به نام خدا

فاز اول پروژه- تحلیل و طراحی سیستم‌ها

دکتر علیرضا آقامحمدی

اسامی افراد تیم:

هیراد داوری ۹۹۱۰۶۱۳۶

ارشیا دادرس ۹۹۱۰۹۱۰۹

سارینا زاهدی دره‌شوری ۹۸۱۷۰۸۳۸

رامتین میرزامحمد خوشنویس ۹۹۱۰۵۷۶۴

مقدمه

در عصر داده‌های بزرگ و خدمات توزیع‌شده، توانایی مدیریت و انتقال پیام‌ها به شیوه‌ای مؤثر و قابل اعتماد حیاتی است. سیستم‌های صف پیام‌رسانی معاصر، چون Kafka و RabbitMQ، زیربنای اساسی را برای معماری‌های ریزسرویسی فراهم می‌کنند و اطمینان می‌دهند که تراکنش‌ها و فرآیندهای کسب‌وکار حتی در شرایط تقاضای سنگین شبکه‌ای، دقیق و بدون وقفه اجرا شوند.

در پروژه‌ی حاضر، ما می‌خواهیم سیستم صف پیام‌رسانی‌ای را پیاده‌سازی کنیم که بتواند داده‌ها را در سطح وسیع با پایداری و کارایی بالا انتقال دهد. این سیستم به گونه‌ای طراحی می‌شود که در برابر خطاها مقاوم بوده و در عین حال امکان پشتیبانی و توسعه راحت شبکه را فراهم کند. در نتیجه، این مستند به جزئیات فنی می‌پردازد که چگونه سیستم صف پیام‌رسانی ما به تحقق این اهداف کمک می‌کند.

این مستندات با جزئیات وارد عمق فنی این پروژه نمی‌شود و فقط سیستم صف پیام‌رسانی ما را از منظرهای مختلفی مانند معماری سیستم، قابلیت‌های کلیدی، و توجیه انتخاب‌های ما بررسی می‌کند.

معماری سیستم

پیاده‌سازی سیستم صف پیام‌رسانی ما بر پایه‌ی چارچوب‌های مقیاس‌پذیر، توزیع‌شده و مرتبط بنا شده است تا از چالش‌های مربوط به تحویل پیام‌ها در مقیاس بالا، مدیریت داده‌ها و برخورد با شرایط خطا به طور مؤثری روبرو شود.

- **سرورهای پیام‌رسانی:** سرورهای پیام‌رسان بنیادی ما با استفاده از زبان برنامه‌نویسی GoLang طراحی شده‌اند، که با بهره‌گیری از معماری مبتنی بر رویداد و مدل‌های همروندی متعالی‌ای که GoLang ارائه می‌دهد، سیستمی با بهره‌وری بالا، پاسخ‌گویی فوری و پردازش موازی ایجاد کنیم. نتیجه، یک سرویس زیرساختی است که قادر به مدیریت بارهای کاری سنگین در پیام‌رسانی توزیع‌شده است.
- **زیرساخت کانتینر:** کانتینرسازی بخشی حیاتی از زیرساخت و اورکستراسیون سیستم ماست. با استفاده از Docker برای کانتینریزاسیون و Kubernetes برای مدیریت آن‌ها، ما به یک مدیریت منسجم منابع، مقیاس‌پذیری سریع و استقرار دقیق برنامه‌ها دست می‌یابیم. این موضوع اطمینان حاصل می‌کند که سرویس‌ها می‌توانند با ثبات و بدون اتکا به زیرساخت خاصی، در محیط‌های ابری گوناگون فعال باشند.
- **پروتکل‌های ارتباطی:** ارتباطات در سیستم صف پیام‌رسانی ما از طریق REST API هایی که بر اساس اصول HTTP واضح و قابل فهم ساخته شده‌اند، مدیریت می‌شوند. برای تأمین امنیت داده‌ها در مسیریابی پیام‌ها، از پروتکل‌های امن TLS/SSL استفاده می‌کنیم که اطلاعات را در برابر شنود و تغییر داده‌ها هنگام انتقال بین کلاینت و سرور محافظت می‌باشند. همه‌ی درخواست‌های API باید از طریق HTTPS انجام شوند، که ترکیبی از پروتکل HTTP با لایه‌ی socket امن TLS/SSL است، تا از جامعیت، احراز هویت و محرمانگی ارتباطات اطمینان حاصل کند. چه در فاز توسعه و چه در استقرار، این رویکرد به ما کمک می‌کند تا از اطلاعات حساس کاربران و سایر داده‌های کاربردی در برابر نقض‌های احتمالی امنیتی دفاع کنیم.

