مقاله پژوهشی

یک چارچوب ریزدانه و آگاه از محل غیرمتمرکز برای مدیریت حافظه نهان در رایانش بدون سرور

محمد کاهانی، سعید ابریشمی و عادل نجاران طوسی

چکیده: امروزه استفاده از برنامه‌های مبتنی بر رایانش بدون سرور و تابع به‌عنوان سرویس به‌طور فزاینده‌ای گسترش یافته است. این رویکرد به کاربران اجازه می‌دهد بدون نیاز به پیکربندی میزبان‌های زیرساختی، برنامه‌های خود را مستقر کرده و از مزایایی همچون مدل هزینه مبتنی بر مصرف، انعطاف‌پذیری و مقیاس‌پذیری خودکار سکوهای بدون سرور بهره‌مند شوند. با این حال، به دلیل ماهیت بدون‌حالت توابع بدون سرور، در سناریوهایی که توابع نیازمند تعامل با یکدیگر هستند یا دسترسی به داده‌های حجیم مطرح است، کارآیی سیستم با محدودیت مواجه می‌شود. در رویکردهای متداول، برای رفع این محدودیت از انباره‌های داده راه دور مانند Amazon S۳ استفاده می‌شود که سربار زمانی قابل توجهی ایجاد می‌کنند. یکی از راهکارهای کاهش این سربار، به‌کارگیری حافظه نهان است. اگرچه مطالعاتی در این زمینه انجام شده است، اما این پژوهش‌ها عموماً حافظه نهان را در سطح میزبان یا در سطح کل برنامه کاربردی مدیریت کرده‌اند که از کارایی مطلوب برخوردار نیست. در این مقاله، با ارائه چارچوبی جدید در بستر متن‌باز بدون سرور، یک سامانه ریزدانه مدیریت حافظه نهان در سطح هر تابع و مبتنی بر آگاهی از محلیت پیشنهاد شده است. این رویکرد ضمن کاهش زمان پاسخ توابع، امکان بهره‌برداری بهینه‌تر از منابع موجود را نیز فراهم می‌سازد.

کلیدواژه: **رایانش بدون سرور، تابع به عنوان خدمت، توابع حالتمند، حافظه نهان در سطح تابع، توابع مبدأ.**

۱- مقدمه

رایانش بدون سرور[1] به‌عنوان یکی از الگوهای نوین رایانش ابری در سال‌های اخیر مطرح شده و به‌تدریج جایگزین بسیاری از تعاملات سنتی کاربران با سرویس‌هایی نظیر زیرساخت به عنوان سرویس[2] و سکو به عنوان سرویس[3] شده است. در این مدل، کاربران بدون نیاز به پیکربندی و مدیریت مستقیم منابع موردنیاز برنامه‌های خود، آن‌ها را در قالب مجموعه‌ای از توابع بر روی سکوهای ارائه‌دهنده مستقر می‌کنند. از این رو، این مدل با عنوان تابع به عنوان سرویس[4] نیز شناخته می‌شود [۱]. در حال حاضر سرویس‌دهندگان متعددی چنین قابلیتی را در اختیار کاربران قرار می‌دهند که از مهمترین آنها می‌توان به AWS Lambda، Microsoft Azure Functions و Google Cloud Functions اشاره کرد.

از ویژگی‌های اصلی رایانش بدون‌سرور می‌توان به پرداخت مبتنی بر میزان مصرف، مقیاس‌پذیری خودکار، و جداسازی محاسبات از سامانه‌های ذخیره‌سازی اشاره کرد [۲]. هرچند این ویژگی‌ها مزایای چشمگیری به همراه دارند، اما چالش‌های تازه‌ای را نیز ایجاد می‌کنند که زمینه‌ساز شکل‌گیری حوزه‌های نوین پژوهشی در این زمینه شده است.

مقیاس‌پذیری خودکار یکی از قابلیت‌های کلیدی در رایانش بدون‌سرور است که با توجه به میزان تقاضا و مصرف منابع، از مقیاس‌پذیری افقی یا عمودی برای مدیریت نمونه‌های توابع استفاده می‌کند. در مقیاس‌پذیری افقی، نسخه‌های جدیدی از یک تابع ایجاد یا نسخه‌های بلااستفاده آزاد می‌شوند. در مقابل، مقیاس‌پذیری عمودی با افزایش یا کاهش منابع تخصیص‌یافته (مانند حافظه یا هسته‌های پردازشی) به نمونه‌های موجود صورت می‌گیرد. معمولاً پس از اتمام اجرای یک تابع، نمونه مربوطه و منابع تخصیص‌یافته به آن برای مدتی در حافظه اصلی باقی می‌مانند تا در صورت ورود درخواست‌های جدید، پاسخ‌گویی سریع‌تری فراهم شود. با این حال، اگر طی این مدت درخواستی برای تابع مذکور دریافت نشود، منابع آن به‌طور خودکار آزاد می‌گردند؛ وضعیتی که به مقیاس‌پذیری تا صفر[5] شهرت دارد. اگرچه این ویژگی موجب بهینه‌سازی مصرف منابع می‌شود، اما پیامدی به نام شروع سرد[6] را به همراه دارد که خود به‌عنوان یکی از چالش‌های اساسی رایانش بدون‌سرور شناخته می‌شود.

جداسازی سرویس محاسبات از سرویس ذخیره‌سازی، ویژگی ارزشمند دیگری در رایانش بدون‌سرور است که امکان مقیاس‌پذیری خودکار برای بارهای کاری غیرقابل پیش‌بینی را فراهم می‌کند [۲]. با این حال، این ویژگی توسعه‌دهندگان را ملزم می‌سازد که از توابع بدون‌حالت[7] استفاده کنند؛ توابعی که امکان نگهداری داده میان فراخوانی‌های متوالی را ندارند. در مقابل، برخی توابع ذاتاً حالتمند[8] هستند و نیاز دارند داده‌هایی را برای فراخوانی‌های بعدی ذخیره کنند. علاوه بر این، بسیاری از برنامه‌ها از مجموعه‌ای از توابع تشکیل شده‌اند که در قالب ساختارهایی مانند گراف‌های جهت‌دار بدون دور یا خط لوله با یکدیگر تعامل دارند [۳]. از

این مقاله در تاریخ ۲۶ اسفند ماه ۱۴۰۳ دریافت و در تاریخ ۴ مهر ماه ۱۴۰۴ بازنگری شد.

محمد کاهانی، گروه مهندسی کامپیوتر، دانشکده مهندسی، دانشگاه فردوسی مشهد، مشهد، ایران، (email: kahani@mail.um.ac.ir).

سعید ابریشمی (نویسنده مسئول)، گروه مهندسی کامپیوتر، دانشکده مهندسی، دانشگاه فردوسی مشهد، مشهد، ایران، (email: s-abrishami@um.ac.ir).

عادل نجاران طوسی، دانشکده سیستمهای محاسباتی و اطلاعاتی، دانشگاه ملبورن، ملبورن، استرالیا (email: adel.toosi@unimelb.edu).

