند؟

۲.۱ انگیزه‌ی تحقیق

رایانش بدون سرور[۳] یکی از مسائل داغ و محبوب این‌روزهای دنیای مهندسی نرم‌افزار و رایانش ابری است. رایانش بدون سرور حوزه‌ی جدیدی را در توسعه‌ی محصول و استقرار[۴] اپلیکیشن‌ها باز کرده است. یکی از دلایل محبوبیت استفاده از پلتفرم‌های بدون‌سرور و تمایل توسعه‌دهندگان برای مهاجرت به سمت آن، استفاده بیش از پیش از معماری میکروسرویس و نانوسرویس در توسعه‌ی محصولات و حرکت معماران و مهندسین نرم‌افزار در تولید و مهاجرت برنامه‌های کاربردی با این معماری‌ها است.

از دید توسعه دهنده، رایانش ابری با حذف دخالت مستقیم کاربران انتهایی در مدیریت زیرساخت ازجمله lrauto-scaling یا lrload-balancing موجب بهبود سرعت توسعه محصول و تمرکز کاربران برروی منطق[۵] برنامه است. همچنین، برای علاوه‌بر آسانی استفاده و پنهان‌سازی پیچیدگی مدیریت سرور از کاربر، به علت اینکه ارائه‌دهندگان خدمات ابری در نقاط مختلف جهان حضور دارند و همچنین کانفیگ بهینه CDNها؛ ارتباطات بین سرورها و کاربران با حداقل تاخیر[۶] صورت می‌گیرد.

به طور کلی، یک پلتفرم بدون‌سرور را هر پلتفرم محاسباتی تعریف کرد که در آن مدیریت مستقیم سرور از کاربران مخفی شده و برنامه‌های کاربردی به صورت اتوماتیک در آن مقیاس‌پذیر می‌شوند و تنها هنگامی که در حال استفاده از پلتفرم هستیم، هزینه آن را پرداخت می‌کنیم. [۱]

یکی از قابلیت‌هایی که در رایانش بدون سرور باعث محبوبیت آن شده‌است، قابلیت Scale-to-Zero است. این بدان معنی است که هنگامی که از یک کانتینر استفاده‌ای نداریم، منابع آن گرفته می‌شوند و کانتینر اصطلاحا

[1] serverless
[2] Overhead
[3] Serverless Computing
[4] Deployment
[5] logic
[6] Latency

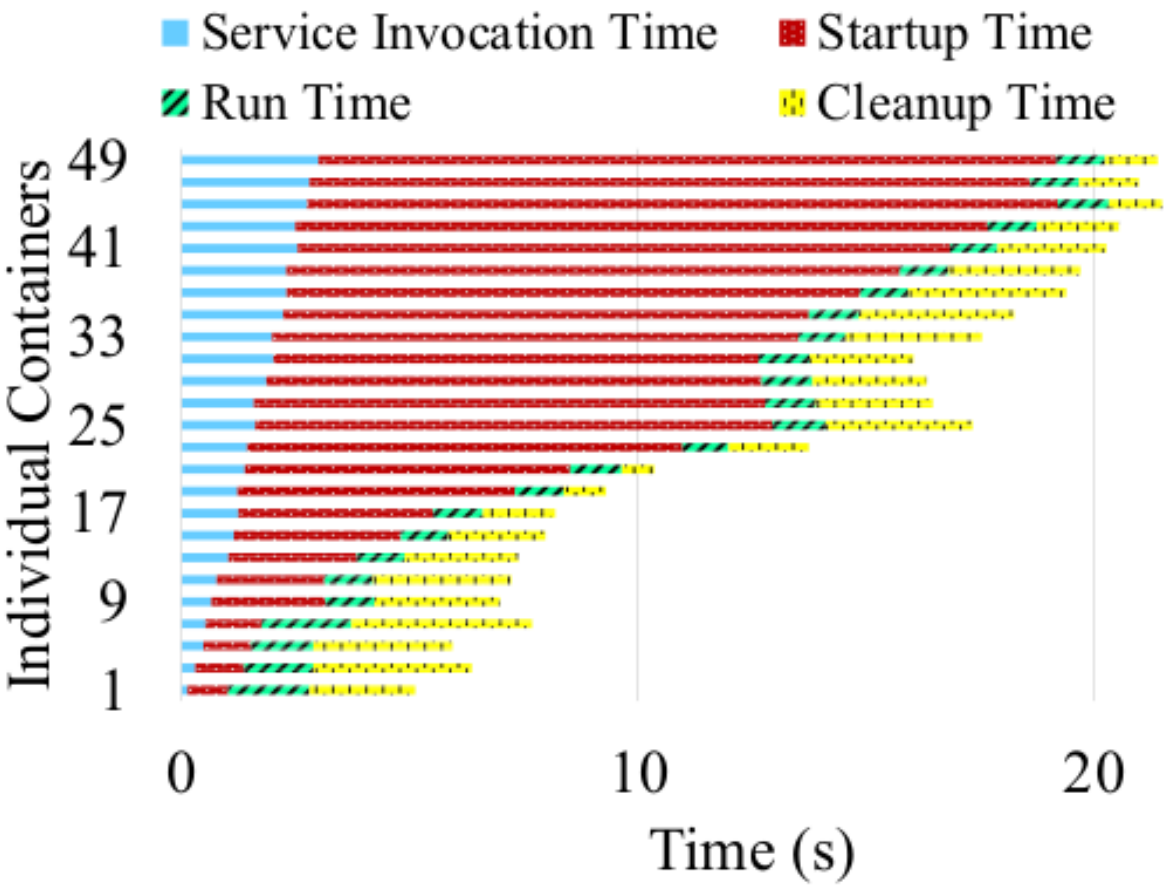
Zero-Scaled می‌شود. این خود موجب قابلیت پرداخت تنها در حین مصرف ما از تابع می‌شود. اما مشکل اصلی زمانی است که درخواست جدیدی برای کانینر Zero-Scale شده می‌رسد؛ در این حالت باید درخواست منتظر مانده تا سلسله‌ای از آماده‌سازی ها انجام شوند تا کانتینر مربوطه مجددا اجرا شود. این خود باعث تاخیری مضاف برای پاسخ‌دهی به درخواست را موجب می‌شود که به این تاخیر مشکل شروع‌سرد [1] گفته می‌شود. در واقع می‌توان گفت تاخیر شروع سرد ناشی از تلاش ما در تعادل بین تاخیر در پاسخگویی به درخواست‌ها و هزینه (هزینه‌های استفاده از رم و سی‌پی‌یو و ...) است.

متاسفانه در سالیان اخیر و در عین داغ‌بودن مبحث و نیاز بازار به حل این مشکل، این مشکل چندان در محیط‌های آکادمیک مورد بررسی قرار نگرفته است. البته در نگاه کلی‌تر، مشکلات و مسائل بار مربوط به رایانش بدون‌سرور، اکثرا در محیط‌های آکادمیکی مثل دانشگاه‌ها با کم محلی روبرو شده‌اند. در این میان، پلتفرم‌های متن‌باز [2] که دارای جوامع بسیار گسترده‌ای نیز هستند، به خاطر این سری مسائل باز که شرکت‌های تجاری در حال صرف هزینه‌های هنگفتی برای حل و فصل مسکلات مربوط به آن هستند، به شدت از رقابت عقب مانده‌اند. انگیزه ما برای انجام این پژوهش در این است که اولا بتوانیم به راهکار مناسب‌تری برای حل مشکل مربوط به شروع سرد در پلتفرم‌های بدون سرور برسیم و ثانیا بتوانیم با مشارکت در بهبود یکی از پلتفرم‌های آزاد در توسعه این پلتفرم‌ها تاثیر کوچکی داشته باشیم.

۳.۱ مثال مرتبط

در مقاله [۲] آقای لین و همکارش توانستند تا با اسفاده از استخر گرم و نگه‌داری کانتینر توابعی که محبوبیت استفاده دارند، مدت زمان پاسخ را در حدود ٪۸۵ کاهش دهند. این بهبود در پلتفرم knative اجرا شد و ایده‌ی مقالات دیگری نیز بوده است.

[1] Cold Start
[2] Open Source



شکل ۱.۱: اهمیت شروع‌سرد

۴.۱ اهمیت موضوع

اگرچه پلتفرم‌های بدون‌سرور از نظر هزینه، scaling، راحتی استفاده و گستردگی پوشش جغرافیایی برای ما بهینه هستند؛ اما مشکل شروع سرد مشکلی نیست که بتوان به سادگی از آن گذر کرد. تصویر ۱.۱ که از [۳] گرفته شده است، نشان می‌دهد که بیش از ٪۸۰ زمان اجرای کامل یک کانتینر در پلتفرم‌های بدون سرور به آماده‌سازی آن یا شروع سرد اولیه[1] مربوط می‌شود.

ستون قرمز رنگ زمان آماده سازی کانتینر را نمایش می‌دهد. این زمان همان زمان شروع سرد است که دلایل مختلفی از جمله آماده‌سازی کانتینر یا حضور در صف انتظار برای تخصیص منابع باشد. بنابراین، با به‌کارگیری یک استراتژی مناسب می‌توان این زمان را به حداقل رسانید.

متاسفانه تاخیر شروع سرد باعث شده تا توسعه‌دهندگان اقبال کمتری به استفاده از پلتفرم‌های بدون سرور

[1] First Cold Start

داشته باشند. به گونه‌ای که از بین ۱۰۰۰ برنامه‌ی کاربردی بزرگ در پلتفرم Microsoft Azure، تنها یک مورد مربوط به یک برنامه تجاری باشد [۴]. این موضوع نشان می‌دهد که علی‌رغم پتانسیل بالای رایانش بدون سرور، وجود مشکلات جدی ازجمله شروع سرد، باعث امتناع توسعه‌دهندگان از مهاجرت به پلتفرم‌های بدون سرور باشد.

۵.۱ نتیجه‌های مهم تحقیق

روش‌های مبتنی بر پیشگیری از شروع سرد، اگرچه در سناریوی موفق خود تعداد رخداد شروع سرد را کاهش می‌دهند، ولی در اغلب سناریوهای ناموفق - که تعداد آن‌ها کم هم نیست -، دچار تاخیر طولانی مدت شروع سرد می‌شوند. مشکل دیگر این روش‌ها این است که سربار زمانی محاسبات برای پیش‌بینی شروع سرد بالاست. در حالت بهینه باید دنبال کردن[1] عملکرد برنامه با ساختمان داده بهینه و حداقل سربار اجرایی باشد. محاسبه فراخوانی بعدی نیز به سرعت انجام بگیرد زیرا با محاسبات طولانی عملا از الگوی برنامه عقب می‌افتیم و قادر به پیش‌بینی آن نخواهیم بود.

می‌توان مسئله شروع سرد را به پدیده‌های مختلف نیز ربط داد و بر اساس راه‌حلی که برای آن مسئله ارائه شده است، اقدام به پیاده‌سازی راهکار مشابه برای مسئله کنیم. مثلا مقاله [۱۸]، از تشابه مسئله میزان اصابت کش‌های TTL مدل مدنظر خود را ایده گرفته و اقدام به حل شروع سرد کرده است.

روش‌های کاهش زمان تاخیر نیز چالش‌های خود را دارند. یکی از این چالش‌های محدودیت‌ها در ساختمان داده ارائه شده است. مثلا اگر بخواهیم از Pause Containerها برای حل مسئله استفاده کنیم؛ به دلیل محدودیت‌های هسته لینوکس در ساخت شبکه‌ها، بیشتر از ۱۰۲۴ PC نخواهیم داشت.

ساختار این گزارش به این ترتیب خواهد بود:

درادبیات موضوع مروری بر واژگان، مفاهیم تخصصی و هر آن‌چه که در ادامه به آن نیاز پیدا خواهیم کرد، خواهیم داشت. سپس در فصل کارهای مرتبط به بیان مشروح تحقیقات انجام شده خواهیم پرداخت و در نتیجه‌گیری و کارهای آینده خلاصه‌ای از مطالعات انجام شده و مسائل باز خواهیم پرداخت.

[1] Tracking

فصل ۲

مروری بر ادبیات

در این بخش سعی داریم تا با مروری بر اصطلاحات و ابزارهای مورد استفاده در پژوهش‌های بررسی شده، با پیش‌نیازهای مبحث موردنظر آشنا شویم.

۱.۲ رایانش بدون سرور

رایانش بدون سرور[1] در سال ۲۰۱۴ توسط شرکت آمازون برای اولین بار معرفی شد. تا قبل از این رایانش بدون سرور یک مفهوم انتزاعی [2] در شبکه بود که شرکت آمازون با ارائه پلتفرم AWS Lambda Functions [؟] به معرفی آن پرداخت. سپس در سال ۲۰۱۶ سایر ارائه دهندگان خدمات ابری نیز به ارائه پلتفرم‌های بدون سرور خود پرداختند. در این سال به ترتیب شرکت‌های گوگل پلتفرم google cloud functions یا به اختصار GCP، شرکت مایکروسافت پلتفرم Microsoft Azure functions و شرکت IBM به معرفی IBM OpenWhisk پرداختند. البته باید توجه داشت که مفهوم رایانش بدون سرور به طور کامل توسط ارائه دهندگاه خدمات ابری پیاده‌سازی نشده است و جای کار بسیاری دارد (با مطالعه این گزارش به مرور متوجه نواقص موجود خواهید شد).

در رایانش بدون سرور ما از نقطه قوت ماشین‌های مجازی که ایزولاسیون برنامه‌های مختلف از همدیگر بود استفاده کرده‌ایم. منتها این مورد را با مفهوم کانتینر ها پیاده‌کرده ایم. در ادامه راجع به کانتینرها نیز بحث خواهیم کرد.

۱.۱.۲ تعریف رایانش بدون سرور

رایانش بدون سرور مبحثی از رایانش ابری است که در آن بحث مدیریت حافظه یا Storage، مدیریت زیرساخت و بحث‌های networking با انتزاع بالایی به مصرف کاربر می‌رسد. به عبارت دیگر، تمامی مدیریت ای بخش‌ها بر عهده ارائه دهندگان است و ما اصلا با این بحث ها سروکاری نداریم. در واقع، هدف اصلی رایانش بدون سرور هم این است که این پیچیدگی‌ها را از کاربر بگیرد.

به طور کلی، یک پلتفرم بدون‌سرور را هر پلتفرم محاسباتی تعریف کرد که در آن مدیریت مستقیم سرور از کاربران مخفی شده و برنامه‌های کاربردی به صورت اتوماتیک در آن مقیاس‌پذیر می‌شوند و تنها هنگامی که در

[1] Serverless Computing
[2] abstract

حال استفاده از پلتفرم هستیم، هزینه آن را پرداخت می‌کنیم. [۱]

بسیاری از افراد، serverless و faas را معادل یک‌دیگر می‌دانند درحالی‌که اصلا این‌گونه نیست. در ادامه راجع به این بحث به طور مفصلی بحث خواهیم کرد اما باید بدانیم که این دو مقوله کاملا جدا از همدگیر هستند و مجددا تاکید می‌کنیم که رایانش بدون سرور یک مدل اجرایی در رایانش ابری است.

ازطرفی رایانش بدون سرور را باید نقطه مقابل رایانش سرور آگاهانه [۱] دانست که در آن از اطلاعات سرور در اختیار گرفته کاملا اگاهیم، کاملا بر مدیریت آن اشراف داریم و هرگونه تغییر از جمله متعادل‌سازی بارها، auto-scaling و ... باید توسط کاربر انجام شود.

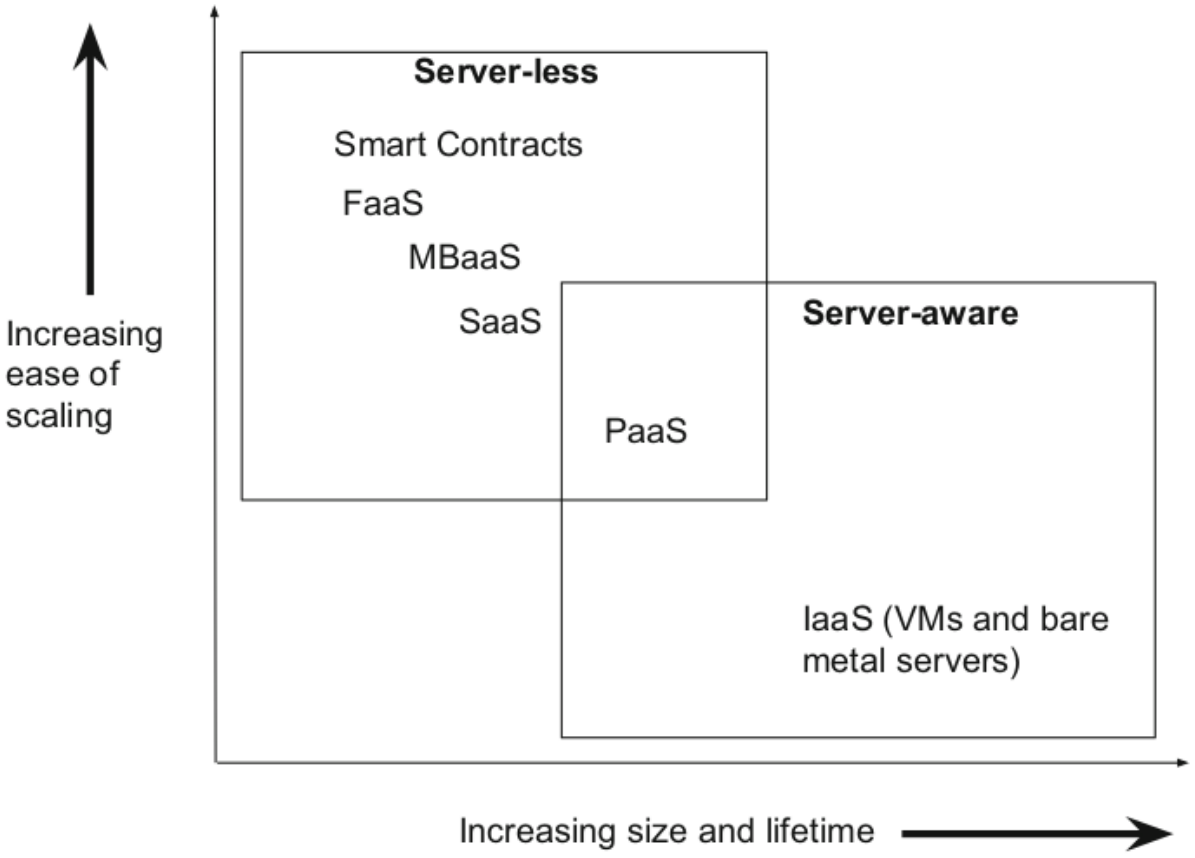
یک مثال از پیاده‌سازی رایانش سرور آگاهانه را در زیرساخت به عنوان سرویس [۲] یا به اختصار IaaS است. در نقطه مقابل در رایانش بدون سرور هیچ کنترلی بر روی سرور نداریم، تنها می‌توانیم یک برنامه را بر روی سرور اجرا کنیم یا اجرای آن را به حالت تعلیق درآورده یا آن را از روی سرور حذف کنیم که هیچ‌کدما از این موارد نیز به صورت مستقیم انجام نمی‌گیرد؛ بلکه رابط گرافیکی و API وجود دارد که از طریق آن‌ها این تغییرات را اعمال می‌کنیم. بنابراین در رایانش بدون سرور، عملا هیچ راهی برای مدیریت مستقیم سرور و زیرساخت نداریم.