- **سرویس‌های زمینه‌ای و داده‌پذیری:** سرویس‌های زمینه‌ای در سیستم ما به‌گونه‌ای طراحی می‌شوند که مدیریت پیام، ذخیره‌سازی داده و بازیابی را به طور اطمینان‌بخشی انجام دهند. معماری داده‌پذیری ما شامل پایگاه داده‌های توزیع‌شده با پشتیبانی از تکثیر و ساز و کارهای تحمل‌پذیر خطا می‌شود، برای اطمینان از دوام و دسترسی مداوم به داده‌ها. این ساختار با هدف کارکرد بدون از دست دادن هیچ پیامی، حتی در شرایط شکست‌های سیستمی طراحی می‌شود.
- **CI/CD:** فرآیند توسعه و انتشار همواره نیازمند نظارت و دقت فراوانی است. ما با بهره‌گیری از قابلیت‌های GitHub Actions، روشی خودکار برای تست و استقرار برنامه‌ها فراهم آورده‌ایم، تا اطمینان حاصل شود که تولید نهایی ما همواره با بالاترین استانداردهای کیفیت سازگار است.
- **مانیتورینگ و تحلیل:** برای آنکه بتوانیم با اطمینان عملکرد سیستم‌مان را در زمان واقعی رصد کرده و پیش‌بینی‌های دقیقی در رابطه با آینده داشته باشیم، ما به استفاده از ابزارهای پیشرفته‌ی مانیتورینگ نظیر Prometheus و Grafana متوسل شده‌ایم. این ابزارها اطلاعات جامع و معناداری از عملکرد سیستم در اختیارمان قرار می‌دهند.

قابلیت‌های کلیدی

- **مقیاس‌پذیری:** سیستم ما با استفاده از معماری توزیع‌شده و کانتینریزه، طراحی شده است تا در برابر افزایش بار کاری، هم از لحاظ عمودی (افزایش قدرت سخت‌افزاری) و هم افقی (توزیع بار کار بر روی سخت‌افزارهای متعدد) مقیاس‌پذیر باشد. این امر از طریق اتوماتیک‌سازی گسترش نودها و توان پردازشی بر اساس نیازهای محاسباتی و داده‌ای میسر می‌شود.
- **تحمل‌پذیری خطا:** سیستم از مکانیزم‌های تحمل‌پذیر خطا چون تکثیر داده و بازیابی پیام‌ها، برای اطمینان از دستیابی و مقاومت در مقابل شکست‌های سیستم‌های فردی بهره می‌برد. عملیات بدون وقفه و بازیابی سریع پس از خطاها، در اولویت قرار دارد.
- **الگوریتم اجماع بی‌رهبر:** برای کاهش خطرات وابستگی به نودهای انفرادی و ایجاد توزیع قدرت در شبکه، سیستم از الگوریتم‌های اجماع پیشرفته استفاده می‌کند که برای تصمیم‌گیری جمعی و هماهنگی در میان نودها اطلاق می‌شود، کاهش داده‌های تک نقطه شکست را به همراه دارد.
- **امنیت:** با استفاده از پروتکل‌های امنیتی مانند TLS/SSL، ما انتقال داده‌ها را در برابر شنود و دیگر تهدیدات محافظت می‌کنیم. این رویکرد امنیتی در جریان درخواست‌های REST API به کار گرفته می‌شود تا تضمین کنیم که تمام تبادلات داده‌ای مطابق با استانداردهای صنعتی امن انجام می‌شوند.
- **مانیتورینگ:** یک نظام مانیتورینگ پیشرفته که طیف گسترده‌ای از معیارهای عملکرد را در زمان واقعی ردیابی می‌کند، به ما اجازه می‌دهد تا مشکلات را به سرعت شناسایی و واکنش نشان دهیم. این سیستم همچنین به ما در شناسایی الگوهای عملیاتی برای بهینه‌سازی مستمر سیستم کمک می‌کند.