1. Serverless Computing
2. Infrastructure as a Service
3. Platform as a Service
4. Function as a Service (FaaS)
5. Scale to Zero
6. Cold Start
7. Stateless
8. Stateful

آنجا که سکوی تابع به‌عنوان سرویس به‌طور مستقیم ارتباط میان توابع را تسهیل نمی‌کند، توابع ناگزیرند نتایج میانی خود را در انباره‌های داده خارجی نظیر Amazon S۳ [۴] یا DynamoDB ذخیره کنند. این امر باعث ایجاد سربار اضافی شده و اجرای برنامه‌های حالتمند را کند می‌کند.

به‌منظور پشتیبانی از توابع حالتمند در محیط‌های بدون‌سرور، سکوهای متعددی طراحی شده‌اند که از مهم‌ترین آن‌ها می‌توان به Cloudstate [۵]، StateFun [۶]، Beldi [۷]، Cloudburst [۸]، Boki [۹] و Crucial [۱۰] اشاره کرد. هر یک از این سکوها راهکارهای متفاوتی برای کاهش سربار ناشی از ذخیره‌سازی حالت در انباره‌های داده راه‌دور به‌کار می‌گیرند. یکی از متداول‌ترین این رویکردها استفاده از حافظه نهان[1] است؛ بدین صورت که داده‌های پرتکرار یا پرکاربرد در حافظه محلی ذخیره می‌شوند تا هزینه دسترسی به انباره‌های راه‌دور کاهش یابد.

پژوهش‌های متعددی از حافظه نهان برای پشتیبانی از توابع حالتمند در سیستم‌های بدون‌سرور بهره برده‌اند [۱۱] تا [۱۳]. با این حال، در هیچ یک از کارهای پیشین مدیریت حافظه نهان در سطح تابع صورت نگرفته است. رویکردهای موجود عمدتاً مدیریت حافظه نهان را در سطح ماشین میزبان، فضای داده اشتراکی یا مرکزی، و در برخی موارد در سطح برنامه انجام داده‌اند. این شیوه‌ها فاقد ریزدانگی لازم برای مدیریت بهینه حافظه نهان بر اساس الگوی واقعی درخواست‌ها هستند. برای نمونه، هر برنامه معمولاً از یک یا چند تابع تشکیل شده است که هدفی مشترک را دنبال می‌کنند. با این وجود، برخی توابع عمومی ممکن است در چندین برنامه به‌کار گرفته شوند. در چنین شرایطی نرخ درخواست این توابع و میزان استفاده آن‌ها از حافظه نهان با الگوی کل برنامه تفاوت دارد. بنابراین، مدیریت حافظه نهان در سطح برنامه از کارایی کافی برخوردار نیست و نیاز به رویکردهای ریزدانه‌تر در سطح تابع احساس می‌شود.

در این پژوهش، یک چارچوب ریزدانه و غیرمتمرکز برای مدیریت حافظه نهان در سطح تابع ارائه شده است که بر بستر متن‌باز کوبرنتیز [۱۴] پیاده سازی شده است. این چارچوب که Kacheless نامیده می‌شود (بر گرفته از واژگان Kubernetes، Cache و Serverless) مفهوم توابع مبدا[2] را برای مدیریت حافظه نهان مشترک میان توابع مستقر بر روی یک یک سرور معرفی می‌کند. افزون بر این، الگوریتم‌های جای‌گذاری توابع و زمان‌بندی درخواست‌ها در محیط کوبرنتیز بازطراحی شده‌اند تا ملاحظات مربوط به محلیت داده‌ها و توزیع متوازن بار میان توابع مبدأ نیز مد نظر قرار گیرد.

ساختار مقاله به شرح زیر است: در بخش ۲ مروری بر کارهای پیشین ارائه می‌شود. بخش ۳ به معرفی چارچوب پیشنهادی اختصاص دارد. بخش ۴ شامل آزمایش‌ها و نتایج حاصل از آن‌ها است و در نهایت، بخش ۵ به نتیجه‌گیری و ترسیم مسیرهای پژوهشی آینده می‌پردازد.

۲- کارهای پیشین

یکی از نخستین سیستم‌های ارائه‌شده در زمینه مدیریت حافظه نهان داده‌ای، Pocket است که با هدف رفع چالش‌های سیستم‌های بدون‌سرور در برنامه‌های تحلیل داده، نظیر الگوریتم‌های نگاشت- کاهش، توسعه یافته است [۱۵] Pocket در حقیقت یک انباره داده توزیع‌شده و موقتی است که به‌جای استفاده از انباره‌های داده راه‌دور مانند S۳ ـ که دسترسی به آن‌ها کند و پرهزینه است ـ از مجموعه‌ای از ماشین‌های مجازی به‌عنوان انباره داده بهره می‌گیرد. این سیستم برای بهینه‌سازی هزینه، از سه نوع ذخیره‌سازی مختلف شامل دیسک مغناطیسی، دیسک‌های پرسرعت و DRAM استفاده می‌کند.

سریکانتی و همکاران [۸]، نیز در طراحی سیستم Cloudburst برای پشتیبانی از توابع حالتمند، از حافظه نهان بهره گرفتند. در این سیستم، ترکیبی از حافظه‌های نهان محلی در سطح ماشین مجازی همراه با یک سامانه ذخیره‌سازی کلید- مقدار به نام Anna به‌کار گرفته شده است. برای تضمین سازگاری داده‌ها در حافظه نهان، چندین پروتکل سازگاری نیز پیشنهاد گردید. ایشان در ادامه و در پژوهش بعدی خود، سیستم [۱۶] HydroCache را معرفی کردند که مسئله اجرای توابع مرتبط با یک برنامه کاربردی بر روی ماشین‌های مختلف را بررسی می‌کرد. در این کار، پروتکل‌هایی برای تضمین سازگاری علّی تراکنشی میان چندین ماشین[3] ارائه شد.