شکل ۱.۲ مرزهای بین رایانش بدون‌سرور و رایانش سرور آگاهانه را نمایش می‌دهد.

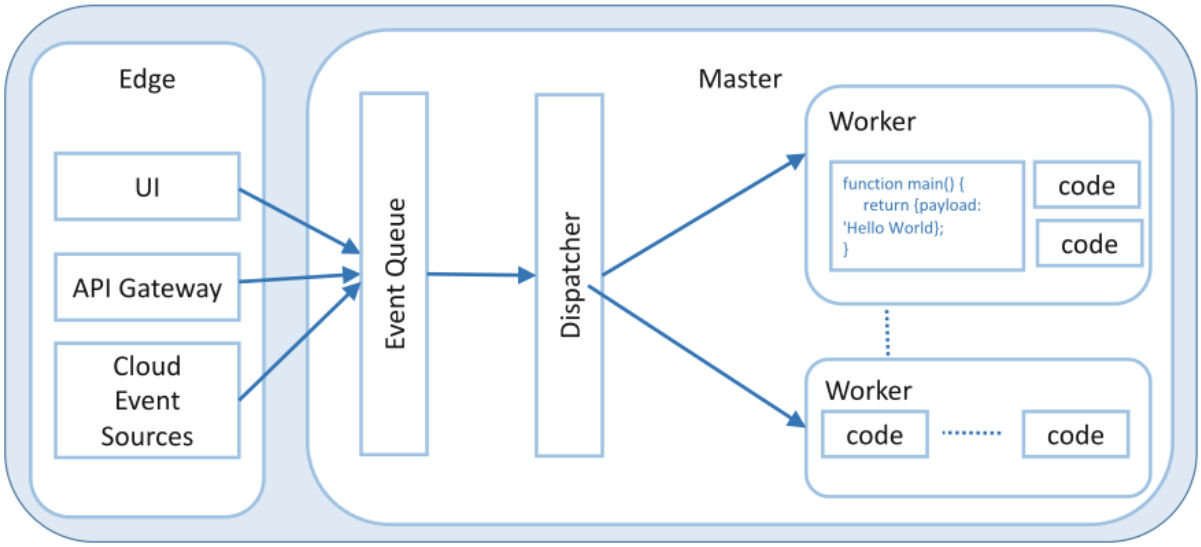
البته باید به این نکته توجه‌داشت که امروزه مرزهای بین رایانش سرور آگاهانه با رایانش بدون سرور در حال کمرنگ شدن و بعضا از بین رفتن است و این تقسیم بندی ابدا قاطعیت ندارد.همچنین تفکیک برخی موارد مانند Platform-as-a-Serice یا به اختصار PaaS به راحتی انجام نمی‌گیرد بلکه این نوع رایانش می‌تواند از نوع باسرور یا بدون سرور باشد. در این شکل هرچه به سمت محور افقی حرکت می‌کنیم دانه‌بندی و طول‌عمر افزایش پیدا می‌کند و هرچه به سمت بالاتر می‌رویم، scalingz راحت‌تر انجام می‌گیرد.

از مزایای رایانش بدون سرور همچنین می‌توان به پشتیبانی و توسعه راحت‌تر اپلیکیشن‌ها با معماری میکروسرویس و نانوسرویس هم اشاره کرد. البته معماری نانوسرویس مبحث جدیدتری است و جای پژوهش‌های بیشتری دارد.

[1] Server Aware
[2] Infrastructure as a service



شکل ۱.۲: مرزهای رایانش بدون سرور و رایانش سرورآگاهانه



شکل ۲.۲: معماری کلی یک پلتفرم بدون سرور

[۱]

۲.۱.۲ معماری

واژه serverless ممکن است این تفکر را به ذهن مبتدر سازد که اصلا در این نوع مدل رایانشی سروری نداریم؛ در حالی‌که این امر بسیار اشتباه است. در رایانش بدون سرور اگرچه سروری برای مدیریت به کاربر اختصاص داده نمی‌شود اما این موضوع بدان معنا نیست که اصلا سروری در کار نیست. در واقع مانند تمامی مدل‌های رایانشی در این‌جا هم سرور داریم، ولی تمامی تصمیمات مثل توزیع بار، تعداد اپ‌های روی سرور، انتخاب سرورها برای اجرای اپلیکیشن، کانفیگ CI/CD و ... بر عهده ارائه دهنده خدمات ابری است. اما برای درک نحوه کار یک پلتفرم بدون سرور، بهتر است با معماری آن آشنا باشیم. شکل ۲.۲ معماری یک پلتفرم خدمات ابری را نشان می‌دهد.

همانگونه که در تصویر مشخص است دوبخش کلی داریم، بخش لبه[۲] و بخش رییس [۳]، بخش لبه شامل رابط گرافیکی کاربر[۴]، Gateway API و Cloud Event Source می‌شود که برای تعامل با سرور رییس[۵] مناسب هستند. از طرف دیگر، در سرور رییس، درخواست‌ها ابتدا به صف رخدادها[۶] رسیده. صف رخدادها مسئول مدیریت رخداد

[2] edge
[3] master
[4] User Interface
[5] master server
[6] Event Queue

و نظم‌دهی به آن‌ها است. هر داده در صف رخ‌داد نوبت‌دهی می‌شود و سپس به بخش توزیع‌کننده[۱] می‌رود. توزیع‌کنند آپلیکیشن را برای دیپلوی، یا درخواست را برای سرویس‌دهی به یک نود کارگر[۲] هدایت می‌کند نود کارگر نیز با ارسال کدهای پاسخ[۳] با سرور رییس در ارتباط است. [۵]

البته بهتر است بدانیم که امروزه ادبیات رییس-کارگر[۴] برای نامیدن این معماری منسوخ شده و به جای آن از ادبیات رییس-گره[۵] استفاده می‌کنند.

۳.۱.۲ ویژگی‌های پلتفرم‌های بدون سرور

امروزه پلتفرم‌های بدون سرور بسیاری وجود دارند که روزانه بر تعداد آن‌ها افزوده می‌شود. اما باید دنبال ویژگی‌هایی برای آن‌ها باشیم که براساس آن‌ها بتوان این پلتفرم‌ها را تفکیک کرد و دست به مقایسه‌ی آن‌ها زد.

شاخص‌هایی که در این قسمت بررسی می‌کنیم شاخص‌های کمی و کیفی برای مقایسه‌ی بین این پلتفرم‌ها است.

۱. هزینه:

به طور معمول در رایانش بدون سرور، از مدل پرداخت پرداخت-به-ازای-استفاده[۶] استفاده می‌کنیم. یک ویژگی اساسی در تمایز بین ارائه‌دهندگان مختلف خدمات ابری، تفاوت آن‌ها در پشتیبانی از ویژگی scale-to-zero در این مدل محاسباتی[۷] است. این مورد باعث تفاوت معنی‌داری از هزینه‌ها در استفاده از پلتفرم‌های مختلف می‌شود. برخی از پلتفرم‌ها هم متن‌باز[۸] هستند که در این صورت، می‌توان به آسانی آن‌ها را بر روی ماشین مجازی یا سرور شخصی خود پیاده سازی کرد و برای استفاده از خدمات آن متحمل هیچ‌گونه هزینه‌ای نشد.

۲. کارایی و محدودیت‌ها[۹]

[1] dispatcher
[2] worker node
[3] Response Code
[4] Master-worker
[5] Master-Node
[6] pay-as-you-go
[7] Computational Model
[8] Open Source
[9] Performance and Limits

ارائه دهندگان مختلف، محدودیت‌های مختلفی را هم بر روی پلتفرم‌های مختلف خودشان اعمال می‌کنند. این محدودیت‌ها می‌توانند تعداد همزمان درخواست ها (number of concurrent)، حداکثر (requests استفاده از RAM و CPU د توسط یک تابع، حداکثر زمان زنده‌ماندن بعد از اجرا توسط تابع و ... است. البته برخی از محدودیت‌ها را می‌توان با صرف هزینه یا خرید پلن‌های درامدی سطح بالاتر برطرف کرد. مثلا پلتفرم functions Lambda AWS می‌تواند با صرف هزینه‌ی بیشتری حداکثر تعداد درخواست را افزایش داد. درحالی‌که این مورد در پلتفرم‌های اوپن سورس وجود ندارد. در حالت کلی پلتفرم‌های متن بازی مثل openfaas محدودیت‌های بیشتری را برای کاربر اعمال می‌کنند. علت این امر می‌تواند این باشد که این پلتفرمّ‌ها از نظر تکنولوژی و بازدهی (performance) کاملا از همتایان تجاری خود عقب هستند.

۳. زبان برنامه‌نویسی[1]

پلتفرم‌های بدون سرور از گستره‌ی عظیمی از زبان‌های برنامه‌نویسی پشتیبانی می‌کنند که شامل جاوااسکریپت، گو، پایتون، جاوا، سی‌شارپ، سویفت و پی‌اچ‌پی می‌شود. اکثر پلتفرم‌ها حداقل از ۵ زبان برنامه‌نویسی پشتیبانی می‌کنند. همچنین بسیاری از پلتفرم‌ها مستقل از زبان[2] هستند. یعنی درحالی‌که در داخل کانتینر اجرا می‌شوند (مثلا کانتینرهای داکر)، دیگر زبان برنامه‌نویسی برای آن‌ها اهمیتی ندارد. این پلتفرم‌ها توابع را در داخل کانتینر اجرا می‌کنند و نتیجه را برمی‌گردانند.

۴. مدل برنامه‌نویسی [3]

مدل‌های مختلفی برای تولید یک متد در پلتفرم‌های بدون سرور داریم. شیوه متداول استفاده از یک متد به نام main است که درون آن تابع اصلی تعریف می‌شود. همچنین معمولا ورودی‌های تابع در قالب شیئ‌های json تعریف می‌شوند.

۵. ترکیب توابع[4]

روش‌های گوناگونی برای اینکه یک تابع یا جریان کاری پیچیده‌را پیاده سازی کنیم وجود دارد. یک روش

[1] Programming Languages
[2] Language Independent
[3] Programming Model
[4] Compositions

استفاده از ترکیب‌های توابع است. تا به حال ۷ ترکیب مختلف شناسایی شده است. پلتفرم‌های تجاری [۱]هایی برای پیاده سازی این ترکیبات در خود تعبیه کرده‌آند. متاسفانه پلتفرم‌های متن باز مثل [۲] از این ترکیب پشتیبانی نمی‌کنند. در ادامه و در بخش کارهای مرتبط این ویژگی‌ها مطرح خواهند شد. کاربرد اصلی ترکیب توابع پیاده‌سازی عملکردهای پیچیده در پلتفرم بدون سرور است.

۶. استقرار[۳]

پلتفرم‌ها سعی می‌کنند پیاده‌سازی‌ها در رایانش بدون سرور را تا حد ممکن ساده کنند. این یکی از دلایل به وجود آمدن این مدل رایانشی بوده. به صورت معمول پلتفرم‌ها برنامه‌ها را در قالب کانتینرهای داکری دریافت می‌کنند و درون کانتینر مربوطه کد را اجرا می‌کنند. علاوه بر داکر پلن‌هایی از جمله دریافت کد باینری، دریافت سورس کد و سپس کانینرایز کردن آن وجود دارد.

۷. امنیت و حسابدری[۴]

این دو مورد در کنارهمدیگر به کار برده می‌شوند که معمولاخارج از بحث‌های رایانش ابری کاملا جدا از همدیگر به کار برده می‌شوند. در رایانش بدون سرور لازم است که اپ‌ها کاملا از همدگیر جدا اجرا شوند به این دلیل که بتوانیم برای هر کاربر هزینه‌ای که باید پرداخت کند را محاسبه کنیم. درصورتی که اجرای کاربران از همدگیر تفکیک شده نباشد، محاسبه‌ی هزینه ممکن نیست. اما اجرای جداگانه‌ی توابع از یکدیگر علت دیگری نیز دارد، امنیت. لازم است که توابع جداگانه اجرا شوند تا در توابع و کاربران نتوانند در کارهای همدیگر دخالتی داشته باشند. این مورد حتی می‌تواند باعث به وجود آمد باگ‌های امینتی و دسترسی کاربران به سیستم کاربران دیگر از طریق مدل رایانشی ما شود.

۸. پایش و اشکال زدایی[۵]

هر پلتفرم رایانشی امکاناتی از جمله پایش اولیه برای درخواست‌ها را به کاربر می‌دهد. البته این بحث یکی از مسائل باز در این حوزه است و نیاز به بررسی بیشتری دارد. در حال حاضر دیباگینگ از طریق تجزیه

[1] orchestrator
[2] openfaas
[3] Deployments
[4] Security and Accounting
[5] Monitoring and Debugging

و تحلیل لاگ‌های سیستم ممکن است ولی ممکن است در آینده بهبودهایی در این حوزه حاصل شود. علت اینکه دیباگینگ بسیار چالش برانگیز است این است که در رایانش بدون سرور اپ‌های ما کانتینرایز می‌شوند و چون محیط کانتیر محیطی ایزوله است، امکان مطالعه و دیباگینگ ممکن نیست. بعلاوه توابع تنها در حالت استفاده در پلتفرم زنده هستند؛ پس مدت زمان اشکال‌زدایی ما نیز بسیار محدود می‌شود. باید به این نکته دقت داشت که به طور متوسط در رایانش بدون سرور، توابع ۹۹درصد زمان را از تاریخ استقرار روی سرور، درخواب هستند. اما در مورد پایش نرم‌افزار پلتفرم بدون سرور در داشبورد مدیریتی خود امکاناتی جهت مشاهده منابع مصرف شده، منابع آزاد مدت زمان استفادهشده، تعداد درخواست‌ها و فراخوانی ها، تعداد شروع‌های سرد و ... دارد. به‌علاوه ابزارهای پایش مانند prometheus به خوبی با این پلتفرم‌ها امکان اتصال دارند و با پنل‌هایی مانند grafana می‌توان از مانیتورینگ مضاعف برای این سرورها بهره برد.

۴.۱.۲ پلتفرم‌های تجاری

پلتفرم‌های اندکی برای این قسمت وجود دارد. معروف‌ترین آن‌ها عبارتند از : AWS Lambda Funcitons ، Google Cloud Functions، Microsoft Azure Functions و IBM OpenWhisk

۱. AWS Lambda Functions

پلتفرم AWS [۶] پلتفرم ارائه شده در بحث رایانش بدون سرور بود که دارای خلاقیت‌های بسیاری بود. از مدل‌برنامه‌نویسی، مدل هزینه‌ای، محدودیت منابع، امنیت و مانیتورینگ مخصوص خود استفاده می‌کند. همچنین AWS از زبان‌هایJava، Node.js، Python و سی‌شارپ پشتیبانی می‌کند. این پلتفرم ارتباط خوبی با سایر خدمات و سرویس‌های AWS دارد و در این اکوسیستم اصطلاحا حل شده است.

۲. Functions Cloud Google

پلتفرم شرکت گوگل با نام Google Cloud Functions [۷]به تازگی از حالت آلفا خارج شده. این سرویس از زبان‌های بسیاری ساپورت نمی‌کند ولی به خوبی به درخواست‌های HTTP و HTTPS پاسخ می‌دهد. درحال حاضر اگرچه عملکرد محدودی برای این پلتفرم شاهد هستیم ولی با توجه به سابقه گوگل و معماری

متفاوت این پلتفرم، آینده خوبی برای آن می‌توان متصور بود. این پلتفرم هنوز به خوبی با سرویس‌های رایانش ابری گوگل ارتباط برقرار نکرده و جای کار بیشتری دارد.

۳. Functions Azure Microsoft

پلتفرم بعدی، پلتفرم Microsoft Azure Functions [۸]است. این پلتفرم وب‌هوک‌های HTTP را برای تعامل با کاربر پیاده سازی کرده است. از زبان‌های Bash، PHP، Python، Node.js، سی‌شارپ و اف‌شارپ یا هر زبان اجرایی (چون از کانتینترهای داکری استفاده می‌کند) پشتیبانی می‌کند. بخشی از کدها و پروژه‌های انجام شده با این پلتفرم توسط مایکروسافت در گیت‌هاب این پروژه متن‌باز شده اند. همچنین برای راحتی دیباگنگ مایکروسافت در CLI مربوطه امکان Caching یا استفاده از حافظه موقت را گنجانده است. این پلتفرم به مقبولیت قابل قبولی در بین جوامع توسعه‌دهندگان رسیده و روز به روز بر امکانات آن افزوده می‌گردد.

۴. OpenWhisk Apache

پلتفرم آخر، پلتفرم OpenWhisk [۹]است که در برابر پلتفرم‌های دیگر البته بسیار ساده‌تر به نظر می‌رسد. این پلتفرم اپن‌سورس توسط شرکت IBM تولید و پشتیبانی می‌شود. از قابلیت استفاده زنجیره‌ای توابع بهره‌می‌برد و در مبحث Orchestration توابع از پلتفرم‌های رقیب خود جلوتر است (منبع به مقاله ۱). همچنین OpenWhisk توانایی اجرای هر تابعی را دارد؛ زیرا از داکر به عنوان runtime نیز استفاده می‌کند. سورس این پروژه در آدرس گیت‌هاب OpenWhisk موجود است. در شکل زیر نیز می‌توان معماری آن را مشاهده کرد.

همانگونه که در شکل بالا مشخص است. این معماری خیلی به معماری مینیمال یک پلتفرم بدون سرور شبیه است. البته در مقایسه با شکل قبل امکانات بیشتری از جمله امنیت، مانیتوریگ و لاگ‌گیری را اضافه کرده است.

۵.۱.۲ پلتفرم‌های آزاد و متن باز

علاوه بر موارد فوق پلتفرم‌های متن بازی برای رایانش بدون سرور ارائه شده که در ادامه شرح خواهیم داد.