توجیه انتخاب‌ها

در این بخش، توجیه انتخاب‌های تکنیکی و فنی که برای معماری سیستم و قابلیت‌های کلیدی آن اتخاح شده‌اند، به روزرسانی و گسترش یافته تا یک دید جامع‌تر و دقیق‌تر از علت‌های پشت تصمیمات مهم فنی ارائه داده شود:

- **انتخاب زبان برنامه‌نویسی GoLang:** زبان Go به دلیل سادگی، قابلیت همروندی (Concurrency) قوی، و کارایی بالا برای ساخت برنامه‌های شبکه‌ای و سیستمی انتخاب شده است. Go قابلیت برخورداری از مقیاس‌پذیری و پاسخ‌دهی بالا را در حین عملیات موازی و دستکاری و استفاده مؤثر از چندین هسته پردازنده فراهم می‌کند.
- **استفاده از Docker و Kubernetes:** با بهره‌برداری از دو فناوری پیشرو در حوزه معماری کانتینری و همچنین اورکستراسیون کانتینر، سیستم می‌تواند از مزایای مقیاس‌پذیری، استقرار سریع‌تر، ورژن‌بندی و مدیریت عملکرد بهینه بهره‌مند شود. Kubernetes در توانایی خود در مدیریت حالات خطا و ترمیم خودکار بی‌نظیر است، که انعطاف‌پذیری و تحمل‌پذیری خطا در سیستم ما را تقویت می‌کند.
- **امنیت از طریق TLS/SSL:** تعهد ما به امنیت منجر به انتخاب استفاده از پروتکل‌های امن TLS/SSL شده است، این کار نه تنها انتقال داده‌ها بلکه کل نظام ارتباطی بین سرورها و کلاینت‌ها را امن می‌سازد. در طی تبادلات از طریق REST API ها، این استراتژی امنیتی در کنار رمزگذاری قوی داده‌ها از اعتمادپذیری سیستم ما اطمینان می‌دهد.
- **رویکرد CI/CD با GitHub Actions:** اتخاذ GitHub Actions برای اتوماسیون فرآیندهای CI/CD، توانایی تیم ما را در دستیابی سریع به بازخورد، به روزآوری، و تحویل مداوم افزایش داده و به‌طور مؤثر بهبود بخشیدن و نگهداری سیستم را ممکن می‌سازد. این رویکرد، خودکارسازی و استانداردسازی فرآیندهای تست و استقرار را تضمین می‌کند.
- **مانیتورینگ دقیق:** بکارگیری ابزارهای مانیتورینگ و تحلیل، جزء اساسی از فرآیند بهبود مستمر ما است. داشتن دید کامل بر عملکرد سیستم و توانایی شناسایی فوری مشکلات پیش از تبدیل شدن آن‌ها به قطعی‌های طولانی‌مدت یا ایجاد خرابی‌های وسیع، افزایش بلادرنگ کارایی و آماده‌سازی برای مقابله با چالش‌های آتی را ممکن می‌کند.

جزئیات معماری

سرویس ما از تعدادی پاد تشکیل شده است که همواره یکی از آن‌ها به عنوان Leader فعلی توسط خود سیستم مشخص می‌شود. Leader همواره یک پاد در دسترس است و در صورت بروز مشکل برای آن یک Leader جدید برای سیستم توسط هسته انتخاب می‌شود. در نتیجه هریک از پادها باید آمادگی و قابلیت معرفی به عنوان Leader را داشته باشند.