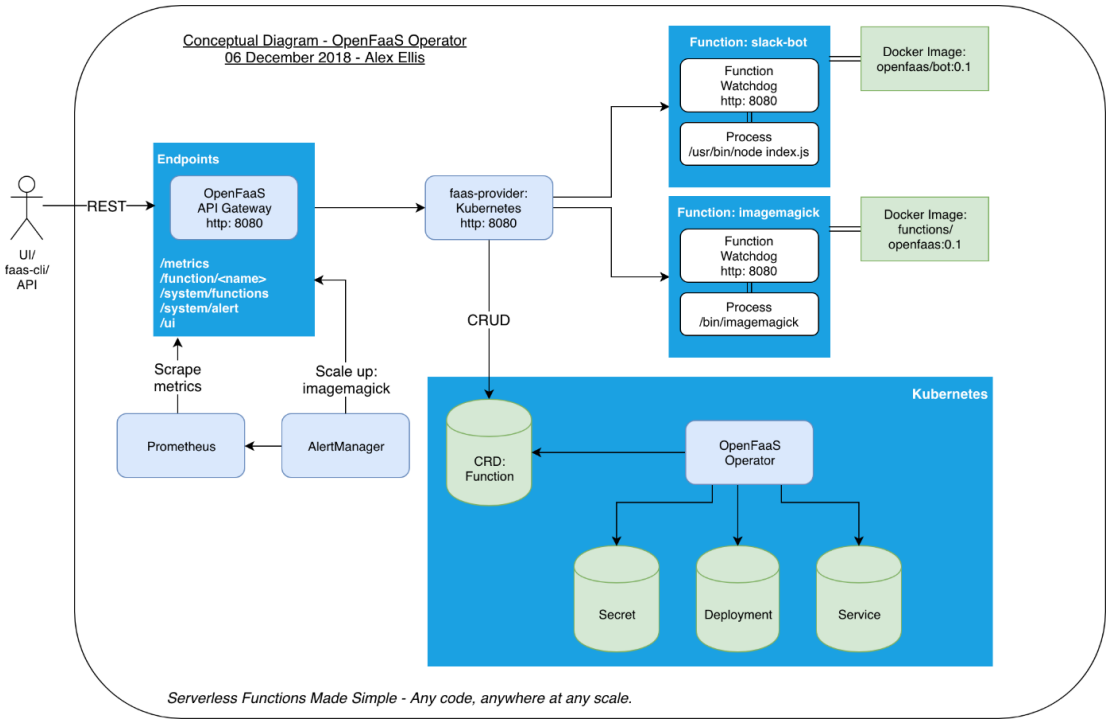
وانگ و همکاران [۱۲] سامانه InfiniCache را بر بستر رایانش بدون سرور آمازون پیاده‌سازی کردند. این سیستم از حافظه داخلی توابع AWS Lambda به‌عنوان حافظه نهان بهره می‌برد و استخری از توابع را به‌عنوان گره‌های حافظه نهان نگهداری می‌کند. داده‌ها در این توابع ذخیره می‌شوند و مکانیزم‌هایی برای تضمین دسترس‌پذیری بالا و تحمل‌پذیری خطا در نظر گرفته شده است. در ادامه، آن‌ها در پژوهش بعدی خود یعنی InfiniStore [۱۳]، تلاش کردند محدودیت‌های رویکرد پیشین را مرتفع سازند. در این نسخه، به‌منظور افزایش کشسانی فضای ذخیره‌سازی حافظه نهان و ماندگاری توابع حامل داده، از الگوی زباله‌روبی در زبان‌های برنامه‌نویسی استفاده شد. بدین ترتیب، زمانی که دسترسی به داده‌های یک تابع از یک بازه زمانی مشخص (مثلاً ۳۰ دقیقه) فراتر رود، داده‌ها به‌عنوان سرد شناسایی شده و حذف می‌شوند. همچنین، با افزودن یک لایه ذخیره‌سازی اشیای ابری، ماندگاری داده‌ها تضمین گردید.

در پژوهشی مشابه، موندو و همکاران [۱۷] سیستم OFC را معرفی کردند. آن‌ها استدلال کردند که بسیاری از توسعه‌دهندگان توابع در محیط‌های بدون‌سرور، حافظه‌ای بیش از نیاز واقعی درخواست می‌کنند؛ بنابراین می‌توان از این بخش اضافی برای ذخیره‌سازی داده‌های حافظه نهان استفاده کرد. در این رویکرد، ابتدا با استفاده از مدل‌های یادگیری ماشین، میزان حافظه موردنیاز هر تابع برآورد می‌شود و سپس بخش مازاد آن به‌عنوان حافظه نهان به‌کار گرفته می‌شود. این سیستم بر بستر متن‌باز OpenWhisk پیاده‌سازی شده است.

رومرو و همکارانش [۱۱] نیز سیستم Faa$T را پیشنهاد دادند که از نزدیک‌ترین کارها به پژوهش حاضر به‌شمار می‌رود. در این سیستم، برای هر برنامه کاربردی و کلیه توابع آن، یک حافظه نهان اختصاصی ایجاد می‌شود. با اتمام اجرای برنامه، حافظه نهان آزاد می‌گردد، اما اطلاعات مرتبط توسط Faa$T ذخیره می‌شود تا در فراخوانی‌های بعدی برای پیش‌گرم‌سازی حافظه نهان با داده‌های پرتکرار استفاده شود. علاوه بر این، اندازه حافظه نهان بر اساس سه عامل تغییر می‌یابد: نرخ فراخوانی برنامه در ثانیه، تعداد دفعات دسترسی به داده، و اندازه شئ داده. این سامانه نیز بر بستر Microsoft Azure Functions پیاده‌سازی شده است.

چنان‌که مشاهده شد، اغلب پژوهش‌های پیشین مسئله مدیریت حافظه نهان را در سطح یک ماشین (فیزیکی یا مجازی) و برخی دیگر در سطح

1. Cache
2. Origin Function
3. Multisite Transactional Causal Consistency



شکل ۱: معماری OpenFaaS در بستر کوبرنتیز [۱۸].

یک برنامه کاربردی بررسی کرده‌اند. در مقابل، نوآوری مقاله حاضر در آن است که مدیریت حافظه نهان را در سطح تابع مورد توجه قرار داده است. این رویکرد ریزدانه، کارایی و مقیاس‌پذیری بیشتری را متناسب با الگوی فراخوانی توابع فراهم می‌سازد.

۳- سیستم پیشنهادی

با توجه به محدودیت‌های موجود در استفاده از سرویس‌دهندگان ابری و نیاز به دسترسی به محیط‌های متن‌باز برای اعمال تغییرات، تمرکز این پژوهش بر روی سکوهای متن‌باز قرار گرفت. در میان سامانه‌های متن‌باز موجود، سه سکوی OpenFaaS، OpenWhisk و OpenLambda از محبوبیت بیشتری برخوردار هستند [۲]. پس از بررسی ویژگی‌های هر سه سامانه، سکوی OpenFaaS به‌عنوان بستر پیاده‌سازی انتخاب شد. این سامانه از کوبرنتیز برای مدیریت توابع بهره می‌گیرد و انعطاف‌پذیری بالایی در زمینه استقرار و مقیاس‌پذیری فراهم می‌آورد. در ادامه، ابتدا محیط‌های OpenFaaS و کوبرنتیز معرفی می‌شوند و سپس چارچوب پیشنهادی تشریح خواهد شد.

۳-۱ سیستم OpenFaaS

سیستم **OpenFaaS** یک چارچوب متن‌باز است که نخستین بار در سال ۲۰۱۷ معرفی شد. هدف اصلی این سامانه، ساده‌سازی فرایند استقرار توابع بدون‌سرور است. ماهیت ترکیب‌پذیر آن امکان می‌دهد تا توابع در زبان‌های برنامه‌نویسی متداولی همچون پایتون، جاوا اسکریپت، سی شارپ، گولنگ و بسیاری زبان‌های دیگر به‌سادگی تعریف، ساخته و مستقر شوند. ساختار کلی OpenFaaS در شکل ۱ نشان داده شده است [۱۸].