۱. پتلفرم Open Whisk

اگرچه این پلتفرم در قسمت قبل معرفی شد، اما به صورت متن باز وجود دارد و تنها IBM با پیاده سازی و ادغام در پلتفرم bluemix که پلتفرم ابری شرکت IBM است، کسب درآمد می‌کند. این پلتفرم را به سادگی می‌توان بر روی ماشین‌های مجازی یا دستگاه‌های شخصی پیاده سازی کرد.

۲. پلتفرم Openfaas

پلتفرم بعدی، پلتفرم openfaas است که توسط جوامع ازاد توسعه داده شده است. پشتیبانی از زبان‌های جاوا، سی‌شارپ، پایتون و ... از ویژگی‌های آن است. برای کانتیرسازی نیز از داکر به عنوان cli و موتور کانتینر سازی استفاده می‌کند. همچنین registery پیش فرض در این ابزار داکر ریجستری است. جامعه‌ی رو به رشدی دارد و برای بسیاری از پروژه‌های کوچک و شرکت‌های متوسط مناسب است.

۳. پلتفرم Open Lambda

این پلتفرم از پلتفرم‌های جدید و متن باز است که تلاش می‌کند بسیاری از چالش‌ها و مسائل باز این حوزه را به طور خلاقانه‌ای حل کند. از ویژگی‌های پلتفرم open lambda میتوان به زمان اجرای سریع‌تر تابع ها به خاطر زمان شروه بهتری نسبت به سایر نمونه ها نام برد. همچنین از توابع state-ful هم پشتیبانی می‌کند. همچنین استفاده از توابع بدون‌سرور با دیتابیس‌ها و دیباگینگ بهتر فراهم شده است. [۱۰]

۲.۲ Function-as-a-Service

واژه‌ی بعدی که در این حوزه بسیار مطرح می‌شود واژه‌ی Function-as-a-Servcie یا به اختصار FaaS است.

FaaS یک دسته‌بندی جدید در سرویس‌های رایانشی است که با استفاده از یک پلتفرم بدون سرور، به اجرا، توسعه یا مدیریت توابع، بدون هیچ‌گونه پیچیدگی خاص یا نگرانی برای نگه‌داری زیرساخت که برای استقرار و پیاده‌سازی یک اپلیکیشن که سابقا و در مدل‌های غیر بدون سرور، باید کانفیگ می‌کرده ایم.

تولید یک نرم افزار بر اساس FaaS، روشی برای رسیدن به یک مدل رایانشی بدون سرور است و در قالب

توسعه‌ی میکروسرویس‌ها و نانوسرویس‌هایی در توابع، به‌دست می‌آید. از آنجایی که FaaS به صورت حین تقاضا[1] به ما خدمات می‌دهد برای توسعه‌ی خدماتی که به تجزیه و تحلیل داده نیاز دارند مانند سرویس‌های اینترنت اشیا، برنامه‌های موبایل و وب اپلیکیش‌ها بسیار کاربرد دارد.

حال می‌توان به مقایسه بین رایانش بدون سرور با FaaS پرداخت، بر اساس تعریف FaaS را می‌توان یک پیاده سازی از رایانش بدون سرور نامید. همچنین رایانش بدون سرور منتهی به اجرای توابع تحت سرویس نمی‌شود. بلکه حوزه های وسیع تری ازجمله Mobile-Backend-as-a-Service یا درمواردی PaaS را شامل می‌شود.

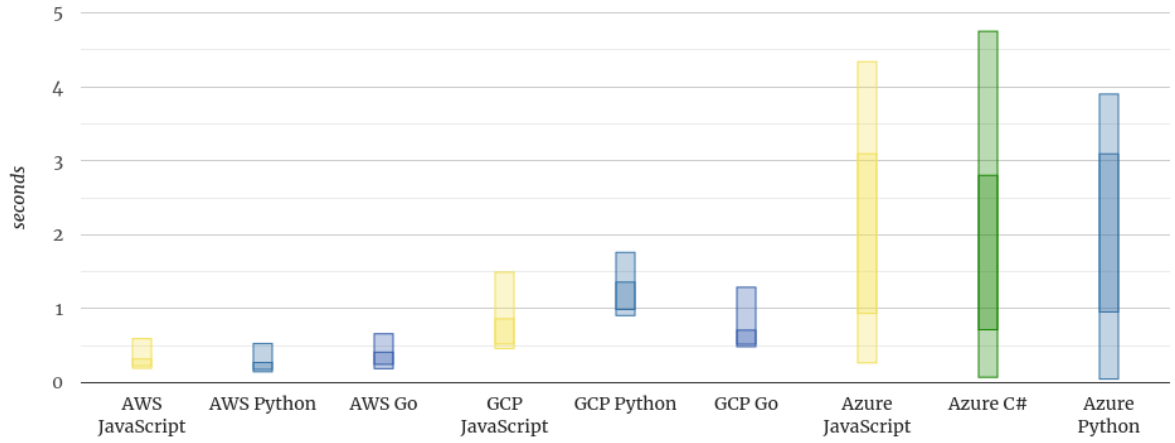
۳.۲ Scale-to-Zero

یکی از ویژگی‌های کلیدی در رایانش بدون سرور، قابلیت Scale-to-Zero است. این قابلیت موجب پیاده‌سازی پلن‌های هزینه مانند پلن هزینه ردخت حین مصرف[2] می‌شود. قابلیت scale-to-zero یعنی اینکه در پلتفرم ما هرگاهی که تابع مدت زیادی بلا استفاده باشد، منابع پردازشی آن گرفته می‌شوند و آن تابع اصطلاحا zero-scale می‌شود تا تابع دیگر فعال نباشد. اگر از دید ارائه دهنده خدمات به این موضوع نگاه کنیم، برای ما این فایده را دارد که منبع بلااستفاده ما در این حالت آزاد می‌شود و آن منبع (RAM و CPU) به کانتینر دیگری که می‌خواهد استفاده شود اختصاص می‌یابد. طبق آمار توابع در FaaS به ندرت صدا زده می‌شوند. ۸ طبقه‌بندی برای فراخوانی توابع انجام شده و مشخص شده که ٪۵۰ توابع زیر ۱ ثانیه و ٪۹۶ آن‌ها زیر ۹ ثانیه اجرا می‌شوند [۴]. بنابراین اگر منبع به یک تابع به مدت زیادی اختصاص یافته باشد دچار اتلاف منابع زیادی خواهیم شد.

از دید کاربر هم اگر بخواهیم به موضوع نگاه کنیم. Scale-to-Zero باعث فراهم شدن ویژگی پرداخت حین استفاده می‌شود. این صرفه اقتصادی بزرگی برای ما دارد. فرض کنید یک اپلیکیشن درحال اجرا داریم که اجرای انفجاری دارد و این اپلیکیشن به ندرت اجرا می‌شود. اگر بخواهیم از مدلّ‌های قدیمی پرداخت استفاده کنیم، باید هزینه زیادی صرف کنیم، درحالی‌که در اکثر اوقات تابع ما هم بلا استفاده است. در حالی که با مدل پرداخت در سیستم‌های بدون سرور، این مورد بسیار برای ما به صرفه می‌شود.

دو مورد بالا از مزایای قابلیت Scale-to-Zero در سیستم‌های بدون‌سرور هستند. اما در این‌صورت با یک

[1] on-demand
[2] pay-as-you-go



شکل ۳.۲: اهمیت زبان و پلتفرم بدون سرور در تاخیر شروع سرد

چالش جدی به نام تاخیر شروع سرد هم مواجه خواهیم شد که در ادامه به آن می‌پردازیم.

۴.۲ تاخیر شروع سرد

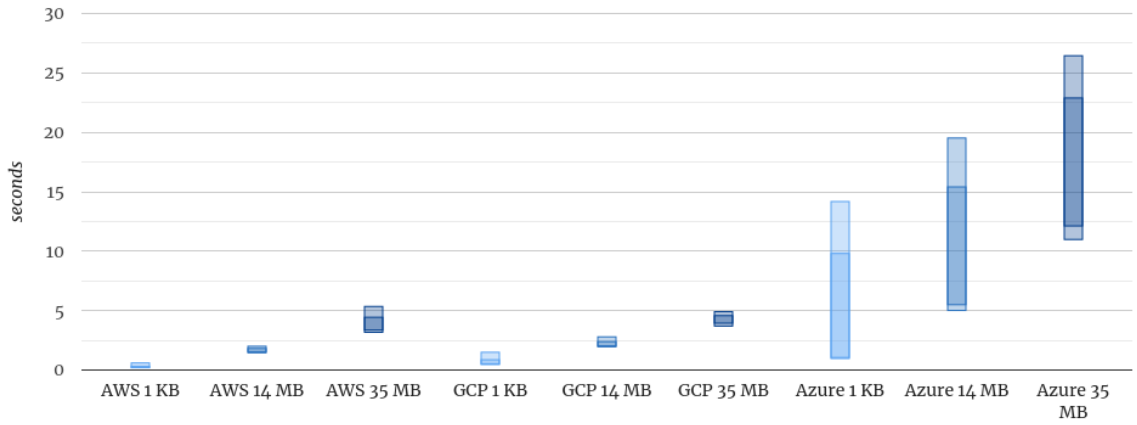
درقسمت قبل به بیان ویژگی Scale-to-Zero در پلتفرم‌های بدون سرور و مزایای آن پرداختیم. اما این ویژگی معایبی هم دارد. یکی از مهم‌ترین معایب آن، تاخیر شروع سرد[1] است.

بگذارید تاخیر شروع سرد را در قالب یک مثال بیان کنیم. فرض کنید پس از مدتی بلا استفاده بودن تابع منابع آن گرفته شده و اصطلاحا سرد شده است. حال یک درخواست برای اجرا برای آن تابع سرد شده می‌رسد. درحالی که تابع ما منبعی ندارد و اجرا نمی‌شود؛ آن درخواست باید مدتی منتظر بماند تا تابع مورد نظر ما دوباره آماده شود و در سرور بارگذاری شود. این آماده سازی، مراحل مختلفی دارد که در تصویر ۳.۲ نشان داده شده است.

همانگونه که در تصویر ۳.۲ می‌بینیم، این آماده سازی شامل آماده سازی کانتینرها، آماده سازی تابع، اختصاص منابع به کانتینر و در نهایت، اجرا در پلتفرم است. این تاخیر بسیار قابل توجه است و در شکل ۱.۱ نیز به خوبی نشان داده شده است.

همچنین، یکی از مواردی که شروع سرد اتفاق می‌افتد، فراخوانی تابع برای اولین بار در پلتفرم است.

[1] Cold Start Latency



شکل ۴.۲: اهمیت اندازه کتابخانه برنامه در تاخیر شروع سرد

حال چه عواملی در شروع سرد نقش دارند؟ عوامل عمده‌ای در این کار دخیل هستند ولی مهم‌ترین آن‌ها زبان برنامه‌نویسی و نوع پلتفرمی که در آن کد را اجرا می‌کنیم و کانفیگ نرم‌افزاری و سخت‌افزاری آن پلتفرم است.

در مورد اهمیت زبان‌های برنامه نویسی می‌توان گفت چون زبان‌های مختلف زمان اجراهای متفاوتی دارند بنابراین موثر هستند. شکل زیر اهمیت زبان‌های برنامه‌نویسی را نشان می‌دهد. در این شکل از پلتفرم‌های مختلف برای اجرای توابع مختلف برای محاسبه‌ی زمان اجرا با درنظر گرفتن تاخیر شروع سرد استفاده شده است.

در آزمایش دیگری با جاوا اسکریپت تابع‌های یکسانی نوشته شده با این تفاوت که حجم کتابخانه‌های هر تابع متفاوت است. این آزمایش در پلتفرم‌های مختلف نیز انجام شده و نتیجه طبق شکل زیر رسم شده است.

بنابراین با توجه به شکل ۴.۲ می‌توان نتیجه گرفت حجم کتابخانه توابع نیز در مدت زمان تاخیر شروع سرد بسیار موثر است.

فصل ۳

کارهای مرتبط

قبل از بررسی درخت موضوعی بهتر است به مبحث ترکیب توابع در پلتفرم‌های بدون سرور بپردازیم. در [۱۱] ۷ ترکیب از توابع در FaaS بررسی شده است. البته این مقاله ذکر می‌کند که ۲ دسته‌بندی اصلی در رایانش بدون‌سرور داریم: MbaaS و FaaS. از MbaaS به عنوان سرویس‌های سمت سرور برنامه‌های کاربردی وب و موبایل استفاده می‌کنند. پلتفرم Firebase یک نمونه از آن است. در حال حاضر این دسته‌از سرویس‌های ابری به شدت در حال توسعه است و البته، تمرکز این مقاله بر روی این موضوع هم نخواهد بود.

یکی از چالش‌های اصلی معماری بدون سرور در ان است که برای تولید یک برنامه کاربردی بزرگ یا تجاری، باید جریان‌های کاری پیچیده‌ای را ایجاد کنیم. برای همین کار باید تعداد زیادی تابع را ایجاد کنیم که این توابع باید به اشکال مختلف هم‌دیگر را فراخوانی کنند. مثلا بعضا ممکن است یک تابع بعدی را فراخوانی کند یا یک تابع لازم باشد همزمان چند تابع را فراخوانی و اجرا کند و ... . پیاده‌سازی یک راه حل درست برای این مورد بسیار برای رسیدن به معماری میکروسرویس برای ما مهم و حیاتی است و هرگونه کوتاهی و خطا در راه‌حل، باعث کارایی پایین محصول نهایی خواهد شد و با مشکلات بسیاری در پیاده‌سازی، ما را مواجه می‌سازد.

برای حل این معضل، ارائه دهندگان خدمات تجاری اقدام به معرفی سرویس‌هایی تحت عنوان FaaS Orchestrator نموده‌اند. هدف این سرویس‌ها ایجاد و پشتیبانی از ترکیب‌توابع و سناریوهای پرکاربرد برای دستیابی به عملکرد روان در برنامه‌های کاربردی تحت پلتفرم بدون سرور است. دو پلتفرم معروف Orchestrator عبارتند از: AWS Step Functions (با به اختصار ASF) و دیگری IBM Cloud Function Sequences.

توابع ترکیب شده توسط Orchestratorها حتما باید ۳ معیار را ارضا کنند:

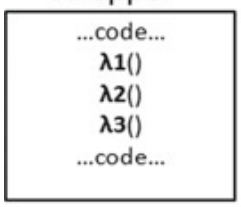
۱. ترکیبات توابع باید به گونه‌ای باشند که جعبه سیاه بودن توابع نقض نشوند. یعنی فقط باید بتوانیم از روی ورودی و خروجی توابع محتوای آن را حدس زد.

۲. این ترکیب‌ها تابع قوانین و قواعد مشخصی باشند. همچنین، این ترکیبات باید قابل جایگزینی باشند.

۳. نباید بگونه‌ای فراخوانی اتفاق بیافتد که برای محاسبه‌ی هزینه همزمان پول ۲ تابع یا بیشتر را بدهیم[1].

ما از این سه قانون با عنوان لم‌های بدون سرور[2] یاد می‌کنیم و هر ترکیبی از توابع که این ۳ معیار بالا را ارضا

[1] double billing
[2] Serverless Trilemma




```
import boto3, json
def lambda_handler(event, context):
    invokeLam = boto3.client("lambda", region_name="us-east-2")
    payload = {"message": "Test"}
    resp = invokeLam.invoke(FunctionName="Lambda1", InvocationType = "Event",
Payload = json.dumps(payload))
    print("Lambda1 Invoked ")
    resp = invokeLam.invoke(FunctionName=" Lambda2", InvocationType =
"Event", Payload = json.dumps(payload))
    print("Lambda2 Invoked ")
    resp = invokeLam.invoke(FunctionName=" Lambda3", InvocationType =
"Event", Payload = json.dumps(payload))
    print("Lambda3 Invoked  ")
       return {
        'statusCode': 200,
        'body': json.dumps('Hello from Lambda!')
    }
```

شکل ۱.۳: الگوی بازتابی در فراخوانی‌ها

کند، ST-Safe نامیده می‌شود. ترکیبات معروف توابع به شرح زیر است:

۱. ترکیب با استفاده از بازتاب[1]: به این صورت است که یک تابع اقدام فراخوانی سایر توابع به صورت همزمان[2] می‌کند. در شکل ۱.۳ یک نمونه مثال از فراخوانی همزمان توابع ذکر شده است. تنها مشکلی که دارد این است که با معضل Double billing مواجه خواهیم شد.

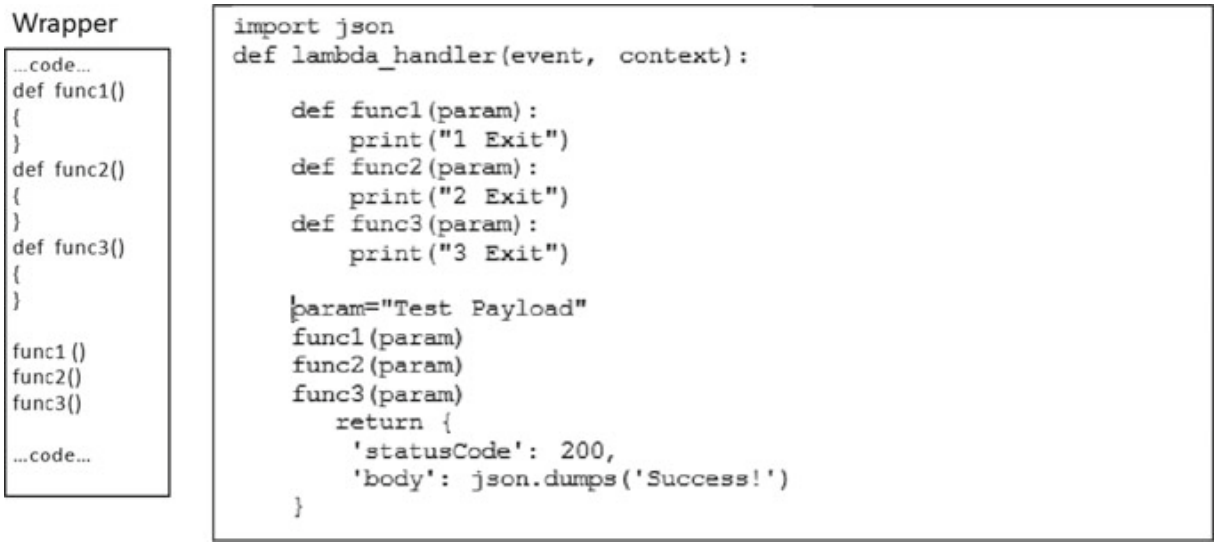
۲. ترکیب همجوشی[3]: در این حالت تابع wrapper، توابعی که فراخوانی کرده‌ایم در تابع اصلی را بارگذاری می‌کند. مشکل اصلی آن این است که اصل جعبه سیاه را نقض می‌کند و همچنین باید توابع حتما با یک زبان نوشته شوند. شکل ۲.۳ یک الگوی همجوشی را نشان می‌دهد.