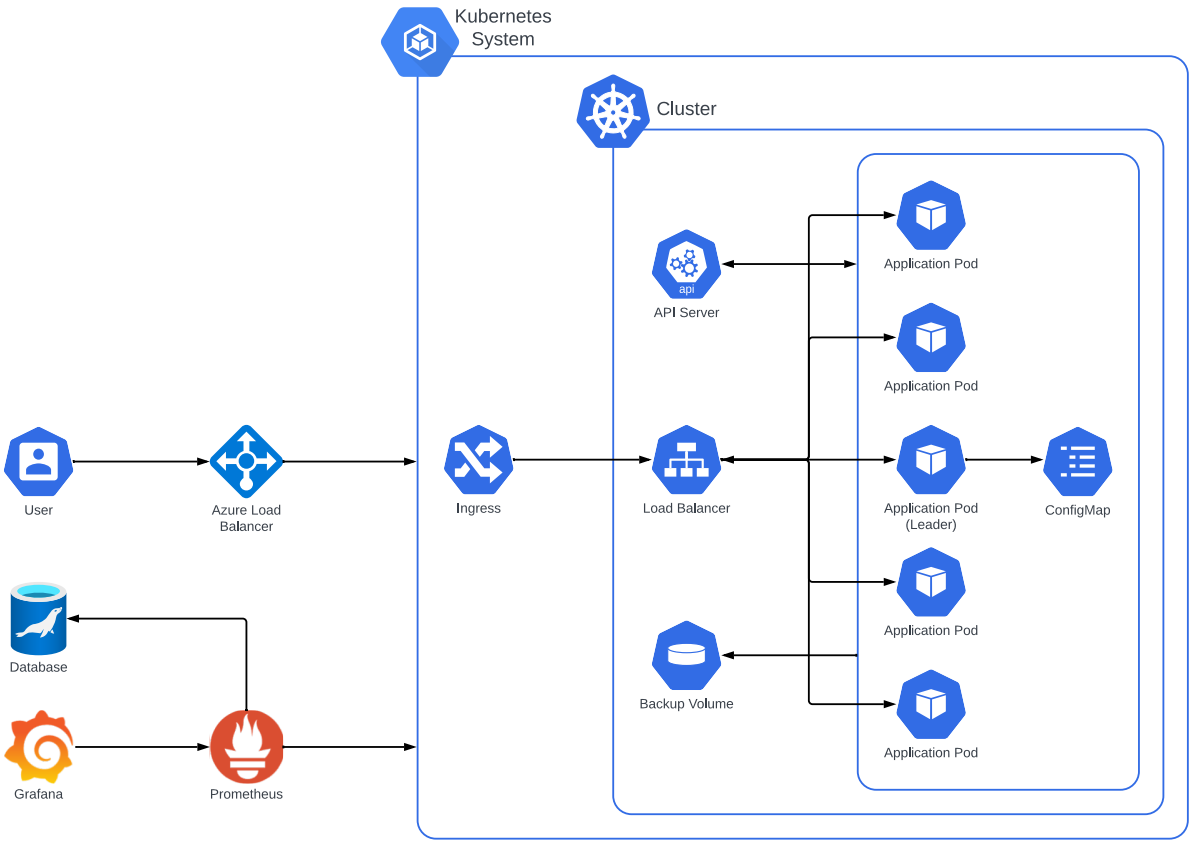
هرکدام از پادها در اصل یک Queue هستند و داده‌های خود را در RAM ذخیره می‌کنند. هر پاد در صورت رسیدن یک درخواست از سمت کاربر به خود، بسته به اینکه Leader باشد یا خیر به دو روش عمل می‌کند؛ اگر Leader سرویس نباشد درخواست را به Leader منتقل می‌کند. در صورت Leader بودن، با توجه به شرایط کنونی تصمیم به انتقال درخواست برای اجرا به تعدادی از پادهای موردنظر خود می‌کند.

فرآیند نوشتن در داخل پادها به این نحو است که پاد Leader پس از دریافت درخواست کاربر، دوتا از سایر پادهای موجود را به عنوان دریافت‌کننده‌ی این داده انتخاب می‌کند. سپس به هرکدام از آن‌ها درخواست اضافه کردن داده به Queue خودشان را می‌دهد. علاوه بر خود داده، دو عدد دیگر توسط Leader به پادها ارسال می‌شود که یکی از آن‌ها شمارنده‌ی درخواست و دیگری دریافت‌کننده‌ی دیگر پیام است.

علاوه بر Queue، در پاد Leader یک داده‌ساختار Heap نیز ذخیره شده است که در آن به ازای هر پاد، یک راس قرار دارد که مقدار آن برابر با شمارنده‌ی درخواست داده‌ی سر Queue پاد مربوطه است. فرآیند خواندن از پادها به این نحو است که Leader بعد از گرفتن درخواست، از پاد راس اول Heap درخواست گرفتن داده‌ی سر Queue آن را می‌کند و آن را به عنوان خروجی به کاربر برمی‌گرداند. همچنین پاد مبدا در کنار این داده، شمارنده‌ی درخواست سر جدید Queue خود را نیز ارسال می‌کند تا Leader با توجه به آن، داده‌های داخل Heap خود را بروزرسانی می‌کند. از آن‌جایی که از هر داده ۲ نسخه داریم، این فرآیند باید برای پاد مربوط به راس اول Heap جدید نیز به عنوان دارنده‌ی دوم داده‌ی خوانده شده، دوباره تکرار شود.