OpenFaaS برای استقرار توابع از مفهوم ظرف[1] و سامانه مدیریت ظرف داکر[2] بهره می‌گیرد. ظرف درواقع یک بسته سبک‌وزن است که شامل کد برنامه (تابع) به همراه کلیه وابستگی‌های آن، مانند کتابخانه‌های موردنیاز یا محیط زمان اجرا، بوده و می‌تواند به‌طور مستقل و جدا از سایر برنامه‌ها اجرا شود. درOpenFaaS ، هر تابع در قالب یک تصویر[3] داکر ارائه می‌شود که تمامی اجزای لازم برای اجرای آن تابع را در بر دارد.

با این حال، وظایف مرتبط با کشف سرویس، مقیاس‌پذیری خودکار، زمان‌بندی و توزیع بار در OpenFaaS بر عهده کوبرنتیز است. در کوبرنتیز کوچک‌ترین واحد استقرار، پاد[4] نام دارد. هر پاد شامل یک یا چند ظرف به‌صورت محکم جفت‌شده[5] است که در کنار یکدیگر به‌عنوان یک واحد غیرقابل‌تفکیک، زمان‌بندی و مقیاس‌بندی می‌شوند. هنگام ایجاد یک پاد، زمان‌بند کوبرنتیز (Kube-Scheduler) میزبان مناسب برای استقرار آن را انتخاب می‌کند.

همان‌گونه که در شکل ۱ نشان داده شده است، در OpenFaaS برای هر تابع کاربر یک پاد اختصاصی ایجاد می‌شود که شامل دو ظرف است: ظرف حاوی کد تابع و ظرف نگهبان[6]. ظرف نگهبان در حقیقت یک دروازه رابط برنامه‌نویسی کاربردی[7] است که فراخوانی‌های ورودی را دریافت و به تابع منتقل می‌کند. نکته مهم آن است که اگر به‌مدت ۳۰ ثانیه هیچ درخواستی برای یک تابع ارسال نشود، پاد مربوطه آزاد شده و در فراخوانی بعدی، پدیده شروع سرد رخ خواهد داد [۳].

۳-۲ چارچوب Kacheless

چارچوب Kacheless بر بستر OpenFaaS توسعه یافته و سه تغییر اساسی در آن اعمال شده است:

۱. معرفی مفهوم توابع مبدأ،

۲. افزودن ظرف Kacheless-Handler،

۳. بازطراحی الگوریتم‌های زمان‌بندی توابع.

1. Container
2. Docker
3. Image
4. Pod
5. Tightly Coupled
6. Watchdog
7. API Gateway

۳-۲-۱ توابع مبدأ

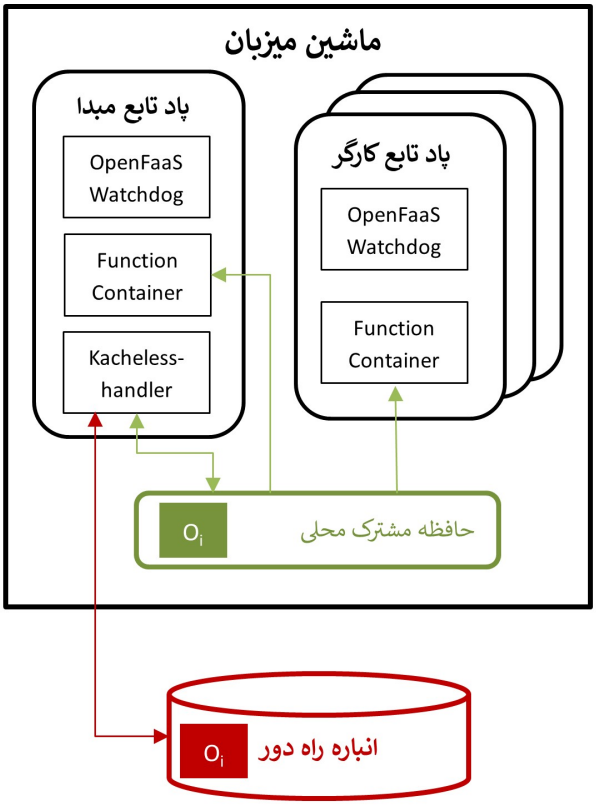
در ادامه، هر یک از این تغییرات به‌صورت تفصیلی مورد بررسی قرار می‌گیرد.

توابع مبدأ هسته اصلی چارچوب Kacheless را تشکیل می‌دهند. به این صورت که هنگام استقرار نخستین نسخه از یک تابع خاص بر روی یک میزبان، آن تابع به‌عنوان تابع مبدأ ایجاد می‌شود، در حالی که نسخه‌های بعدی همان تابع روی همان میزبان به‌صورت توابع کارگر راه‌اندازی خواهند شد. شکل ۲ ساختار این دو نوع تابع را بر روی یک میزبان نشان می‌دهد. همان‌طور که در شکل مشخص است، توابع کارگر مشابه پادهای عادی کوبرنتیز ایجاد می‌شوند؛ اما پاد مربوط به تابع مبدأ یک ظرف اضافی به نام Kacheless-handler دارد که مسئول مدیریت حافظه نهان مربوط به آن تابع است.

در چارچوب پیشنهادی، هر نوع تابع دارای یک حافظه نهان محلی بر روی هر میزبان است. این حافظه محلی بخشی از حافظه اصلی میزبان است که میان تابع مبدأ و تمامی توابع کارگر متناظر با آن بر روی همان میزبان به‌اشتراک گذاشته می‌شود. نخستین چالش در این زمینه، امکان‌پذیر نبودن اشتراک‌گذاری مستقیم حافظه اصلی میان پادهای مختلف در کوبرنتیز است. اگرچه کوبرنتیز اجازه اشتراک حافظه میان ظرف‌های یک پاد را فراهم می‌کند، اما این قابلیت برای پادهای جداگانه وجود ندارد. برای رفع این محدودیت، ابتدا بخشی از حافظه اصلی به کمک قابلیت tmpfs در یک مسیر مشخص سوار[1] شد. سرویس tmpfs یک سیستم فایل در هسته لینوکس است که داده‌ها را به جای دیسک جانبی در حافظه اصلی ذخیره می‌کند. سپس با بهره‌گیری از OpenEBS [۱۹] – که ابزاری برای ایجاد فضای ذخیره‌سازی پایا[2] میان پادهای کوبرنتیز است – این حافظه میان پادهای مبدأ و کارگر متعلق به یک تابع خاص به‌اشتراک گذاشته شد.

برای دسترسی به این حافظه اشتراکی، یک ابزار توسعه نرم افزار یا SDK[3] در اختیار برنامه نویسان توابع قرار گرفت که متدهایی برای عملیات خواندن و نوشتن روی اشیای داده[4] در انباره راه دور فراهم می‌کند. در زمان اجرای توابع (اعم از مبدأ و کارگر)، اگر یک شئ داده خوانده شود، ابتدا وجود آن در حافظه نهان محلی بررسی می‌گردد و در صورت موجود بودن مستقیماً بازگردانده می‌شود. در غیر این صورت، درخواست به Kacheless-handler تابع مبدأ ارسال می‌شود تا داده مورد نظر از انباره راه دور بازیابی شده و در حافظه نهان ذخیره گردد. در مقابل، عملیات نوشتن همواره باید از طریق Kacheless-handler انجام شود. به بیان دیگر، توابع کارگر تنها می‌توانند داده‌ها را به صورت مستقیم از حافظه نهان محلی بخوانند (در صورت موجود بودن شئ داده)، اما برای نوشتن، لازم است به تابع مبدأ متکی باشند. برای مدیریت این وابستگی، همه توابع کارگر باید به آدرس تابع مبدأ مربوطه دسترسی داشته باشند؛ بدین منظور از یک پایگاه‌داده مشترک Redis برای ذخیره و بازیابی آدرس توابع مبدأ استفاده می‌شود.