۳. ترکیب غیرهمزمان: در این حالت، تابع اول به فراخوانی تابع دوم می‌پردازد درحالی که اولی دیگر فعال نیست. این ترکیب نیز قانون دوم ST را نقض می‌کند.

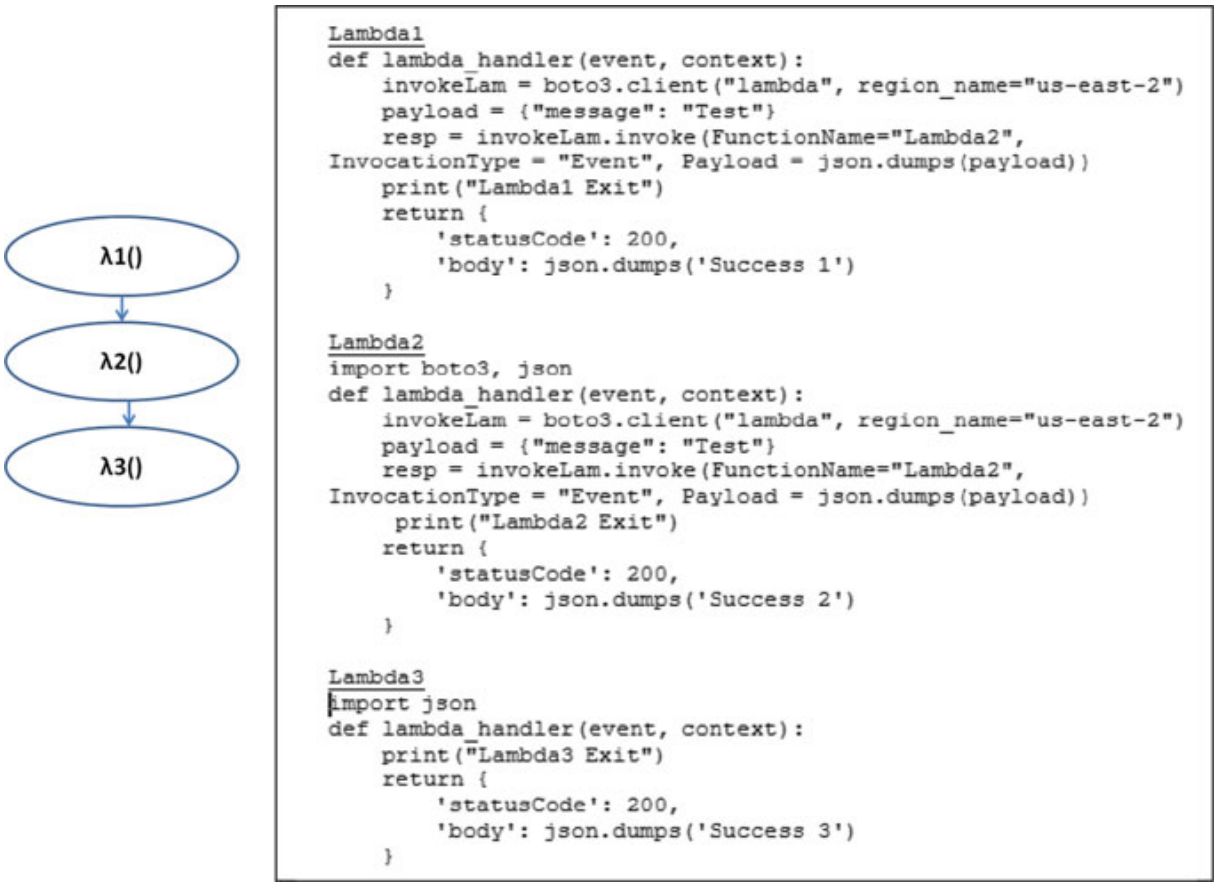
۴. ترکیب توسط مشتری: در این حالت توابع ساخته می‌شود و خود مشتری خارج از سیستم بدون سرور اقدام به ترکیب توابع می‌کند. معروف ترین نمونه آن ASF است و اصل اول را نقض می‌کند.

۵. ترکیب زنجیره‌وار: در این ترکیب، یک ترکیب پس از اتمام اقدام به فراخوانی تابع بعدی می‌کند و همینطور ادامه پیدا می‌کند. این ترکیب مشکل double billing دارد و ST-Safe نیست. در شکل ۳.۳ الگوی زنجیره‌ای نمایش داده‌شده است.

[1] Reflection
[2] synchronous
[3] Fusion



شکل ۲.۳: الگوی همجوشی در فراخوانی‌ها



شکل ۳.۳: الگوی همجوشی در فراخوانی‌ها

جدول ۱.۳: مقایسه‌ی بین روش‌های ترکیب توابع

نوع ترکیب	پشتیبانی از poly glot	کدام محدودیت ST نقض می‌شود؟	مدت زمان اجرای برنامه تست
بازتاب	بله	پرداخت مجدد[۱]	311.0.
همجوشی	خیر	جعبه‌سیاه[۲]	2.47
غیرهمزمانی[۳]	نامشخص	نامشخص	نامشخص
ترکیب توسط مشتری[۴]	بله	قاعده ترکیب توابع	331.83
زنجیره‌ای	بله	پرداخت مجدد	1104.78

مقایسه‌ی این ترکیبات در جدول ۱.۳ به طور خلاصه بیان شده است: [۵]

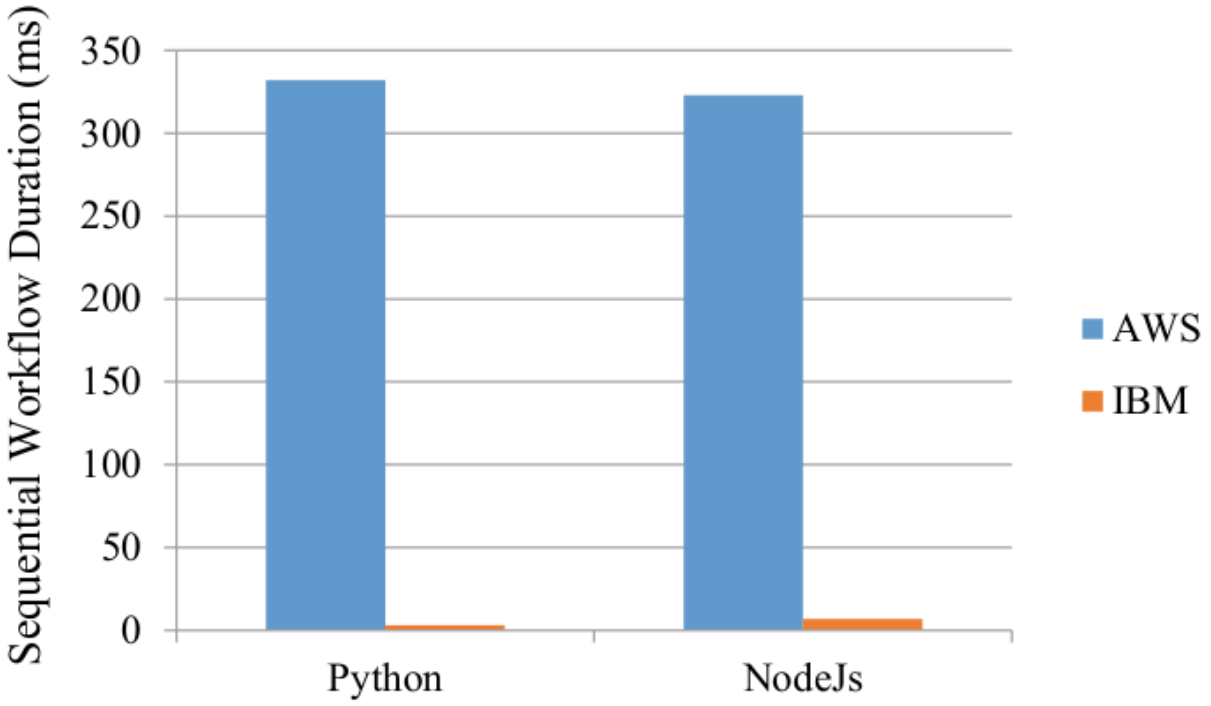
در نهایت با مقایسه دو پلتفرم ASF و IBM Cloud Function Sequences برای یک برنامه تست مشابه به مدت‌زمان های اجرای شکل ؟؟ می‌رسیم.

۱.۳ درخت موضوعی

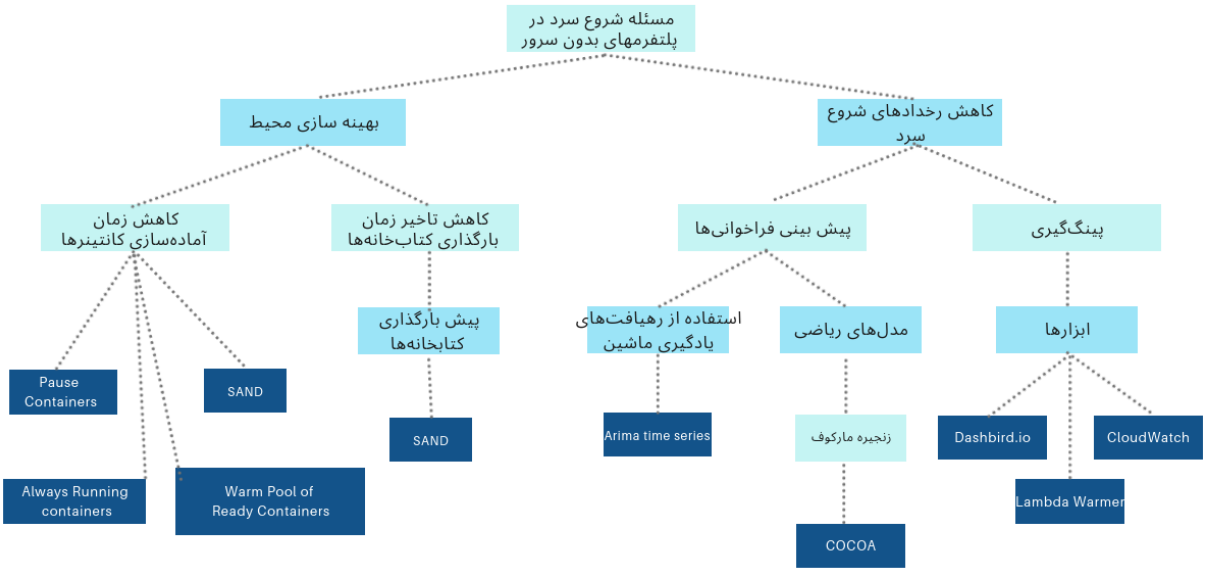
در این قسمت به بیان کارهای انجام شده در این حوزه خواهیم پرداخت و برای این کار درخت موضوعی مرتبط با آن رسم شده. هر گره ار درخت موضوعی یک راهکار برای جلوگیری از رخداد شروع سرد است و هر گره دارای بخش‌هایی است تا به برگ برسیم. در شکل ۵.۳ درخت موضوعی نمایش داده شده است و هر گره را به ترتیب بررسی خواهیم کرد.

ریشه درخت مسئله شروع سرد است که ۲ گره اصلی دارد. گره سمت چپ که بهینه سازی محیط نام دارد، تلاش می‌کند تا در صورت وجود رخداد شروع سرد زمان آن‌را به حداقل برساند. این گره ۲ فرزند دارد که عبارتند از کاهش تاخیر آماده سازی کانتینرها و کاهش زمان بارگذاری کتابخانه‌ها. یک روش برای کاهش زمان بارگذاری کتاب‌خانه ها استفاده از روش پیش-بارگذاری است.

گره سمت راست ریشه هم کاهش رخدادهای شروع سرد است. تمرکز این گره در این است نگذاریم شروع سرد



شکل ۴.۳: مقایسه زمان اجرا ۲ برنامه مشابه با Node.js و پایتون در دو پلتفرم ASF و IBM Cloud Function Sequences



شکل ۵.۳: درخت موضوعی

رخ دهد. گره سمت راست این نود، روش پینگ‌گیری است؛ در این روش سعی می‌کنیم با استفاده از ابزارهایی جلوگیری کنیم از سرد شدن توابع. در آخرین سطح این گره هم ابزارهای مرتبط معرفی شده‌اند. فرزند سمت چپ گره کاهش رخداد، پیش‌بینی فراخوانی‌ها است. در پیش‌بینی فراخوانی ها می‌توان از رهیافت‌های بدست آمده در یادگیری ماشین بهره جست. روش دیگر استفاده از مدل‌های ریاضیاتی غیریادگیری ماشین برای پیش‌بینی فراخوانی‌ها است. یکی از این روش‌های استفاده از ماشینّ‌های حالت محدود بدست آمده با روش‌زنجیره مارکوف است.

در برگ‌های این درخت هرکدام یک مقاله را بررسی می‌کنیم. در ابتدا به توضیح درباره روش‌های بهینه‌سازی محیط می‌پردازیم و سپس به سراغ کاهش رخداد میرویم. در هر حوزه یک مقاله نمونه بررسی خواهد شد.

۲.۳ بهینه سازی محیط

در این روش قرار به این است که مانع وقوع شروع سرد نشویم (در واقع هم نمی‌توان مانع از اتفاق شروع سرد شد)؛ بنابران به دنبال روش یا روش‌هایی برای کاهش زمان شروع سرد هستیم. همانگونه که بالاتر ذکر کردیم، این موضوع را می‌توان از دو جنبه بررسی کرد. اول اینکه آماده‌سازی کانتینرها را کاهش دهیم. روش دیگر این است که زمان لودشدن کتابخانه‌ها برای اجرای تابع را کاهش دهیم. برای این کار می‌شود از روش پیش-بارگذاری کتابخانه ها استفاده کرد.

۱.۲.۳ کاهش زمان آماده‌سازی کانتینرها

در این روش دنبال کمینه‌سازی زمان شروع سرد با استفاده از روش‌هایی برای کانفیگ بهینه محیط برای مواجهه با شروع سرد هستیم. عمده کارهایی که در این بخش انجام می‌دهیم در سطح شبکه یا کانتینرها برای بهینه سازی است که روش‌های نسبتا سطح پایینی محسوب می‌شوند.

یکی از مواردی که شروع سرد به شدت رخ می‌دهد، زمانی است که بنابه‌دلایلی تابع درخواست‌های زیادی دارد. این موضوع در [۲] ذکر شده است. نویسنده معتقد با انجام این بهبودها در حدود ٪۸۵ مدت‌زمان شروع سرد برای این توابع صرفه‌جویی خواهد شد. این مقاله از پلتفرم Knative برای پیاده‌سازی تغییرات استفاده می‌کند.

علت انتخاب Knative این است که بر روی بستر کوبرنتیز ساخته می‌شود و از مفاهیمی مثل Pod ها برای اجرای توابع و جریان‌های کاری استفاده می‌کند. بنابراین، از آنجایی که کوبرنتیز دست ما را برای انجام تغییرات باز می‌گذارد، می‌توان به آسانی به پیاده سازی سیاست‌های[1] خودمان بپردازیم. در پلتفرم Knative هر تابع در درون یک پاد اجرا می‌شود. پادها، ابتدایی‌ترین و ساده‌ترین بارهای‌کاری (به هر برنامه درحال اجرا در کوبرنتیز بارهای کاری می‌گوییم. توجه داشته باشید در کوبرنتیز بارکاری یک موجودیت نیست در واقع مفهومی است که به اجرای کانتینرها و تخصیص CPU و ... اشاره دارد.) در کوبرنتیز هستند. در درون هر پاد تعدادی کانتینر اجرا می‌شود. در یک پاد شبکه‌ها، ذخیره سازی (Storage) به صورت مشترک است. البته باز هم به خاطر وجود بحث‌هایی مثل Cgroups و namespace که ساختمان داده اصلی کانتینرها هستند، کانتینرهای داخل یک پاد از هم ایزوله هستند. پادهادر کوبرنتیز موجودیت‌های موقتی هستند و درصورت از دست رفتن نود، اتمام کار، کمبود منابع سرور و دلایل دیگر می‌توانند از سرور خارج شوند و دیگر قابل بازیابی نیستند.

نکته‌ای که باید توجه داشت این است که یک پاد از جنس یک پردازه[2] نیست؛ بلکه، محیطی منطقی برای اجرای کانتینرهاست و این کانتینرها هستند که از جنس پردازه‌ها هستند. داده‌های درون کانتینرها وابسته به پادها هستند و با ری‌استارت شدن پادها محتویات ذخیره شده در کانتینرها از بین می‌روند مگر اینکه در ذخیره‌سازها ذخیره بشوند. [۱۲]

با توجه به مقدماتی که در مورد پادها ذکر شد، اکنون منطقی به نظر می‌رسد برای مدیریت کانتینرها بخواهیم از پادها استفاده کنیم و این رهیافت دست ما را برای اعمال تغییرات مختلف روی پلتفرم باز می‌کند.

ما به صورت ایده‌آل دنبال کمترین سربار برای فراخوانی توابع هستیم. هنگامی که برای اولین بار تابع را فراخوانی می‌کنیم دچار تاخیر شروع سرد می‌شویم که در بخش ادبیات موضوع (رفرنس به شروع سرد) در مورد آن مفصلا بحث کردیم. این مشکل در تمامی پلتفرم‌های بدون سرور، مشکل رایجی است. سربار شروع سرد را می‌توان به ۲ قسمت تعیین کرد :

۱. سربار ناشی از اجرای پلتفرم

علت اصلی این سربار اجرای پلتفرم است و به علت قرار گرفتن در صف یا دلایل دیگر باعث تاخیر می‌شود.

[1] policy
[2] Process

از این دسته خطاها می‌توان به network، provisioning pod bootstraping یا sidecar nework اشاره کرد.

۲. سربار ناشی از خود اپلیکیشن

علت اصلی این سربار مشکلات خود برنامه است. این نوع تاخیر به مواردی از جمله زبان برنامه‌نویسی، حجم برنامه و نوع کتاب‌خانه‌هایی که از آن‌ها استفاده می‌کنیم بستگی دارد.