در صورت پایین آمدن هرکدام از پادهای سیستم، Leader به تمام پادهای دیگر درخواست می‌کند که تمام اطلاعاتی که از پاد از دست رفته دارند را برای آن ارسال کنند تا با توجه به آن‌ها، اطلاعات داخل پاد از دست رفته را بازیابی کند. اگر خود پاد Leader نیز از کار بیافتد، یک Leader جدید توسط سیستم انتخاب می‌شود. سپس برای بازیابی Heap پاد Leader و شمارنده‌ی پیام آن، هر پاد آخرین شمارنده‌ی درخواست داده‌های داخل Queue خود، داده‌های مربوط به پاد Leader سابق (برای بازیابی پاد از دست رفته توسط Leader جدید) و شمارنده‌ی درخواست سر Queue اش را به هسته گزارش می‌دهد تا به کمک آن کارهای مربوطه انجام شود.

در حین فرآیند اضافه کردن داده به پادها نیز در صورت پایین بودن پاد انتخاب شده، آن پاد بازیابی می‌شود و پاد جدیدی برای اضافه کردن داده در آن لحظه انتخاب می‌شود. همچنین در صورت پایین بودن پاد خواسته شده در فرآیند خواندن، آن پاد بازیابی شده و راس مربوط به آن از داخل Heap حذف می‌شود و پاد مربوط به راس اول Heap جدید به عنوان دارنده‌ی داده معرفی می‌شود.



Leader باید همواره پادهای سالم خوشه را در دسترس داشته باشد تا در فرآیند انتخاب و فراخوانی مشکلی برای او پیش نیاید (این موضوع می‌تواند هم به صورت بررسی لحظه‌ای تمام پادها انجام شود و هم می‌تواند با ذخیره‌سازی یک داده‌ساختار LinkedList از پادهای سالم در مرتبه‌ی زمانی بهتر انجام شود). همچنین هرکدام از پادها نیز باید داده‌های مربوط به یک پاد خاص که به آن‌ها نیز داده شده است را در دسترس داشته باشند (این موضوع نیز می‌تواند هم به صورت Hashing به کمک آرایه و هم به کمک یک Dictionary از داده‌های مربوط به هر پاد در مرتبه‌ی زمانی بهتر انجام شود). اطلاعات داخل Queue هر پاد را نیز می‌توان از یک میزانی بزرگ‌تر به صورت Chunk شده در دیسک ذخیره کرد تا مشکل پر شدن پادها را نیز نداشته باشیم.

ارزیابی مالی

این ارزیابی با توجه به روش گفته شده در تعرفه‌ی نرخ پایه خدمات فنی ـ تخصصی انفورماتیک حساب شده است. ابتدا f_1 و سپس f_2 را محاسبه می‌کنیم. برای محاسبه‌ی f_1، باید سه پارامتر C_1، C_2 و C_3 را برای شرکت خود محاسبه کنیم. از آنجایی که شرکت استارتاپی بدون رتبه حساب می‌شود، $C_1 = 1 + \frac{1}{1+1} = 1.5$ خواهد بود. همچنین چون شرکت نرم‌افزاری است $C_2 = 1.2$ و در نهایت $C_3 = 1$. پس داریم $f_1 = C_1 \times C_2 \times C_3 = 1.8$. سپس برای محاسبه‌ی f_2، نیاز است تخمین بزنیم که ما در ابتدا به چه مقدار نیروی انسانی نیاز داریم. برای نیروی انسانی مورد نیاز برای ایجاد نسخه اولیه این محصول، به ۵ نفر متشکل از یک Product Manager، یک Senior Developer، دو Junior Developer و همینطور یک Administrator برای مدیریت محصول نیاز خواهیم داشت. حال با توجه به داده‌های موجود در فایلی که در اختیارمان گذاشته شد، پارامترهای زیر را محاسبه می‌کنیم.

	N	P_1	P_2	P_3	P_4	$f_2 = P_1P_2P_3P_4$
Product Manager	1	2.16	1.33	1.1	1	3.16
Senior Developer	1	1.17	1.33	1.1	1	1.71
Junior Developer	2	1.17	1.11	1.1	1	1.43
Administrator	1	1.17	1	1.1	1	1.29

با توجه به جدول بالا حداقل هزینه ماهانه‌ی شرکت در این مورد 10.8 میلیون تومان خواهد بود. حالا وقت آن است که هزینه‌ی پرسنل شرکت برای کل پروژه را برآورد کنیم. فرض می‌کنیم هر کدام از نیروها ۱۲۰ ساعت وقت برای این پروژه صرف می‌کنند.