یک نکته مهم در طراحی این چارچوب آن است که دسترسی به



شکل ۲: ساختار توابع مبدا و توابع کارگر بر روی یک میزبان.

داده‌های ذخیره‌شده در حافظه نهان به صورت بدون کپی[5] انجام می‌شود. بدین معنا که نیازی به کپی‌کردن کل داده در حافظه تابع نیست و دسترسی مستقیماً از طریق اشاره‌گرهای حافظه مشترک صورت می‌گیرد. این رویکرد به شکل چشمگیری مصرف حافظه را کاهش می‌دهد.

۳-۲-۲ ظرف Kacheless-handler

همان‌طور که در بخش قبل بیان شد، Kacheless-handler قلب اصلی چارچوب پیشنهادی است و مسئولیت مدیریت حافظه نهان محلی را بر عهده دارد. این جزء با استفاده از مفهوم ظرف جانبی[6] [۲۰] در کوبرنتیز پیاده‌سازی شده است. ظرف جانبی یک ظرف ثانویه است که همراه با ظرف(های) اصلی در یک پاد اجرا می‌شود و بدون نیاز به تغییر در کد برنامه اصلی، خدمات و قابلیت‌های مکملی همچون نظارت، امنیت یا همگام‌سازی داده‌ها را فراهم می‌آورد.

برای الحاق این ظرف جانبی به توابع مبدأ، از وب‌هوک‌های[7] کوبرنتیز استفاده شده است. وب‌هوک در واقع یک بازفراخوانی[8] HTTP است که در هنگام دریافت درخواست اجرای یک تابع توسط کاربر، فراخوانده می‌شود. بدین ترتیب، هنگام ایجاد یک پاد جدید، بررسی می‌شود که آیا این نخستین پاد مربوط به آن تابع بر روی میزبان است یا خیر. در صورت مثبت بودن، ظرف جانبی Kacheless-handler به پاد در حال ساخت اضافه می‌گردد.

همان‌طور که پیش‌تر اشاره شد، Kacheless-handler تنها جزء چارچوب است که به‌طور مستقیم با انباره راه دور در ارتباط بوده و تمامی عملیات به‌روزرسانی حافظه نهان محلی باید از طریق آن انجام شود. بدین

1. Mount
2. Persistent Volume
3. Software Development Kit
4. Data Object
5. Zero-Copy
6. Sidecar Container
7. Webhook
8. Callback

منظور، هر درخواست برای خواندن داده‌ای که در حافظه محلی موجود نیست، و نیز هرگونه عملیات نوشتن (چه از سوی توابع کارگر و چه حتی تابع مبدأ)، به Kacheless-handler ارسال شده و توسط آن پردازش می‌گردد. افزون بر این، Kacheless-handler به‌طور مستمر وضعیت انباره راه دور را رصد کرده و در صورت تغییر در داده‌های مرتبط با حافظه نهان، نسخه‌های محلی را به‌روزرسانی می‌کند.

برای مدیریت فضای حافظه نهان، از الگوریتم LRU[1] استفاده شده است. همچنین در صورت ناکافی بودن فضای اختصاص‌یافته به حافظه نهان، امکان افزایش پویای آن پیش‌بینی شده است. معیار تصمیم‌گیری برای این افزایش، نرخ خطای حافظه نهان[2] است که بیانگر کارایی مکانیزم ذخیره‌سازی محلی می‌باشد.

۳-۲-۳ الگوریتم زمانبندی توابع

یکی از چالش‌های کلیدی در طراحی چارچوب Kacheless آن است که زمان‌بند پیش‌فرض کوبرنتیز کارایی مطلوبی برای این چارچوب ندارد و لازم است با یک راهکار مناسب‌تر جایگزین شود. پیش از تشریح جزئیات، باید توجه داشت که مفهوم زمان‌بندی در سامانه‌های بدون‌سرور دو جنبه متفاوت دارد:

۱. جایگذاری توابع[3]: انتخاب یک میزبان مناسب برای استقرار یک پاد جدید از یک تابع.
۲. زمان‌بندی درخواست‌ها: تخصیص درخواست‌های ورودی کاربران به یکی از نسخه‌های در حال اجرای آن تابع.

در کوبرنتیز، وظیفه جایگذاری پادها بر عهده Kube-Scheduler است که بر اساس امتیازدهی به میزبان‌ها عمل کرده و عمدتاً به دنبال ایجاد توازن بار میان ماشین‌های میزبان است. این رویکرد با الزامات Kacheless هم‌سو نیست، زیرا در چارچوب پیشنهادی محلیت[4] اهمیت ویژه‌ای دارد؛ به این معنا که اجرای نسخه‌های متعدد یک تابع خاص بر روی یک میزبان، امکان بهره‌برداری مؤثر از حافظه نهان مشترک را فراهم می‌سازد. با این حال، تمرکز صرف بر محلیت می‌تواند موجب ایجاد گلوگاه در میزبان‌های پر بار شود. بنابراین، لازم است الگوریتمی طراحی شود که تعادلی میان محلیت و بار میزبان برقرار نماید [۲۱] و [۲۲].

برای این منظور، در Kacheless یک الگوریتم حریصانه ساده به کار گرفته شده است. در فرآیند جایگذاری، ابتدا جستجو می‌شود که آیا میزبان‌هایی با یک تابع مبدأ فعال و بار کمتر از حد آستانه مشخص وجود دارد یا خیر. در صورت وجود چند گزینه، میزبانی با کمترین بار انتخاب می‌شود. اما اگر چنین میزبانی یافت نشود (به دلیل نبود تابع مبدأ فعال یا بار بیش از حد آستانه در تمامی میزبان‌های دارای تابع مبدأ)، میزبانی با کمترین بار در کل سیستم انتخاب شده و یک تابع مبدأ جدید بر روی آن ایجاد می‌گردد. برای سنجش بار میزبان، از میانگین بار یک‌دقیقه‌ای استاندارد در سیستم‌عامل لینوکس استفاده شده است.