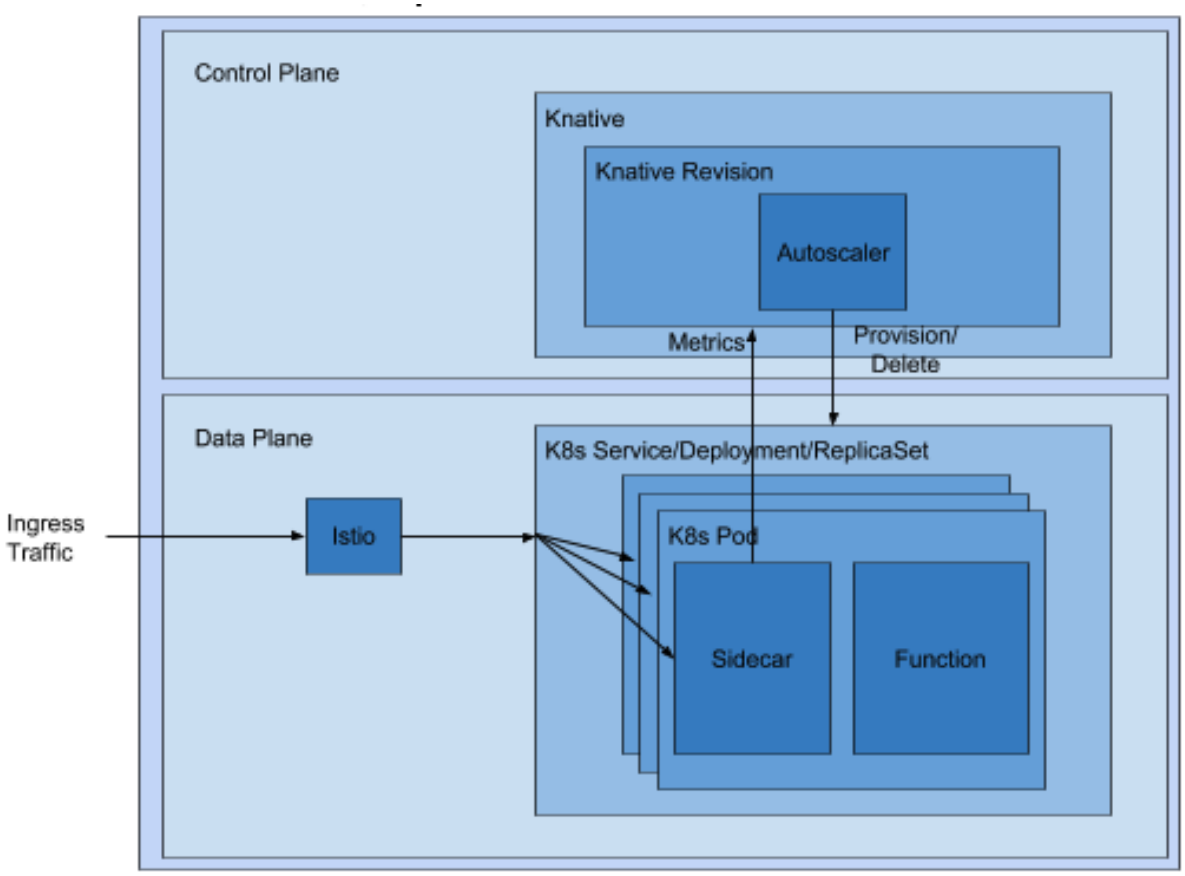
برای مثال اجرای یک HTTP Server ساده شروع سردی در حدود ۵ ثانیه را به خود اختصاص می‌دهد؛ درحالی‌که با اجرای یک برنامه کلاس‌بندی عکس زمان شروع سرد به چیزی در حدود ۴۰ ثانیه هم برسد.

بنابراین، ایده این مقاله این است که برای اجرای توابعی که اخیرا محبوب شده‌اند از رهیافت استفاده از یک استخر گرم برای نگه‌داری این پاد استفاده‌کنیم. دقت کنید که در اینجا، در داخل هر پاد تنها یک کانتینر که آن کانتینر هم برای یک تابع است، اجرا می‌شود. هرگاه که درخواست جدید برای تابع می‌رسد در ابتدا استخر گرم را چک می‌کنیم که آیا پاد در آن موجود است یا خیر؟ اگر پاد در آن وجود داشت دیگر منتظر نمی‌مانیم، سریع تابع را در پلتفرم اجرا کرده و دیگر تاخیر شروع سرد را نخواهیم داشت. بنا به محاسبه نویسنده مقاله، این روش تا ٪۸۵ زمان شروع سرد را برای توابع on-demand نسبت به حالت عادی، کاهش می‌دهد.

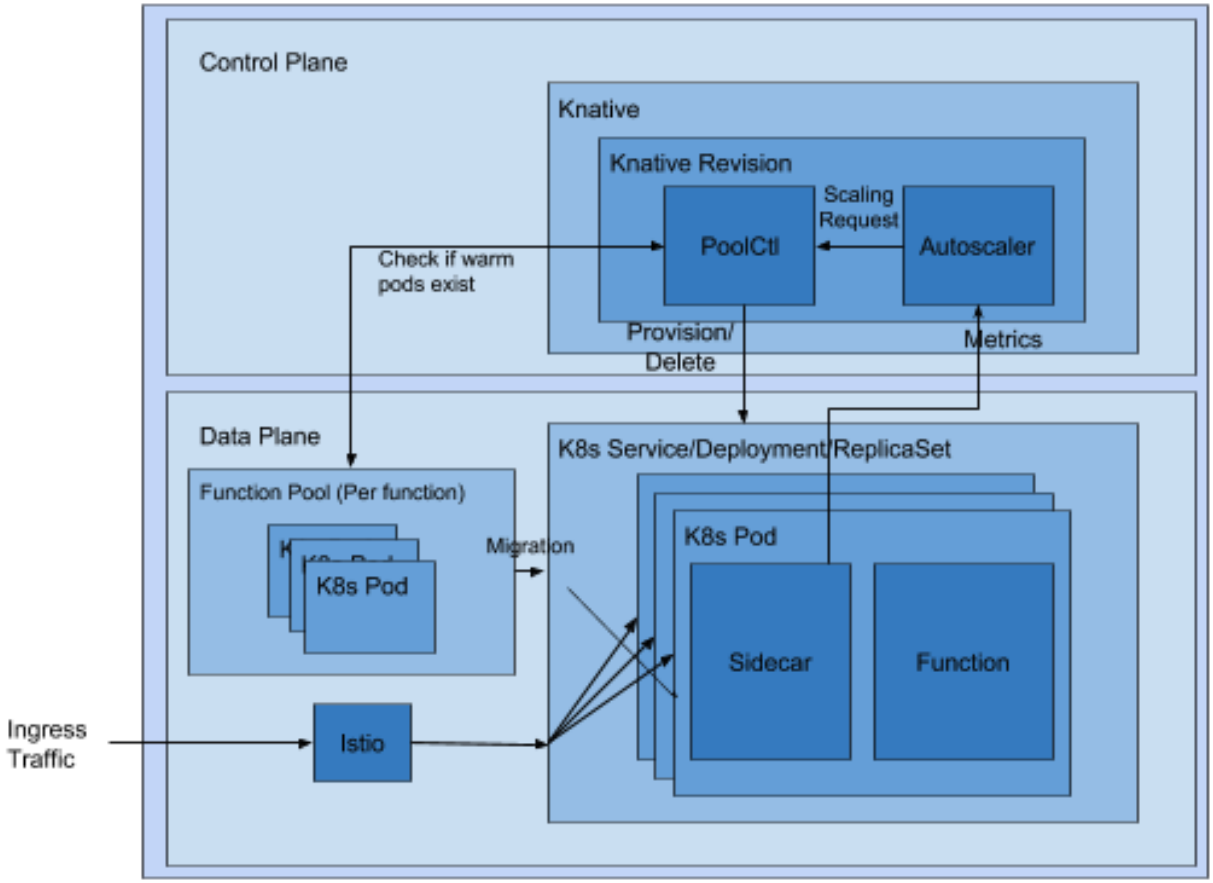
اما چگونه این تغییرات انجام شده است؟ شکل ؟؟ نمایشگر معماری پلتفرم بدون سرور Knative است. انتظار داریم با بهینه‌سازی‌هایی این رهیافت را برای مدیریت شروع‌های سرد اعمال کنیم.

همانگونه که در این شکل می‌بینیم، وظیفه مولفه autoscaler، انجام وظایف مربوط به Scale up/down است. هنگامی که دچار شروع سرد می‌شویم، مولفه autoscaler دستور به ساخت پاد را می‌دهد ولی از آنجایی که این امر وقت گیر است انجام آن بسیار طول می‌کشد. نهایتا اینکه پس از مدت زیادی پاد ساخته شده و داخل بخش data plane اجرا می‌شود.

برای حل این مشکل، مقاله پیشنهاد می‌دهد تا در بخش control plane و داخل revision پلتفرم knative یک مولفه مدیریت استخر قرار دهیم که آن با توجه به درخواست‌هایی که برای auto-scaler می‌رسد، اقدام به ساخت پادهایی و نگه‌داری آن در استخر گرم که در بخش data plane توسعه داده شده، می‌کند. استخر گرم محدودیت هایی مثل اندازه استخر دارد و تنها تعداد محدودی پاد در آن می‌توان نگه داشت. حال اگر درخواستی برای پلتفرم



شکل ۶.۳: معماری پلتفرم Knative

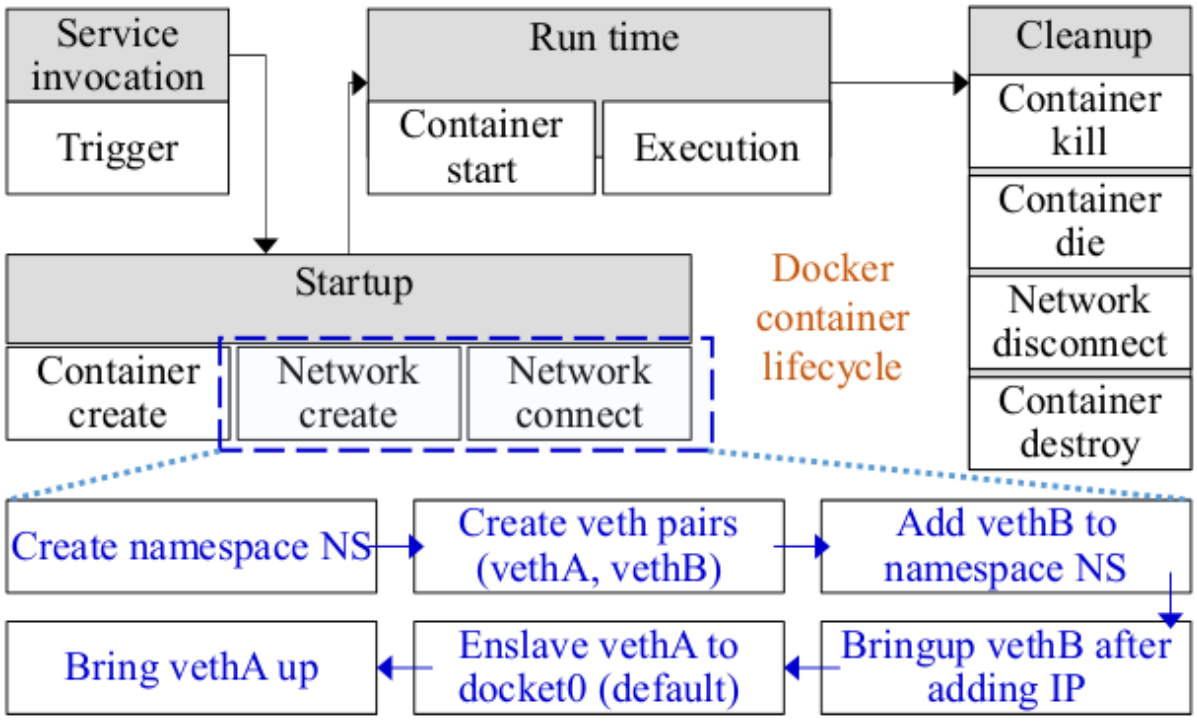


شکل ۷.۳: معماری پیشنهادی مقاله برای حل تاخیر شروع سرد

برسد، autoscaler ابتدا از کنترل کننده استخر[1] وضعیت موجودی در استخر گرم را بررسی می‌کند. اگر در استخر پاد موجود باشد در اینصورت بلافاصله مهاجرت(migration) از استخر گرم به سرویس رخ می‌دهد. از آنجایی که در استخر گرم منابع به پاد اختصاص داده شده و پاد کاملا آماده‌ی اجرا است؛ بنابراین، تاخیر شروع سرد بسیار ناچیزی خواهیم داشت. اما از طرفی، اگر پاد در استخر گرم موجود نباشد، دچار تاخیر شروع سرد خواهیم شد. نکته‌ی منفی این روش این است که در صورت رخداد شروع سرد، به میزان تاخیرهای قبل، تاخیر ناشی از استعلام از استخر گرم هم اضافه خواهد شد. شکل ۷.۳ مدل جدید مقاله برای مدیریت شروع سرد را نشان می‌دهد. [۱۲]

مقاله دیگری که در این حوزه اقدام به بررسی تاثیر آماده سازی کانتیرها پرداخته از ایده container pause ها که مفهومی در کوبرنتیز است استفاده می‌کند. ؟؟ این مقاله اقدام به بررسی تاخیر شروع سرد ناشی از اجرای همزمان تعدادی تابع کرد و نتایج خروجی آن را در شکل ۱.۱ نشان داده‌اند.

[1] pool controller



شکل ۸.۳: مراحل ساخت یک فضای نام در داکر

در این واقع، این مقاله تاخیر شروع سرد را ناشی از آماده سازی کانتینرها می‌بیند و سعی می‌کند تا حد امکان با آن مقابله کند. برای نمایش و پیاده‌سازی سناریو خود از پلتفرم Apache OpenWhisk استفاده کرده‌اند. طبق همان شکل نتیجه می‌گیریم که بخش عمده‌ای از تاخیرها ناشی از تاخیر در آماده‌سازی کانتینرها است، زیرا باید گام‌های طولانی برای ساخت و استقرار یک کانتینر و اختصاص شبکه به آن برداریم. شکل ۸.۳ این گام ها را نشان می‌دهد.

همانگونه که در شکل مشخص است برای اجرای یک تابع در پلتفرم، ابتدا باید کانتیر آن ساخته شود. بسیاری از پلتفرم‌ها از جمله پلتفرم OpenWhisk - که این مقاله تغییرات را بر بستر این پلتفرم بدون سرور انجام می‌دهد، - ازموتور داکر به عنوان موتنو کانتینری در پلتفرم خود پشتیبانی می‌کنند. در این موتور مراحل شکل فوق باید انجام بشود تا یک کانتینر کاملا آماده شود.

با توجه به شکل، تمامی بهبودهایی که باید انجام دهیم در مرحله شروع اولیه کانتینر است، جایی که دقیقا ۳ مرحله داریم. ساخت کانتینرها، ساخت شبکه‌ها و اتصال کانتینرها به آن‌ها و در نهایت اتصال شبکه‌ها.

در مرحله‌ای اول باید فضای نام‌را برای هر شبکه ایجاد کنیم. این کار در داکر توسط یک ویژگی کرنل لینوکس

به نام فضای نام [۱] انجام می‌گیرد. فضای نام یه مانع یا ایزوله کننده شبکه است که فرایند مختلف را از یکدیگر جدا می‌کند. در هر فضای نام پس از ایزوله سازی می‌توان مطمئن بود که دسترسی به پردازه‌های دیگر به شدت محدود شده است. اما، ما به دنبال ایزوله کردن پروسه نیستیم، بلکه به دنبال این هستیم که اجرا پردازه در سیستم عامل ایزوله باشد ولی ارتباط با آن نیز ممکن باشد. بنابراین باید تنظیمات شبکه در آن را انجام دهیم.

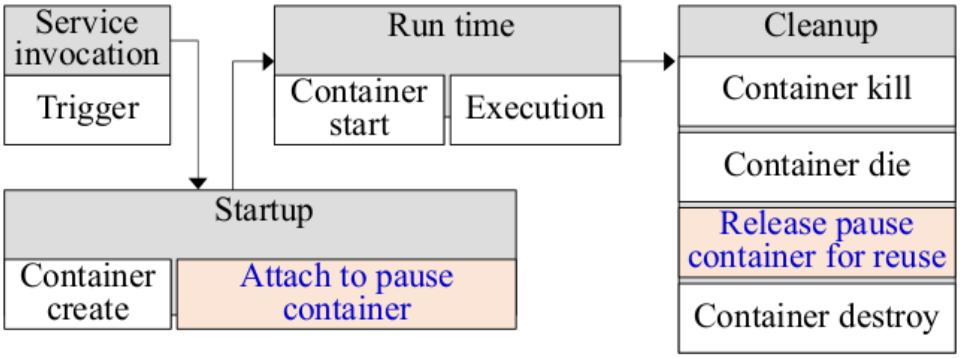
بنابراین به دنبال ایجاد جفت‌های veth [۲] هستیم. جفت‌های veth یک سری کابل مجازی هستند (به طور دقیق از جنس خط لوله‌ها در سیستم عامل هستند) که وظیفه انتقال یک طرفه از کانتینر به فضای بیرون از آن و بالعکس را دارا می‌باشند. پس اقدام به اضافه کردن vethها به شبکه می‌کنیم. این دوقسمتی که مطرح شد خود شامل ۶ مرحله کلی می‌شود که توضیح آن در این گزارش جایی ندارد.

حال نقش شبکه‌ها در شروع سرد چیست؟ همانگونه که قبلا گفتیم، هر کانتینر از ۴ مرحله می‌گذرد. مرحله اول، مرحله فراخوانی سرویس هاست که در آن یک درخواست ساخت کانتینر برای docker daemon ارسال می‌شود. مرحله دوم مرحله آغازکردن [۳] نام دارد که در طی آن یک کانتینر باید برای اتصال به محیط پیرامون آماده شود بنابراین کانتینر ساخته شده، شبکه درون و بیرون کانتینر کانفیگ می‌شود و به هم متصل می‌شوند.. مرحله سوم مرحله اجرا [۴] است که در طی آن، تابع اجرا می‌شود و در انتها مرحله‌ی نهایی یا مرحله پاکسازی [۵] را داریم که شامل توقف اجرای کانتینر، قطع اتصال شبکه و نابود سازی آن است. [۱۳]

علاوه ساخت کانتینرها، به این توجه کنید که ما به دنبال اجرای همزمان آن‌ها نیز هستیم. این مورد زمان و سربار اجرا را نیز به طرز قابل توجهی بالا می‌برد. اگر به شکل ۱.۱ نگاه کنید مجددا می‌بینید که به ازای اجراهای همزمان، زمان آماده‌سازی بسیار طولانی‌تر شده است درحالی‌که، زمان اجرا تغییر چندانی نکرده است.

مقاله می‌گوید که بر طبق آمارهای گرفته شده، در مرحله آماده‌سازی کانتینرها، ٪۹۰ زمان آماده‌سازی مربوط به ۲ مرحله ساخت شبکه‌ها و اتصال شبکه‌ها می‌شود. بنابراین معتقد است که با بهبود در این وضعیت تاخیر شروع سرد تا حد قابل توجهی برای تمامی توابع، کاهش خواهد یافت. حال مشکل اینجااست که این مراحل به کندی انجام می‌شوند مثلا برای ساخت شبکه این کار یکی یکی و در یک صف به نوبت انجام می‌شود. با توجه به اینکه در

[1] namespace
[2] namespace
[3] Startup
[4] execution
[5] Cleanup



شکل ۹.۳: استفاده از Pause Containerها برای کاهش تاخیر شروع سرد

پلتفرم‌های بزرگ و تجاری در هر لحظه تعداد زیادی کانتینر باید ساخته یا حذف شوند این تاخیر به شدت افزایش پیدا خواهد کرد. برای اینکار آزمایشی انجام شد که نتایج آن در جدول ۲.۳ قابل مشاهده است.

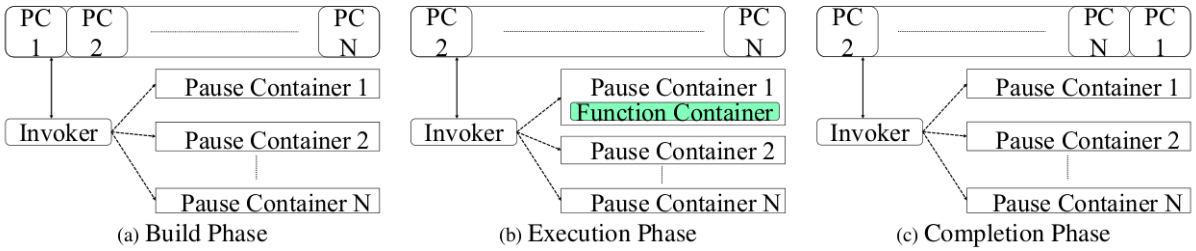
جدول ۲.۳: زمان ساخت و پاکسازی کانتیرهای همزمان

تعداد فضاهای نام همزمان	1	10	50	100
زمان ساخت	0.28	1.27	6.28	14.41
زمان پاکسازی	0.20	0.71	3.24	7.77

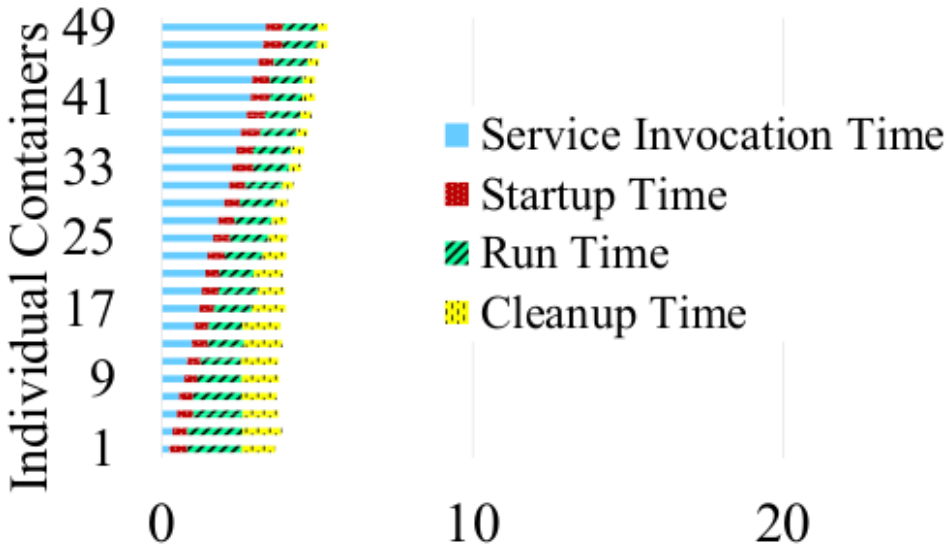
همانگونه که در این جدول قابل مشاهده است، زمان آماده‌سازی سرویس‌ها به ازای تعداد تعداد کانتینرهای همزمان، به طور نمایی افزایش می‌یابد. در حالی‌که ما در پلتفرم‌های تجاری همزمان تعداد زیادی کانتینر را هم بسازیم یا از بین ببریم.

بهتر است به این دید نگاه کنیم که pause container ها از قبل شبکه را ساخته و کانفیگ‌های مربوطه را انجام داده‌اند. پس کانتینری که در درون آن اجرا می‌شود تنها به معرفی برای اختصاص آدرس IP دارند که کار زمان‌بری نیست. بنابراین، می‌توان پس از ساخت یک کانتینر، تنها به اجرای آن در pause container ها بپردازیم و از این طریق در زمان ساخت بسیار صرفه جویی کنیم. همچنین برای پاکسازی تنها باید کانتینر تابع مربوطه را حذف کنیم و pause container بازیابی می‌شود. این رهیافت در شکل ۹.۳ نشان داده شده است.

در این حالت دو مرحله ساخت و اتصال شبکه جایگزین شده. همچنین برای مدیریت pause container ها نیز یک استخر مربوط به آن‌ها ساخته‌شده است. در شکل ۱۰.۳ نحوه عملکرد این رهیافت توضیح داده شده است.



شکل ۳.۱۰: مدیر استخر Pause Containerها



شکل ۳.۱۱: نتیاج و بهبود حاصل شده با استفاده از Pause Containerها

در قسمت اول که فاز ساخت است۷ از قبل تعدادی کانتینر ساخته شده و در یک صف نگهداری می‌شوند. یک فراخواننده[1] داریم که وضعیت تمامی PC[2]ها را می‌داند. در قسمت بعدی می‌خواهیم یک تابع را اجرا کنیم؛ برای اینکار تنها کافی‌است که آن کانتینر را درون PC بارگذاری کنیم. و هرگاه کار تابع تمام شد تنها کافی است آن کانتینر در PC را نابود کنیم. در انتها خود PC بازیابی می‌شود و به انتهای صف استخرهای PC اضافه می‌شود.

نتایج این رهیافت در شکل زیر نشان داده شده است و می‌تواند تا ٪۸۰ زمان شروع سرد را برای توابع کاهش دهد. برای مقایسه میزان بهبود داده شده شکل را با شکل مقایسه کنید.

[1] invoker
[2] مخفف Pause Container

۳.۳ کاهش رخدادهای شروع سرد

در این قسمت به دنبال روش‌هایی برای کاهش احتمال شروع سرد در پلتفرم‌های بدون سرور هستیم. در واقع ما به دنبال این هستیم که جلوی رخداد شروع سرد را با هایی بگیریم. از دسته‌بندی‌های کلی در این مبحث می‌وان به روش‌های جلوگیری از شروع سرد با استفاده از پینگ‌گیری یا روش‌های پیش‌بینی شروع سرد، اشاره کرد. روش دیگر برای جلوگیری از اتفاق شروع سرد عبارت‌است از پیش‌بینی شروع سرد. در این مرحله با استفاده از مدل‌های ریاضی و روش‌های یادگیری ماشین می‌توانیم شروه سرد را پیش‌بینی کنیم. توضیحات مربوط به هرکدام در زیربخش مربوطه آمده است.

۱.۳.۳ پینگ‌گیری

در این روش‌ها ما به دنبال کاهش تعدادی شروع‌های سرد در چرخه فراخوانی‌های یک تابع در پلتفرم، با استفاده از ابزارهایی برای جلوگیری از سرد شدن آن تابع هستیم. شیوه عملکرد این ابزارها به این‌گونه است که اقدام به فراخوانی توابع در بازه‌های زمانی مشخص برای جلوگیری از سرد شدن تابع می‌شوند. اگرچه ممکن است این روش ها کارایی خوبی از نظر مصرف منابع نداشته باشند، اما به دلیل ارزانی و سادگی استفاده و همچنین تطابق با پلتفرم‌های حال حاضر، محبوبیت قابل توجهی دارند. بنابراین، با استفاده از ابزارهای آماده‌ای مثل cloudwatch [۱۴] که برای مانیتور کردن سیستم توسعه داده شده یا افزونه‌هایی مثل Lambda Warmer [۱۵] که روی پلتفرم AWS Lamba نصب می‌شود به سادگی شروع‌های سرد را کاهش داد. اتفاقا این روش با توجه به پشتیبانی خوب از پلفترم‌ها و سادگی استفاده در صنعت با اقبال بیشتری نسبت به روش‌های ابتکاری دیگر مواجه شده است. روش‌های ابتکاری بیشتر در حوزه پژوهش برای ایجاد روشی که بعدا در صنعت استفاده شود، هستند. درحالی‌که، عمده مقالات این قسمت در پیاده‌سازی و استفاده‌های صنعتی با این تکنیک است. در ادامه توضیحات بیشتری خواهیم داد.

استفاده از ابزارها

همانگونه که گفتیم، ابزارها نقش جدی در مقابله با شروع سرد در صنعت برای برنامه‌های پیاده‌سازی شده در صنعت را دارند. در مقاله [۱۶] همین موضوع به خوبی بیان شده. برای اینکار مقاله به دنبال پیاده سازی

یک برنامه با معماری monolithic برروی پلتفرم بدون سرور هستیم. بنابرین لازم است از کل اپلیکیشن به عنوان یک میکروسرویس استفاده کنیم و اقدام به راه‌اندازی برنامه کنیم. این موضوع علی‌رغم سربار زیاد و عدم رعایت استانداردهای بهترین تمرین[1]، در عمل قابل انجام است. بنابراین این کار را می‌کنیم. برنامه‌ای که در ابتدا کانتینرایز و سپس مستقر می‌شود یک برنامه‌کاربردی برای محاسبات حجم روان‌آب‌های ناشی از بارش‌های باران در حوزه محصولات محیط زیستی است. این برنامه به این صورت است که یک سری خروجی می‌گیرد و جدول روان آب‌ها در مناطق مختلف آمریکا را محاسبه می‌کند.

مشکلی که با این برنامه‌داریم این است که باید حداقل زمان پاسخ را برای حداقل ۱۰۰ درخواست همزمان داشته باشد. بنابراین لازم است از روش‌هایی مانع سرد شدن این سیستم شد. همچنین در هر لحظه باید حداقل ۱۰۰ کانیتنر گرم‌هم برای رسیدگی به درخواست‌ها داشته باشد. برای این‌کار باید پس از استقرار نرم‌افزار یک اسکریپت برای فراخوانی ۱۰۰ کانتینر به صورت همزمان داشته باشیم. این گونه می‌توان ۱۰۰ بار کاری همزمان داشت.

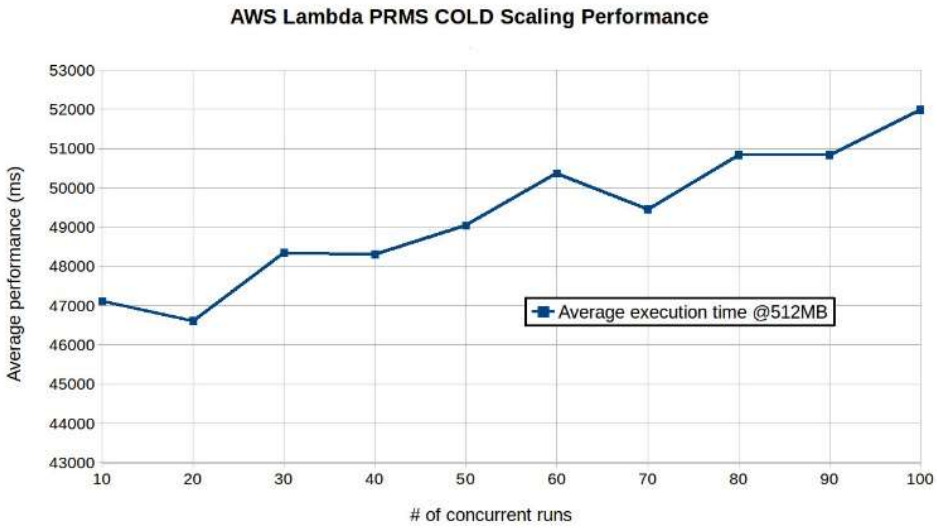
حال باید از چرخه ذوب/یخ زدن[2] جلوگیری کرد تا بارهای کاری ما همیشه زنده باشند. برای این‌کار می‌توان اسکریپتی نوشت که هر چندوقت یکبار اقدام به فراخوانی ۱۰۰ تابع همزمان کند. برای اینکار باید خود سروری که این برنامه‌را اجرا می‌کند همیشه زنده باشد و این یک چالش است اگر تنها بخواهیم از پلتفرم‌های FaaS برای اجرای آن اسکریپت استفاده کنیم.

راه حل بعدی استفاده از ابزارهای یا افزونه‌های[3] مربوطه برای جلوگیری از سرد شدن است. در این مقاله البته از ابزار Cloud watch برای این منظور استفاده شده است. در این مقاله هر ۵ دقیقه نسبت به پینگ‌گیری از توابع برای جلوگیری از سرد شدن اقدام کرده اند.

از جمله نتیجه‌گیری های مقاله همچنین پایین رفتن کارایی پلتفرم بدون سرور در حین اجرای تعداد توابع همزمان است که در شکل ۱۲.۳ نشان داده شده است.

تصویر ۱۲.۳ یکی از مواردی است که حتما باید به آن توجه داشته باشیم. چون باید محاسبه کنیم به ازای تعداد کانتینرهای در حال اجرا باید انتظار چه میزان متوسط زمان پاسخ را داشته باشیم. این خود یکی از موارد

[1] Best Practice
[2] freeze/thaw
[3] extensions



شکل ۱۲.۳: متوسط کارایی (زمان اجرا)، به تعداد کانتنرهای درحال اجرا

مهم در مقایسه با سایر روش‌های اجرای پروژه است.

حال نویسنده این رهیافت را با روش استفاده از VM ها از نظر اقتصادی و زمان اجرا مقایسه کرده است که نتیجه‌ی آن جدول ۳.۳ است.

۲.۳.۳ پیش‌بینی فراخوانی‌ها

در این دسته‌بندی، قرار است مضرات استفاده از سیستم پینگ‌گیری را به حداقل برسانیم. در سیستم پینگ‌گیری با استفاده از فراخوانی در بازه‌های مشخص مانع از اتفاق افتادن شروع سرد می‌شویم ولی از طرفی بازدهی کمی داریم. یعنی اصلا از مزیت ویژگی Scale-to-zero که از ویژگی‌های کلیدی در پلتفرم‌های بدون سرور است استفاده‌ای نکرده ایم. برای پیش‌بینی شروع سرد می‌توانیم از روش‌های ریاضی یا از یادگیری ماشین برای پیش‌بینی شروع سرد استفاده کنیم.

استفاده از مدل‌های ریاضی

در این روش‌ها ما به دنبال استفاده از روش‌های ریاضی برای محاسبه احتمال شروع سرد و به دنبال آن برنامه ریزی برای گریز از آن هستیم. معروفترین روش برای مدل‌سازی این رخدادها استفاده از زنجیره مارکوف است. کتاب [۱۷] به توضیح کامل عملکرد زنجیره مارکوف پرداخته است. از کاربردهای زنجیره مارکوف می‌توان به

جدول ۳.۳: مقایسه هزینه در به ازای استفاده از سیستم بدون سرور با بازه‌های فراخوانی Keep-Alive مختلف در مقایسه با ماشین‌های مجازی

نوع زیرساخت	هزینه کلی در سال	صرفه‌جویی
Lambda + EC2 با بازه‌های زمانی ۳ دقیقه برای KeepAlive	$4496.76	892%
Lambda + EC2 با بازه‌های زمانی ۴ دقیقه برای KeepAlive	$4487.71	893%
Lambda + EC2 با بازه‌های زمانی ۵ دقیقه برای KeepAlive	$4484.00	894%
Lambda + CloudWatch با بازه‌های زمانی ۵ دقیقه برای KeepAlive	$2278.06	1759%
Lambda + CloudWatch با بازه‌های زمانی ۴ دقیقه برای KeepAlive	$2847.57	1407%
سرویس Spot EC2	$12579.84	319%
سرویس On-demand EC2	$40077.00	مبنای پایه

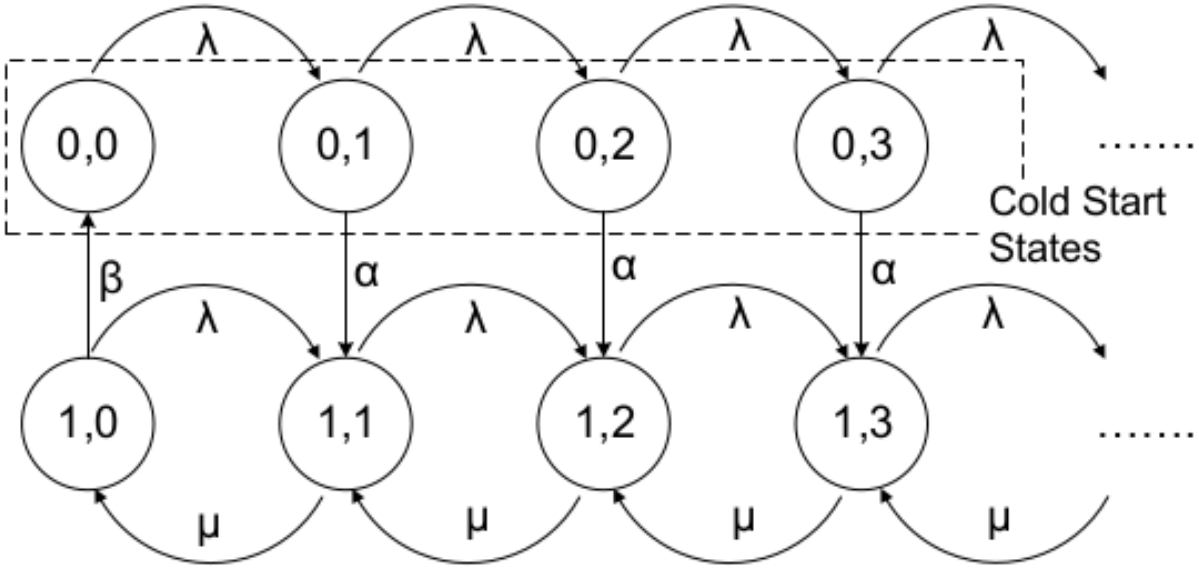
مدلسازی آب‌وهوا و فرآیند زاد و مرگ پرداخت. [۱۷]

در [۱۸] به معرفی روش COCOA برای مدیریت شروع‌های سرد پرداخته شده است. مقاله مدیریت شروع‌های سرد را مثل مدیریت حافظه‌های کش در شبکه‌های توزیع کننده محتوا[1] یا به اختصار CDN می‌بینند. البته این دو تفاوت‌هایی هم باهم دارند. برای مثال،شبکه‌های توزیع کننده محتوا اندازه اشیا[2] مساوری است درحالی که در سیستم‌های بدون سرور اندازه توابع مساوی نیستند.