در زمینه زمان‌بندی درخواست‌ها، کوبرنتیز به‌طور پیش‌فرض از الگوریتم‌های IPVS [۲۳] بهره می‌گیرد که نیازهای Kacheless را پوشش نمی‌دهند. مهم‌ترین دغدغه ما آن است که توابع مبدأ تا حد امکان فعال باقی بمانند، زیرا حذف هر تابع مبدأ منجر به از دست رفتن داده‌های حافظه نهان آن می‌شود. از سوی دیگر، در OpenFaaS هر تابعی که طی ۳۰ ثانیه متوالی فراخوانی نشود، از حافظه خارج می‌گردد. از این رو، الگوریتم پیشنهادی زمان‌بندی درخواست‌ها به این صورت عمل می‌کند: ابتدا تلاش می‌شود درخواست به یک تابع مبدأ با کمترین بار کاری تخصیص یابد که بار کاری آن از حد آستانه کمتر باشد؛ و اگر چنین گزینه‌ای وجود نداشت، درخواست به یک تابع کارگر با کمترین بار ارسال می‌شود. بدین ترتیب، در شرایط کاهش تعداد درخواست‌ها، ابتدا توابع کارگر از چرخه خارج می‌شوند و تنها در نهایت، توابع مبدأ آزاد خواهند شد.

این دو الگوریتم (جایگذاری توابع و زمان‌بندی درخواست‌ها) جایگزین مکانیزم‌های پیش‌فرض کوبرنتیز شده‌اند. جزئیات پیاده‌سازی فنی این جایگزینی خارج از حوزه این مقاله است. شایان ذکر است که در پیکربندی پیش‌فرض کوبرنتیز امکان زمان‌بندی مستقل فراخوانی‌های هر تابع وجود ندارد؛ ازاین‌رو تنظیمات خاصی برای فعال‌سازی این قابلیت اعمال شد که به دلیل پرهیز از طولانی‌شدن بحث، از ذکر آن صرف‌نظر می‌گردد.

٤- آزمایش‌ها و نتایج به‌دست آمده

در این بخش به آزمایشات انجام شده و نتایج حاصل از آنها خواهیم پرداخت. در ابتدا محیط آزمایشها به طور کامل بررسی شده و سپس نتایج به دست آمده را بررسی خواهیم کرد.

در این بخش به آزمایش‌های انجام‌شده و نتایج حاصل از آنها خواهیم پرداخت. در ابتدا محیط آزمایش‌ها به طور کامل بررسی شده و سپس نتایج به دست آمده را بررسی خواهیم کرد.

٤-١ محیط آزمایش

برای ارزیابی عمکرد چارچوب Kacheless، یک خوشه محاسباتی شامل پنج میزبان مورد استفاده قرار گرفت که هر یک مجهز به ۸ هسته پردازشی و ۱۶ گیگابایت حافظه اصلی بودند. پس از نصب کوبرنتیز و OpenFaaS بر روی این خوشه، تغییرات موردنیاز برای پیاده‌سازی چارچوب پیشنهادی اعمال شد. در این پیاده‌سازی، ماژول Kacheless-handler با استفاده از زبان پایتون توسعه یافت و الگوریتم‌های جایگذاری و زمان‌بندی نیز با بهره‌گیری از زبان گولنگ پیاده‌سازی شدند.

برای ارزیابی عمکرد چارچوب Kacheless، یک خوشه محاسباتی شامل پنج میزبان مورد استفاده قرار گرفت که هر یک مجهز به ۸ هسته پردازشی و ۱۶ گیگابایت حافظه اصلی بودند. پس از نصب کوبرنتیز و OpenFaaS بر روی این خوشه، تغییرات موردنیاز برای پیاده‌سازی چارچوب پیشنهادی اعمال شد. در این پیاده‌سازی، ماژول Kacheless-handler با استفاده از زبان پایتون توسعه یافت و الگوریتم‌های جایگذاری و زمان‌بندی نیز با بهره‌گیری از زبان گولنگ پیاده‌سازی شدند.

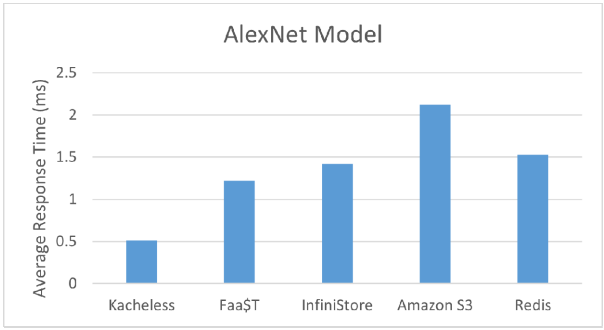
٤-١-١ توابع محک

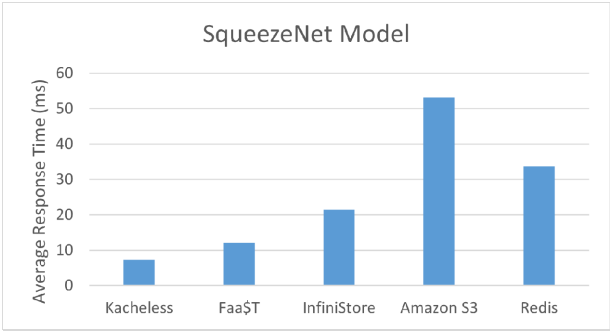
برای آزمایش‌های مرتبط با زمان پاسخ برنامه‌ها، از توابع محک موجود در [۲۴] FunctionBench استفاده شد. بدین منظور، دو برنامه استنتاج تک‌مدلی انتخاب گردید: SqueezeNet (با حجم ۵ مگابایت) و AlexNet (با حجم ۲۳۹ مگابایت).

برای آزمایش‌های مرتبط با میزان مصرف حافظه، از ردپای دو هفته‌ای فراخوانی توابع در Azure Functions، در بازه زمانی ۳۱ ژانویه تا ۱۳ فوریه ۲۰۲۱ استفاده گردید [۲۵]. این آزمایش‌ها در دو بخش انجام شدند:

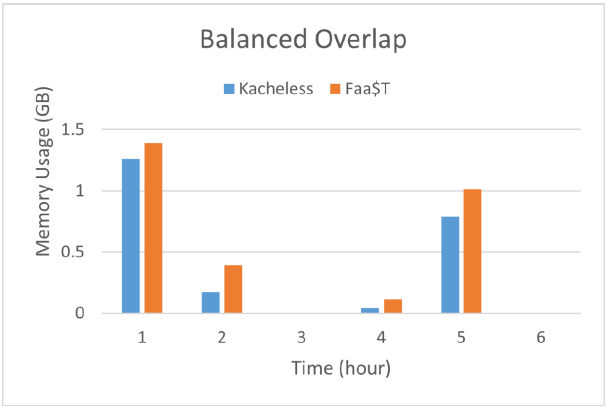
۱. هم‌پوشانی متعادل: در این بخش دو برنامه استنتاج خطلوله‌ای انتخاب شدند که حدود نیمی از توابع آن‌ها مشترک بودند.

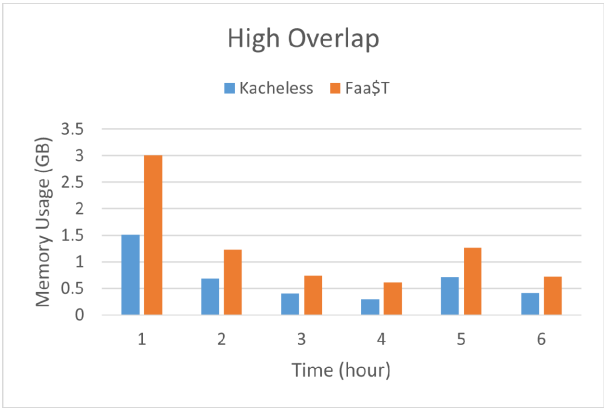
1. Least Recently Used
2. Cache Miss Rate
3. Function Placement
4. Locality





شکل ۳: مقایسه میانگین زمان پاسخ بین معماری پیشنهادی و راهکارهای رقیب.