روش COCOA یک رهیافت تصادفی برای مدل‌سازی تصادفی [3] است که از شبکه‌های صف‌بندی لایه‌ای [4] نیز برای محاسبه زمان پاسخ استفاده می‌کند.

در این مقاله ۲ فاز داریم، در فاز اول به محاسبه‌ی احتمال شروع سرد تابع می‌پردازیم؛ سپس، با درنظر گرفتن شروع سرد آن تابع، زمان پاسخ را با احتمال خوبی محاسبه می‌کنیم. برای محاسبه‌ی شروع سرد از یک مدل از زنجیره‌های مارکوف، M/G/۱/Setup/DelayedOff استفاده می‌کنیم. این مدل خود یک گونه‌ای مدل‌های

[1] Content Delivery Networks
[2] Objects
[3] Stochastic Modeling Approach
[4] Layered Queue Network



شکل ۱۳.۳: ماشین CMTC برای محاسبه اتمال اتفاق افتادن شروع سرد

M/M/k است که در [۱۹] به طور مفصل راجع‌به آن توضیح داده شده است. در اینجا، k برابر است با تعداد سرورهایی که به طور مستقل به اجرای تابع می‌پردازند. در این‌جا، ما تنها یک سرور برای اجرای توابع داریم؛ بنابراین، مقدار k برابر ۱ می‌شود.

مدل CTMC[1] مد نظر مقاله در شکل ۱۳.۳ نشان داده شده است. در این مدل حالت هر گره با دو متغیر i و j نمایش داده شده است.

همانگونه که گفتیم، وضعیت هر گره در CTMC با (i, j) مشخص می‌شود. مقدار i مشخص می‌کند که آیا تابع در داخل حافظه، بارگذاری شده است یا خیر؟ ($i \in \{0, 1\}$). متغیر j هم نشان‌گر تعداد کارهای داخل صف هستند که منتظر هستند داخل صف شوند. ($j \in \{\mathbb{Z}\}$).

هر کدام از انتقال‌ها معانی متفاوتی در زنجیره مارکوف دارند. مثلا انتقال از (i, n) به $(i, n + 1)$، با هزینه λانجام می‌گیرد. انتقال از $(i, n + 1)$ به (i, n) با هزینه μ انجام میگیرد. انتقال از $(1, j)$ به $(0, j)$ نیز هزینه‌ای برابر α دارد. همچنین، یک حالت خاص داریم که انتقال از $(1, 0)$ به $(0, 0)$ است که هنگامی رخ می‌دهد که تابع بیکار سرد شود. این انتقال را هم با نماد β نشان می‌دهیم.

حال چه زمانی متسعد شروع سرد هستیم؟ زمانی که صرف نظر از مقدار j در خانه $(0, j)$ باشیم. در این حالت،

[1] Continues Time Markov Chain

هرگاه درخواستی برسد حتما شروع سرد رخ خواهد داد زیرا باید از سمت)

حال با استفاده از نمادهای بدست آمده به مفاهیمی میرسیم. مقدار $\frac{1}{\lambda}$ میزان زمان رسیدن نوبت به تابع در صف است[1]. مقدار $\frac{1}{\mu}$ یعنی مقدار زمانی که طول می‌کشد یک کار در صف انجام بگیرد و صف یکی به سمت جلو حرکت کند. این مقدار را متوسط زمان سرویس‌دهی می‌گویند[2]. مقدار $\frac{1}{\alpha}$ هم یعنی چقدر طول می‌کشد تا یک ماشین از حالت $(0, j)$ به حالت $(1, j)$ برسد که به این زمان، زمان شروع سرد گفته می‌شود.

حال احتما شروع سرد می‌شود مجموع انتقال‌های ما از حالت سرد به حالت گرم برای خانه‌های با $i = 0$ است. این احتمال با فرمول زیر محاسبه می‌شود.

$$p = \sum_j \pi_{0,j}$$

مقدار π همان احتمال رفتن به خانه‌های $(0, j)$ است.

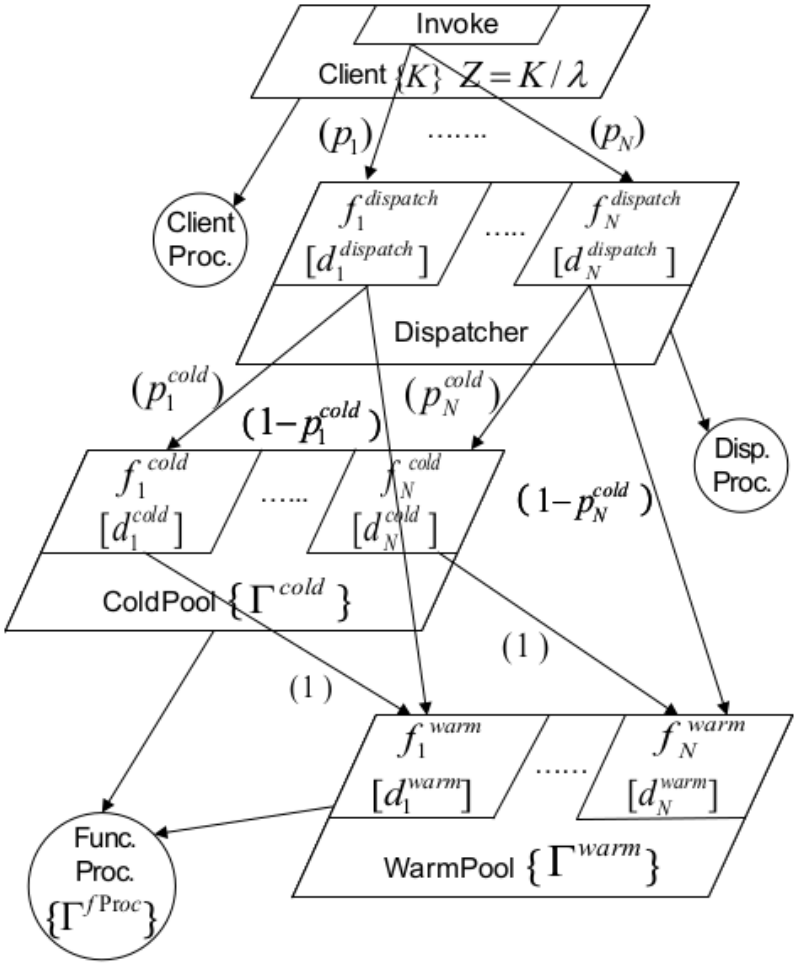
اگرچه حل زنجیره مارکوف بالا، احتمال شروع سرد را به ما می‌دهد؛ اما، ما به دنبال محاسبه زمان پاسخ یک تابع با محاسبه احتمال شروع سرد هستیم. بنابراین، با درنظر گرفتن نتیجه ،p باید زمان پاسخ را محاسبه کنیم. برای محاسبه زمان پاسخ از مدل LQN که در شکل ۱۴.۳ نشان داده شده است، استفاده شده است.

اییک تابع پس از فراخوانی[3]، به قسمت توزیع می‌رود و از آنجا تصمیم گرفته می‌شود به صف توابع سرد اضافه شود یا خیر. هر گاه داخل صف‌های سرد رفت تاخیر ناشی از انتقال کار از صف سرد به صف گرم، تاخیر شروع سرد را موجب می‌شود. در واقع در هر اجرای تابع، حالت استخر گرم[4] حتما اتفاق می‌افتد. این حالت استخر سرد است که با توجه به شرایط ممکن است اتفاق بی‌افتد یا خیر.

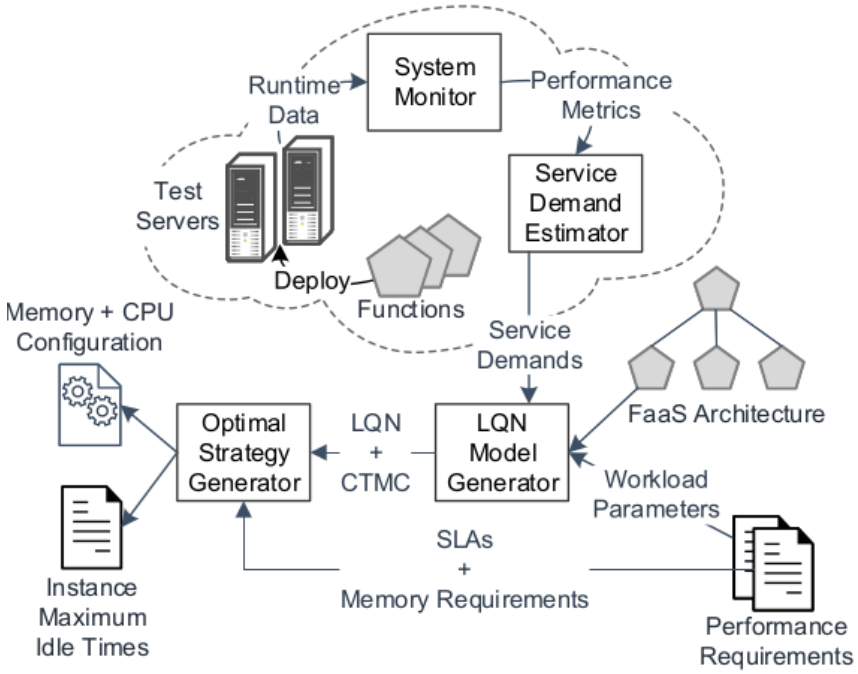
حال برای ارضای توافقات سطح سرویس[5] یا به اختصار SLA لازم است تا با توجه به منابع در اختیار، میزان احتمال شروع سرد و زمان پاسخ که طبق شبکه‌ی لایه‌ای محاسبه کردیم را استخراج کنیم. این همان رهیافت مدل COCOA است. شکل زیر خلاصه این رهیافت را نشان می‌دهد.

با توجه به شکل ۱۵.۳، ورودی‌ها مولد شبکه ،LQN نیازمندی‌های سرویس[6]، معماری ،FaaS و پارامترهای

[1] Inter-Arrival Time
[2] Mean Service Time
[3] Invocation
[4] Warm Pool
[5] Service Level Agreement
[6] Service Demands



شکل ۱۴.۳: مدل LQN برای محاسبه احتمال شروع سرد



شکل ۱۵.۳: خلاصه‌ای از رهیافت COCOA

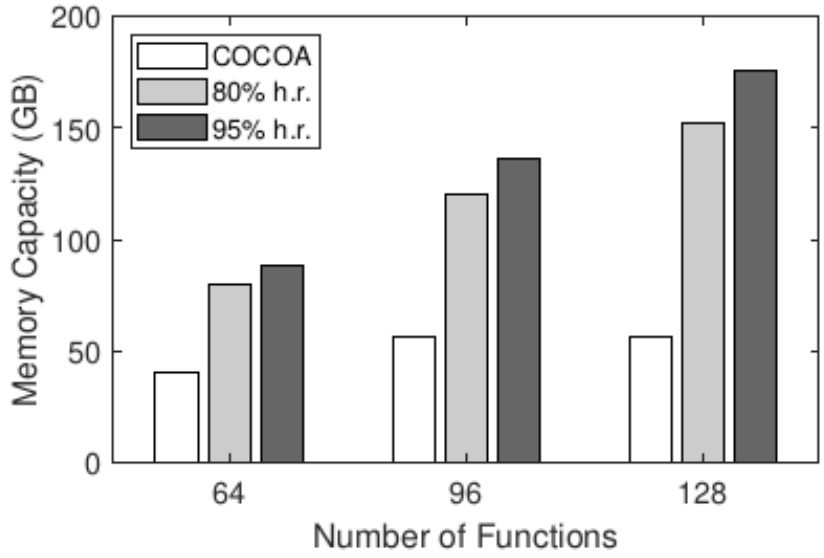
بارهاکاری[1]، هستند. خروجی این مرحله یک مولد LQN برای سیستم است. حال ترکیب این شبکه با زنجیره مارکوف مربوط به تابع، ما را به مرحله تولید استراتژی بهینه برای تولید استراتژی مناسب می‌برد. تولید کننده استراتژی بهینه[2] نیز با توجه به مقدار توافق‌های سطح سرویس ونیازمندی‌های سیستم، محاسبه می‌شود.

حال برای مقایسه این روش با Fixed Alive-Time سه عامل ارضای SLA، مقدار مصرف حافظه و مصرف ظرفیت محاسبه می‌شود. شکل زیر میزان مصرف حافظه را در مقایسه با زمان ثابت Hit Rateهای مختلف را نشان می‌دهد.

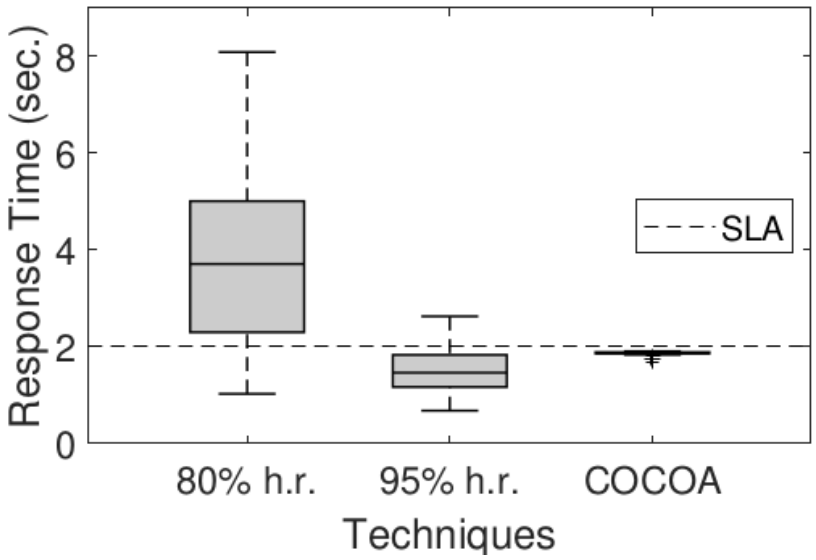
در شکل ۱۶.۳، مقدار حافظه مصرفی در مقایسه با حالت‌های استفاده از حافظه کش با hit rate های بالا بسیار بهتر است. همچنین در شکل زیر میزان زمان پاسخ بر حسب هایSLA مختلف،مورد بحث قرار می‌گیرد.

همانگونه که در شکل ۱۷.۳ مشخص است، استفاده از COCOA بسیار بهتر از مقدارهای Hit rate بالا توانسته است توافق‌های سطح سرویس مانند حداکثر زمان پاسخ را ارضا کند.

[1] Workload Parameters
[2] Optimal Strategy Generator



شکل ۱۶.۳: مقدار پیش‌بینی شده برای مصرف حافظه براساس



شکل ۱۷.۳: مقایسه ارضای زمان SLA با تکنیک COCOA یا استفاده از HitRate

استفاده از رهیافت‌های یادگیری ماشین

روش دیگر برای پیش‌بینی شروع سرد، استفاده از رهیافت‌های مبتنی بر یادگیری ماشین است. این موضوع در مقاله x مورد بررسی واقع شده است. البته این مقاله تنها به کاربرد استفاده از روش‌های یادگیری ماشین برای جلوگیری از اتفاق افتادن شروع سرد بسنده نمی‌کند بلکه رویکرد اصلی این مقاله - یا بهتر است بگوییم این دسته از مقاله‌ها -، استفاده از روش‌های یادگیری ماشین در کنار سایر روش‌ها برای این موضوع است. در مقاله [۴] از مدل Arima در کنار روش‌های دیگر به عنوان یک روش مکمل استفاده شده است تا به بازدهی بهتری برای مدیریت شروع‌های سرد برسیم.

ابتدای مقاله، تقسیم بندی‌هایی برای فعالسازهای توابع[1] داریم. توابع به ۷ دسته اصلی تقسیم می‌شوند که عبارتند از:

۱. HTTP
۲. درخواست‌های صف[2]
۳. رخداد[3]
۴. درخواست‌های Orchestration
۵. زمانبند[4]
۶. حافظه[5]
۷. سایر درخواست‌ها

در شکل ۱۸.۳ نسبت تعداد توابع و نسبت فراخوانی‌ها در پلتفرم Azure آمده است:

[1] Triggers
[2] Queue
[3] Event
[4] Timer

[5] درخواست‌هایی مثل ارتباط با پایگاه داده در ذیل آن قرار می‌گیرد.

Trigger	%Functions	%Invocations
HTTP	55.0	35.9
Queue	15.2	33.5
Event	2.2	24.7
Orchestration	6.9	2.3
Timer	15.6	2.0
Storage	2.8	0.7
Others	2.2	1.0

شکل ۱۸.۳: تعداد و میزان فراخوانی فعال‌سازها در پلتفرم Azure

پرسشی که مطرح می‌شود این است که آیا برنامه‌ها تنها از یک نوع فعال‌ساز استفاده می‌کنند؟ جواب منطقا خیر است. برنامه‌های کاربری قاعدتا از انواع ترکیبات ممکن است استفاده کنند. مثلا می‌دانیم یک برنامه‌کاربردی ساده تحت وب حتما از ترکیب حافظه[1] و فراخوانی‌های HTTP حتما استفاده می‌کند. بنابراین نمی‌توان چنداد تفکیک قائل شد. درصد ترکیبات فعال‌سازها در شکل ۱۹.۳ نشان داده شده است.

آمارهایی برای درصد استفاده از توابع داریم که برای درک اهمیت شروع سرد مفید است. مثلا در ٪۷۵ از توابع، زمان اجرای برنامه زیر ۱۰ ثانیه است. بنابراین شروع سرد وقوع شروع سرد زمان اجرا را ممکن است تا ٪۲۰۰ هم بالا ببرد. حدود ٪۸۱ توابع حداکثر ۱ فراخوانی در دقیقه را ثبت می‌کنند یعنی اینکه زمان زنده‌ماندن[2] بالا برای این توابع اصلا به صرفه نیست. نکته جالب این است که کمتر از ٪۲۰ توابع مسئول فراخوانی ۹۹.٪۶ از توابع هستند.

مقاله به دنبال یک سیاست‌گذاری خودتطبیق برای سازگاری با شرایط است که برحسب آن زمان زنده بودن تابع تفاوت کند. برای تنظیم سیاست خودتطبیق از فلوچارت شکل ؟؟ برای سیاست‌گذاری پیروی می‌کند.

بر اساس شکل ۲۰.۳ سه سیاست اصلی برای مواجهه با شروع سرد داریم

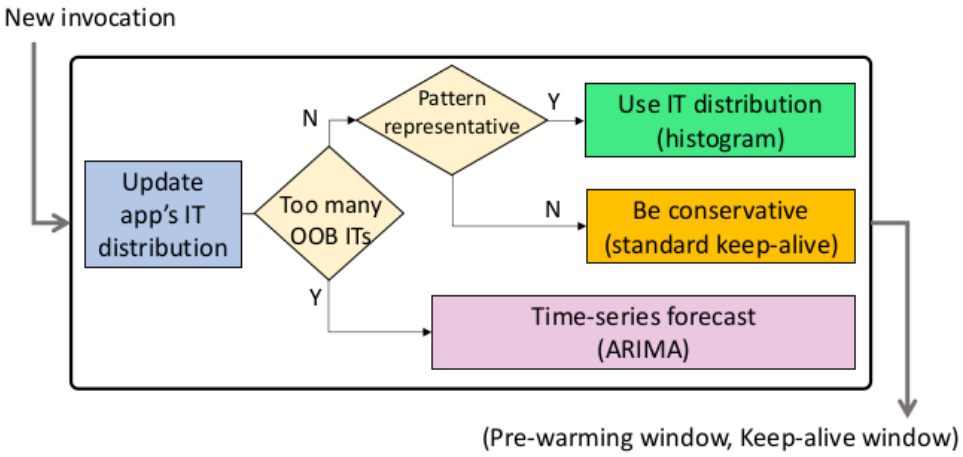
۱. برنامه خطای خارج از محدودیت[3] نمی‌دهد.

[1] Storage
[2] Keep-Alive
[3] Out of Bound

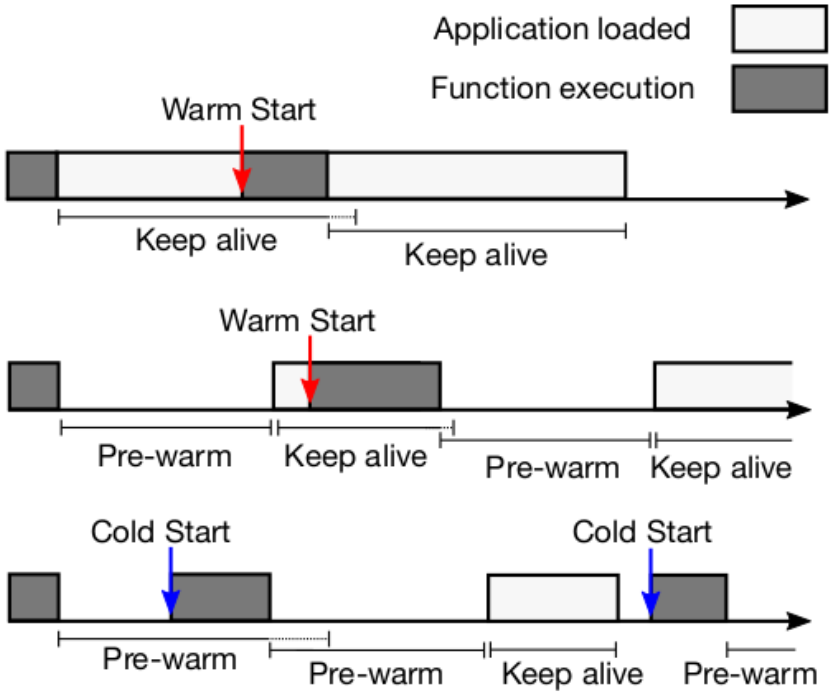
Trigger Type	% Apps
HTTP (H)	64.07
Timer (T)	29.15
Queue (Q)	23.70
Storage (S)	6.83
Event (E)	5.79
Orchestration (O)	3.09
Others (o)	6.28

Trigger Types	Fraction of Apps (%)	Cum. Frac. (%)
H	43.27	43.27
T	13.36	56.63
Q	9.47	66.10
HT	4.59	70.69
HQ	4.22	74.92
E	3.01	77.92
S	2.80	80.73
TQ	2.57	83.30
HTQ	2.48	85.78
Ho	1.69	87.48
HS	1.05	88.53
HO	1.03	89.56

شکل ۱۹.۳: نسبت ترکیب فعال‌سازها در برنامه‌های کاربردی



شکل ۲۰.۳: تنظیم سیاست‌گذاری



شکل ۲۱.۳: ترکیب Pre-Warm و keep-alive برای پیاده سازی شروع سرد برای الگوی شناسایی شده

(آ) الگوی برنامه قابل شناسایی است.

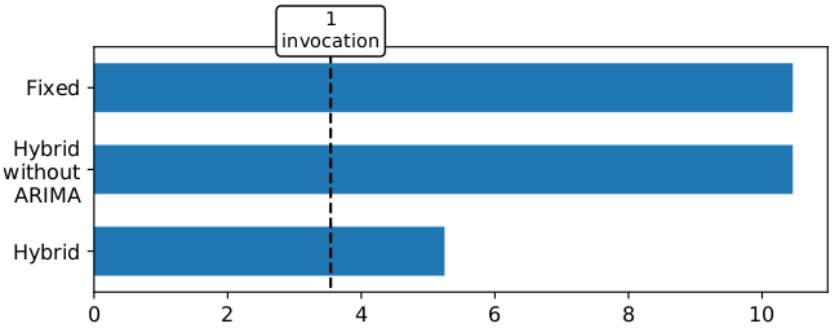
در این حالت از روش هیستوگرام استفاده می‌کنیم. در این حالت دو متغیر زمانی Pre-Warm و Keep-Alive را تعریف می‌کنیم. وظیفه متغیر Prewarm این است که بلافاصله پس از اجرای تابع آن را به مدت زمانی مقرر شده سرد کند. این متغیر برای توابعی کاربرد دارد که الگوی فراخوانی آن‌ها به گونه‌ای است که تابع پس از فراخوانی تا مدتی را بلا استفاده می‌ماند. کاربرد متغیر Keep-Alive هم در این است که تابع پس از اتمام Pre-warm که مجددا گرم می‌شود تا چه زمانی گرم بماند.

ترکیب این دو در تصویر ۲۱.۳ به خوبی نشان داده شده است.

نکته ای که باید به آن توجه داشت این است که Pre-warm را برای صرفه‌جویی در مصرف منابع پیاده سازی کرده‌ایم و حتی ممکن است باعث شروع‌سردهای بیشتری هم بشود.

(ب) الگوی برنامه قابل شناسایی نیست.

در این حالت از روش fixed-alive-time استفاده می‌شود که همان روش سنتی برای مقابله با شروع سرد است.



شکل ۲۲.۳: مقایسه استفاده از ۳ سیاست‌گذاری مقابله با شروع سرد

۲. برنامه خطای خارج از محدودیت می‌دهد.

در این روش از روش پیش‌بینی فراخوانی براساس مدل ARIMA استفاده می‌شود. توضیحات نحوه عملکرد این مدل در [۲۰] ذکر شده است.

نتیجه استفاده از این روش در شکل ۲۲.۳ آمده است.

همانگونه که در شکل ۲۲.۳ می‌بینیم، استفاده از روش هیبریدی درصد شروع‌های سرد را به زیر ٪۶ رسانده است در حالی‌که در این آمار اولین شروع سرد که در حدود ٪۸.۳ از شروع‌های سرد را محاسبه می‌کند آمده است. در حالی که در دو سیاست دیگر میزان اتفاق افتادن شروع‌های سرد بالای ٪۱۰ است پس از این نظر سیاست‌گذاری هیبریدی برای مدیریت شروع‌های سرد بسیار موثر است.

فصل ۴

نتیجه‌گیری

۱.۴ ایده و محدوده‌ی کاری در آینده

با مطالعه در ادبیات موضوع۲ و کارهای انجام۳ شده فهمیدیم که مشکل شروع سرد مشکل گسترده‌ای است و محدود به حوزه‌ی خاصی نمی‌شود. همانگونه که مطرح شد عواملی مثل حجم حافظه مصرفی تابع، سربار تابع، نوع کانفیگ سرور، مشخصات سخت‌افزاری سرور زیرساخت، وضعیت شبکه، زبان برنامه‌نویسی‌ای که تابع با آن نوشته شده و ... در تاخیر شروع سرد بسیار موثر و تاثیرگذار هستند. البته در این گزارش از بی روش‌های کنترل شروع سرد بر روی ویژگی‌های شبکه برای کاهش مدت زمان آماده‌سازی و روش‌های پیشگیری از شروع سرد توقف کردیم. متاسفانه مدت زمان کافی برای مطالعه سایر روش‌ها فراهم نشد.

روش‌های پیشگیری از شروع سرد، اگر چه در سناریوی موفق خود باعث به صفر رسیدن تاخیر شروع سرد می‌شوند، اما در سناریوهای ناموفق خود تاخیر شروع سرد به طور عادی برای تابع اتفاق می‌افتد. نکته‌ی منفی درمورد اکثر این روش‌های این است که علاوه‌بر دقت پایین یا خطاهای بسیار برای شناسایی الگوی شروه سرد، - که کار بسیار دشواری هم هست، - این روش‌ها سربار محاسباتی بسیار بالایی برای پیش‌بینی دارند درحالی‌که وقت ما برای پیش بینی بسیار محدود است. از طرف دیگر، پیچیدگی اجرا و مدت‌زمان سربار محاسباتی برای روش‌های مبتنی بر کاهش زمان شروع سرد بسیار کمتر است. شاید به همین دلیل است که بیشتر در صنعت و پژوهش‌های عملی مورد استقبال قرار گرفته اند. از طرفی هم این روش ها مانع از تاخیر شروع سرد نمی‌شوند و نمی‌توانند به طور کامل جایگزین روش پیشگیری از شروع سرد شوند.

به نظر می‌رسد ترکیب این دو روش به عنوان کار آینده گزینه جذابی باشد. مثلا از راهکارهایی برای فرار از شروع سرد استفاده کرد. در کنار این قضیه، از راهکارهایی نیز برای جلوگیری از شروع سرد استفاده کنیم. در بهترین حالت، علاوه بر اینکه زمان پاسخ را کم کرده‌ایم توانسته ایم با دقت خوبی هم از به وقوع پیوستن شروع سرد نیز جلوگیری کنیم. اما از طرفی ممکن است باعث بیش‌مهندسی[1] شدن قضیه بشویم. این موضوع نیازمند بررسی بیشتر است و نیاز به تحقیق بیشتری برای قطعی شدن موضوع داریم.

در قدم پیشرفته‌تر می‌توان از این کار برای به حداقل رسانی شروع سرد در ترکیبات توابع استفاده کرد. این موضوع برای به حداقل رسانی توابعی که در Orchestrator ترکیب شده‌اند بسیار مفید می‌تواند باشد. فرض کنید

[1] Over Engineering

۲ تابع داریم که با الگوی زنجیره‌ای[1] به هم وصل شده‌اند. پس با فراخوانی تابع اول می‌دانیم تابع دوم هم حتما فراخوانی می‌شود. پس باید قبل از فراخوانی، تابع گرم شود. برای این کار پژوهش‌های انجام شده در [۲، ۳] می‌تواند بسیار کمک کننده باشد.

اگر قرار به استفاده از دیتاستی باشد تنها دیتاست در دسترس در [۲۱] وجود دارد که اطلاعات مصرف و فراخوانی توابع در سال‌های ۲۰۱۹ و ۲۰۲۰ را به صورت متن‌باز منتشر کرده‌است. برای این کار مجبور هستیم اقدام به پیاده‌سازی پلتفرم مد نظر خود کنیم. برای این کار احتمالا مجبور شویم کدپروژه پلتفرمی را که می‌خواهیم تغییر دهیم را برای سازگاری با سیاست‌های خودمان تغییر دهیم. نتیجه این تغییرات پلتفرم جدیدی بر مبنای پلتفرم پایه خواهد بود که آن را در آینده ارائه خواهیم داد.

۲.۴ نتیجه‌گیری کلی

در فصل ۱ به بیان مقدمه‌ای از موضوع و در فصل ۲ به بیان ادبیات موضوع پرداختیم تا خواننده را با موضوعاتی که قرار است در پژوهش در باره آن بحث کنیم آشنا سازیم. پس از آشنایی با مباحث پایه‌ای به بیان کارهای انجام شده و ایده‌هایی که تاکنون در این حوزه مطرح شده است، در فصل ۳ پرداختیم.

به طور خلاصه برای مدیریت شروع سرد دو سیاست کلی می‌توان داشت. سیاست اول این است که زمان آماده‌سازی برنامه را کم کنیم تا مدت زمان تاخیر به حداقل برسد. یعنی در واقع، ما اجازه وقوع شروع سرد را به تابع می‌دهیم ولی مدت زمان ناشی از تغییر را حداقل کرده‌ایم. سیاست دوم این است که مانع وقوع شروع سرد بشویم. هر دو سیاست از موضوعات ترند و پرپژوهش در این حوزه هستند.

همانگونه که می‌دانید روش پایه‌ای برای مدیریت شروع سرد، روش زمان ثابت زنده ماندن توابع[2] است. در این روش، تابع تا مدت زمان ثابتی پس از آخرین فراخوانی گرم می‌ماند و اگر در آن بازه صدا زده نشود سرد می‌شد. این روش بسیر ابتدایی است و هرگز نمی‌توانست الگوهای فراخوانی را تشخصی دهد و در فراخوانی‌های متناوب در بازه‌های طولانی‌تر از بازه Keep Alive، همواره شروع سرد را تجربه می‌کردیم. اگرچه در مصرف منابع هم به نسبت بازدهی خوبی نداریم. بنابراین این روش در مقایسه با روش‌هایی که به دنبال مدل‌سازی برای گرم کردن تابع

[1] Chaining
[2] Fixed-Time alive

هستند نه بازدهی خوبی از نظر منابع دارد و نه توانسته است از نظر مدت‌زمان پاسخ[1] پاسخ خوبی داشته باشد.

از طرفی روش‌های بهینه‌سازی سرور وجود دارند که اگرچه مدت زمان پاسخ با درنظر گرفتن شروع سرد را کاهش می‌دهند اما خود باید مراقب سربارهای احتمالی مانند سایز حافظه کش، سایز استخر گرم و ... باشند تا بتوان گفت از نظر مصرف منابع بسیار بهینه عمل می‌کنند. این روشّا اگر چه مدت زمان را کم می‌کند اما همچنان از تاخیر شروع سرد رنج می‌بریم. همچنین مدت‌زمانی که زمان پاسخ برمی‌گردد آن‌قدر کاهش نمی‌یابد که بگوییم مشکل زمان پاسخ حل شده است.

حال ایده‌ای که مطرح می‌شود این است که آیا با ترکیب این روش‌ها می‌توان به روش بهینه‌ای برای مدیریت شروع‌های سرد رسید یا خیر؟ این سوال اصلی این پژوهش است و برای پاسخ به آن به مطالعه‌ی خیلی بیشتری نیاز است. اما اگر بتوانیم با استفاده از مدل پیشگیری مناسب مانع اتفاق افتادن شروع‌های سرد شویم و از ظرف دیگر با استفاده از یک تکنیک بهینه مدت زمان لود شدن تابع را حتی در صورت اتفاق افتادن شروع سرد کاهش بدهیم مسئله را با دقت بسیار خوبی حل کرده ایم.

البته باید توجه داشت این یک کار بسیار چالش برانگیز است. در [۱۸] ماباید برای هر تابع بتوانیم زنجیره مارکوف آن را رسم کرده و با ترکیب با LQN به پیش‌بینی بهینه بر اساس توافقات سطح سرویس شویم. اینکار بسیار طولانی و چالشی است؛ اگرچه که دقت خوبی دارد و به خوبی توانسته است سطح انتظارات را برآورده سازد. یا مثلا در [۴] ممکن است حتی مسئله را اصطلاحا بیش مهندسی کرده باشیم.

ما همواره در مهندسی نرم‌افزار و شبکه به دنبال کم کردن تاخیر ناشی از ارتباطات در شبکه و سرور با کاربران هسنیم. پس می‌توان گفت تا کنون به مبحث شروع سرد بسیار بی‌توجهی شده است. موضوع دیگری که در این بیین بسیار مغوف مانده است، امکان کاهش شروع سرد برای رکیبات توابع است که پاسخی برای این موضوع مشاهده نشده است.

[1] Response Time

مراجع

[1] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Commun. ACM*, vol.62, p.44-54, Nov. 2019.

[2] P.-M. Lin and A. Glikson, "Mitigating cold starts in serverless platforms: A pool-based approach," *arXiv preprint arXiv:1903.12221*, 2019.

[3] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, "Agile cold starts for scalable serverless," in *11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[4] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, pp.205-218, 2020.

[5] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, *et al.*, "Serverless computing: Current trends and open problems," in *Research Advances in Cloud Computing*, pp.1-20, Springer, 2017.

[6] A. AWS, "Aws lambda functions," `https://aws.amazon.com/lambda/`.

[7] G. INC., "Google cloud fucntions," `https://cloud.google.com/functions`.

[8] Microsoft, "Microsoft azure platform," `https://azure.microsoft.com/en-us/services/functions/`.

[9] A. openwhisk corporation, "Ibm openwhisk platform," `https://openwhisk.apache.org/`.

[10] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with openlambda," in *8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[11] U. Bharti, D. Bajaj, A. Goel, and S. Gupta, "Sequential workflow in production serverless faas orchestration platform," in *Proceedings of International Conference on Intelligent Computing, Information and Control Systems*, pp.681-693, Springer, 2021.

[12] M. Luksa. *Kubernetes in Action*. 2018.

[13] S. K. Jeff Nickoloff. *Docker In Action*. 2019.

[14] C. Watch, "Cloud watch monitoring tool," `https://aws.amazon.com/cloudwatch/`.

[15] L. warmer extension, "Lambda warmer extension to mitigate cold start in aws lambda functions," `https://github.com/jeremydaly/lambda-warmer`.

[16] W. Lloyd, M. Vu, B. Zhang, O. David, and G. Leavesley, "Improving application migration to serverless computing platforms: Latency mitigation with keep-alive workloads," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pp.195-200, IEEE, 2018.

[17] S. Brooks, A. Gelman, G. Jones, and X.-L. Meng. *Handbook of markov chain monte carlo*. CRC press, 2011.

[18] A. U. Gias and G. Casale, "Cocoa: Cold start aware capacity planning for function-as-a-service platforms," in *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp.1-8, IEEE, 2020.

[19] A. Gandhi, S. Doroudi, M. Harchol-Balter, and A. Scheller-Wolf, "Exact analysis of the m/m/k/setup class of markov chains via recursive renewal reward," in *Proceedings of the ACM SIGMETRICS/international conference on Measurement and modeling of computer systems*, pp.153-166, 2013.

[20] Arima, "How to create an arima model for time series forecasting in python," `https://machinelearningmastery.com/arima-for-time-series-forecasting-with-python/`.

[21] A. public dataset, "Microsoft co.," `https://github.com/Azure/AzurePublicDataset/blob/master/AzureFunctionsDataset2019.md`.