شکل ۴: مقایسه ردپای حافظه مصرفی بین روشهای Kacheless و Faa$T.

۲. هم‌پوشانی بالا: در این بخش نیز دو برنامه انتخاب شدند که حدود ۹۰ درصد توابع آن‌ها مشترک بودند.

این انتخاب‌ها امکان بررسی سناریوهایی را فراهم می‌سازد که در آن‌ها، الگوی فراخوانی برخی توابع با الگوی فراخوانی کل برنامه تفاوت دارد.

٤-۱-۲ راهکارهای مورد مقایسه

برای راهکارهای مقایسه‌ای، دو روش کلاسیک و دو رویکرد جدید و پرارجاع انتخاب شدند. روش‌های کلاسیک شامل استفاده از انبارههای شئ راه دور نظیر Amazon S3 و همچنین بهره‌گیری از مخازن ذخیره‌سازی در حافظه اصلی مانند Redis هستند. برای پیاده‌سازی روش نخست، با بهره‌گیری از MinIO [۲۴] یک خوشه انباره شئ سازگار با S۳ در همان شبکه کوبرنتیز مستقر گردید. برای روش دوم نیز یک خوشه از گره‌های Redis بر بستر کوبرنتیز راه‌اندازی شد.

برای مقایسه با رویکردهای جدید نیز دو راهکار Faa$T [۱۱] و InfiniStore [۱۳] انتخاب شدند که در میان روش‌های موجود از جایگاه بالایی در استنادها برخوردارند. جزئیات بیشتر درباره این دو روش در بخش ۲ ارائه شده است. نکته حائز اهمیت آن است که Faa$T به طور خاص برای محیط Azure Functions و InfiniStore برای محیط AWS Lambda طراحی شده‌اند. با این حال، با اعمال تغییراتی در هر دو روش، امکان استفاده از آنها در محیط OpenFaaS فراهم گردید.

٤-۲ نتایج آزمایش‌ها

برای ارزیابی کارآیی چارچوب Kacheless در مقایسه با سایر روش‌ها، ابتدا معیار میانگین زمان پاسخ مورد بررسی قرار گرفت. منظور از زمان پاسخ، مدت زمانی است که طول می‌کشد تا یک درخواست اجرای تابع پردازش شده و نتیجه آن بازگردانده شود. این معیار برای دو برنامه SqueezeNet و AlexNet در ده مرتبه اجرا اندازه‌گیری شد و میانگین نتایج در شکل ۳ ارائه گردیده است. همان‌گونه که مشاهده می‌شود، چارچوب Kacheless در مقایسه با سایر روش‌ها زمان پاسخ کمتری داشته است. همچنین نتایج نشان می‌دهند که با افزایش حجم داده‌ها در مدل AlexNet، برتری کارآیی Kacheless نسبت به روش‌های دیگر بیشتر شده و فاصله عملکردی آن افزایش یافته است.

در ادامه، به منظور بررسی کارآمدی روش‌ها در استفاده از منابع، ردپای مصرف حافظه در برنامه‌های استنتاج خط لوله‌ای برای دو روش Kacheless و Faa$T مقایسه شد. همان‌طور که پیش‌تر بیان شد، روش‌های پیشین مانند Faa$T و InfiniStore مدیریت حافظه نهان را در سطح برنامه انجام می‌دهند، در حالی که Kacheless این مدیریت را در سطح هر تابع اعمال می‌کند و همین امر موجب بهینه‌سازی مصرف منابع می‌شود. در این بخش، تنها روش Faa$T به دلیل کارآیی بالاتر به‌عنوان رقیب در نظر گرفته شد و روش‌های کلاسیک به دلیل عدم پشتیبانی از حافظه نهان در مقایسه لحاظ نگردیدند.

شکل ۴ مجموع مصرف حافظه در هر ساعت را برای دو برنامه منتخب، در دو سناریوی هم‌پوشانی متعادل و هم‌پوشانی بالا طی شش ساعت متوالی نشان می‌دهد. نتایج نشان می‌دهد که Kacheless مصرف حافظه کمتری نسبت به Faa$T دارد. این برتری به‌ویژه در سناریوی هم‌پوشانی بالا چشمگیرتر است؛ جایی که Kacheless با مدیریت حافظه نهان در سطح هر تابع (به جای کل برنامه)، موفق شده است میزان مصرف حافظه را در مقایسه با Faa$Tتا حدود ۵۰ درصد کاهش دهد.

۵- نتیجه‌گیری و کارهای آینده

در این مقاله، چارچوب Kacheless برای اجرای توابع حالتمند در بستر محاسبات بدون سرور معرفی شد. این چارچوب بر روی سکوهای متن‌باز OpenFaaS و کوبرنتیز پیاده‌سازی گردید. در معماری پیشنهادی، از حافظه نهان در سطح تابع برای افزایش سرعت پاسخ‌دهی به کاربر استفاده شده است. برای مدیریت حافظه نهان، مفهومی تحت عنوان توابع

مبدأ تعریف شد که وظیفه هماهنگی و مدیریت داده‌های نهان بر روی هر میزبان را بر عهده داشته و این امکان را فراهم می‌کنند که توابع کارگر بتوانند با سرعت بیشتری به داده‌های خود از طریق حافظه نهان دسترسی یابند.

علاوه بر این، الگوریتم‌های جایگذاری توابع و زمانبندی درخواست‌ها در کوبرنتیز متناسب با چارچوب پیشنهادی تغییر یافته‌اند. نتایج آزمایش‌ها نشان داد که این روش نسبت به راهکارهای کلاسیک و همچنین روش‌های جدید مدیریت حافظه نهان، از جمله Faa$T و InfiniStore، از میانگین زمان پاسخ کمتری برخوردار بوده و در عین حال مصرف حافظه را نیز کاهش می‌دهد.

به عنوان مسیرهای آینده، می‌توان به امکان همکاری و هماهنگی بین توابع مبدأ برای بروزرسانی داده‌های حافظه نهان، بدون نیاز به مراجعه مداوم به انباره ذخیره‌سازی راه دور اشاره کرد. همچنین به‌کارگیری الگوریتم‌های پیشرفته‌تر در حوزه جایگذاری توابع و زمانبندی درخواست‌ها می‌تواند موجب بهبود بیشتر کارایی شود. جایگذاری توابع و زمانبندی پیچیده‌تر، برای بالارفتن کارآیی استفاده کرد.

مراجع

[1] G. P. Mattia and R. Beraldi, "P2PFaaS: A framework for FaaS peer-to-peer scheduling and load balancing in fog and edge computing," *SoftwareX,* vol. 21, Article ID: 101290, Feb. 2023.

[2] J. Wen, Z. Chen, X. Jin, and X. Liu, "Rise of the planet of serverless computing: A systematic review," *ACM Trans. Software Engineeing. Methodology*, vol. 32, no. 5, Article ID: 13, 2023.

[3] H. Shafiei, A. Khonsari, and P. Mousavi, "Serverless computing: A survey of opportunities, challenges, and applications," *ACM Computing Survey*, vol. 54, no. 11S, Article ID: 239, Jan. 2022.

[4] Amazon, Amazon S3, [Online] Available: https://aws.amazon.com/s3/ [Accessed Jan. 2024]

[5] –, *Cloudstate: Distributed State Management for Serverless*, [Online]. Available: https://github.com/cloudstateio/cloudstate [Accessed Jan. 2024].

[6] –, *Stateful Functions: A Platform-Independent Stateful Serverless Stack*, [Online]. Available: https://nightlies.apache.org/flink/flink-statefun-docs-master/ [Accessed Ja. 2024].

[7] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu, "Fault-tolerant and transactional stateful serverless workflows," in *Proc. 4th USENIX Symp. on Operating Systems Design and Implementation*, pp. 1187-1204, 4-6 Nov. Aug. 2020.

[8] V. Sreekanti, *et al*., "Cloudburst: stateful functions-as-a-service," *Proc. of the VLDB Endowment*, vol. 13, no. 11, pp. 2438-2452, 2020.

[9] Z. Jia and E. Witchel, "Boki: Stateful serverless computing with shared logs," in *Proc. of the ACM SIGOPS 28th Symp. on Operating Systems Principles*, pp. 691-707, Virtual Event Germany, 26-29 Oct. 2021.

[10] D. Barcelona-Pons, P. Sutra, M. Sanchez-Artigas, G. Paris, and P. Garcia-Lopez, "Stateful serverless computing with crucial," *ACM Trans. on Software Engineering and Methodology,* vol. 31, no. 3, Article ID: 39, 38 pp., Jul. 2022.

[11] F. Romero, *et al.*, "Faa$T: A transparent auto-scaling cache for serverless applications," in *Proc. of the ACM Symp. on Cloud Computing*, pp. 122-137, Seattle, WA, USA, 1-4 Nov. 2021.

[12] Ao Wang, *et al.*, "InfiniCache: exploiting ephemeral serverless functions to build a cost-effective memory cache," in *Proc. of the 18th USENIX Conf, on File and Storage Technologies*, pp. 267-282, Santa Clar, CA, USA, 24-27 Feb. 2020.

[13] J. Zhang, *et al.*, "InfiniStore: Elastic serverless cloud storage," *Proc. of the VLDB Endowment,* vol. 16, no. 7, p. 1629-1642, Mar. 2023.

[14] –, *Kubernetes*, [Online]. Available: https://kubernetes.io/ [Accessed Jan. 2024].

[15] A. Klimovic, *et al.*, "Pocket: Elastic ephemeral storage for serverless analytics," in *Proc. of the 13th USENIX Symp. on Operating Systems Design and Implementation*, pp. 428-444, Carlsbad, CA, USA, 8-10 Oct. 2018.

[16] C. Wu, V. Sreekanti, and J. M. Hellerstein, "Transactional causal consistency for serverless computing," in *Proc. of the* 2020 ACM SIGMOD Int. Conf. on Management of Data, pp. 83-97, Portland, OR, USA, 14-19 Jun. 2020.

[17] D.Mvondo, *et al.*, "OFC: an opportunistic caching system for FaaS platforms," in *Proc. of the 16th European Conf. on Computer Systems*, pp. 228-244, Virtual Event UKy, 26-28 Apr. 2021.

[18] –, *OpenFaaS*, [Online]. Available: https://docs.openfaas.com/architecture/gateway/ [Accessed Jan. 2024]

[19] –, *OpenEBS*, [Online]. Available: https://openebs.io/docs/ [Accessed January 2024]

[20] –, *Sidecar Containers, 2023*. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/pods/sidecar-containers/ [Accessed 5 Sept. 2023].

[21] A. Fuerst and P. Sharma, "Locality-aware load-balancing for serverless clusters," in *Proc. of the 31st Int. Symp. on High-Performance Parallel and Distributed Computing*, pp. 227-239, Minneapolis, MN, USA, 27 Jun.-1 Jul. 2022.

[22] Y. Lee and S. Choi, "A greedy load balancing algorithm for FaaS platforms," in *Proc. of the 2021 5th Int. Conf. on Cloud and Big Data Computing*, pp. 63-69, Liverpool, UK, 13-15 Aug. 2021.

[23] –, *IPVS Scheduling Algorithm*, [Online]. Available: https://keepalived-pqa.readthedocs.io/en/latest/scheduling_algorithms.html [Accessed Jan. 2024].

[24] J. Kim and K. Lee, "FunctionBench: A suite of workloads for serverless cloud function service," in *Proc. 2019 IEEE 12th Int. Conf. on Cloud Computing*, pp. 502-504, Milan, Italy, 8-13 Jul. 2019.

[25] S. Elnikety, *Azure Public Dataset*, [Online]. Available: https://github.com/Azure/AzurePublicDataset/blob/master/AzureFunctionsInvocationTrace2021.md [Accessed Jan. 2024].

محمد کاهانی در سال ۱۴۰۰ مدرک کارشناسی مهندسی کامپیوتر و در سال ۱۴۰۳ مدرک کارشناسی ارشد مهندسی نرم‌افزار خود را از دانشگاه فردوسی مشهد دریافت نمود. نام‌برده از سال ۱۴۰۱ تاکنون به عنوان مهندس ارشد ابر در شرکت ویسپو مشغول به کار است. زمینه‌های علمی مورد علاقه ایشان شامل موضوعاتی مانند پردازش‌های ابری، پردازش‌های بدون سرور و سیستم‌های توزیع‌شده می‌باشد.

سعید ابریشمی تحصیلات خود را در مقاطع کارشناسی ارشد و دکترای مهندسی کامپیوتر به‌ترتیب در سال‌های ۱۳۷۸ و ۱۳۹۰ از دانشگاه فردوسی مشهد به پایان رسانده است و اکنون دانشیار دانشکده مهندسی دانشگاه فردوسی مشهد می‌باشد. زمینه‌های تحقیقاتی مورد علاقه ایشان عبارتند از: مدیریت منابع و زمانبندی در محیط‌های توزیع شده، به ویژه محیط رایانش ابری، رایانش لبه‌ای و مهی، و محیط‌های بدون خدمتگزار.

عادل نجاران طوسی مدرک دکترای مهندسی کامپیوتر خود را از دانشگاه ملبورن استرالیا در سال ۱۳۹۴ دریافت کرد. وی از سال ۱۳۹۷ تا سال ۱۴۰۳ به عنوان استادیار در دانشگاه موناش استرالیا مشغول به کار بود و پس از آن به عنوان دانشیار در دانشگاه ملبورن مشغول به کار گردید. زمینه‌های علمی مورد علاقه نام‌برده شامل رایانش ابری، رایانش لبه‌ای، محاسبات بدون خدمتگزار، رایانش سبز و انرژی-کارا می باشد.