می‌دانیم فرمول برآورد هزینه‌ی پرسنل، $B = s \times \sum_{i=1}^{N} t_i \times f_{1_i} \times f_{2_i}$ خواهد بود. از آنجایی که می‌دانیم، $s = 1.547 \times 10^6\ (R)$ و $t_i = 120\ (h)$ و $f_{1_i} = 1.8$ پس خواهیم داشت:

$$B = s \times 1.8 \times 120(3.16 + 1.71 + 2 \times 1.43 + 1.29) = 3.01 \times 10^8\ (T)$$

دقت کنید که این مبلغ به تومان و برای کل پروژه است.

حالا به قسمت تخمین هزینه‌ی سخت‌افزاری می‌رسیم. فرض کنید برای استفاده از CPU از Queuing Theorem استفاده می‌کنیم. در نظر می‌گیریم که Producer و Consumer از توزیع نمایی با نرخ به ترتیب $\lambda_{p\cdot}$ و $\lambda_{c\cdot}$ پیروی می‌کنند. در این صورت امیدریاضی طول صف ما برابر خواهد بود با:

$$P_n = (1 - \frac{\lambda_{p\cdot}}{\lambda_{c\cdot}})(\frac{\lambda_{p\cdot}}{\lambda_{c\cdot}}) \implies L = \Sigma(n-1)P_n = \frac{\lambda_{p\cdot}}{\lambda_{c\cdot} - \lambda_{p\cdot}} + \frac{\lambda_{c\cdot}}{\lambda_{p\cdot}}$$

پس متوسط زمان مصرف CPU برابر خواهد بود با:

$$T = \lambda_{p\cdot} t_p + \lambda_{c\cdot} t_c$$

حالا Memory و Storage را با روش زیر تخمین می‌زنیم. دقت کنید که مقادیر M_0 و S_0 تخمینی از یک سیستم حداقلی لینوکسی هستند و مقدار جمع شده با آن‌ها تخمین استفاده‌ی سرویس ما است. همینطور L همان طول صف متوسط و ضرب ۲ بخاطر Replication است.

$$M = L \times (m_{struct} + m_{reserve}) + M_0 \ , M_0 \approx 3GB$$

$$S = 2 \times L(s_{reserve} + s_{value}) + S_0 \ , S_0 \approx 30GB$$

اگر یک سیستم بسیار Heavy Load در نظر بگیریم احتمالا نرخ‌های معرفی شده در ابتدای صفحه تقریبا $\lambda_{p\cdot} = 10^4 \ \frac{req}{s}$ و $\lambda_{c\cdot} = 10^5 \ \frac{req}{s}$ فرض منطقی ای خواهد بود.

$$m_{struct} = 0.5 \ KB \ , m_{reserve} = 1 \ KB$$

$$s_{reserve} = 100 \ KB \ , s_{value} = 1 \ MB$$

$$L = 10 + \frac{1}{9} \approx 10 \ , M = 15 \ KB \ , S = 2.2 \ MB$$

$$t_p = t_c = 80\mu s \implies T = 80 \times 10 - 6 \times 1.1 \times 10^5 = 8.8s \implies 8 \text{ cores}$$

پس یک سرور با ۹ هسته، ۴ گیگ رم و ۵۰ گیگ حافظه احتمالا جوابگوی خواسته‌های ما خواهد بود. برآورد هزینه‌ی یک سرور ابری با این خصوصیات، ماهانه T 831000 خواهد بود.

نتیجه‌گیری

معماری سیستم صف پیام‌رسانی‌ای که معرفی کردیم، نتیجه‌ی دقت و برنامه‌ریزی هوشمندانه برای ساخت یک راه‌کار جامع است که هم پاسخگوی نیازهای مدرن است و هم قادر به تطابق با تغییرات آینده. از طراحی مقیاس‌پذیر و تحمل‌پذیر برای تضمین کیفیت و پایداری خدمات در مواجهه با شکست‌های سیستمی گرفته تا استفاده از تکنولوژی‌های برتر برای بهبود قابلیت اطمینان و کارایی سیستم، هر انتخاب مهندسی شده به نحوی است که به کارآمدی و امنیت بیشتر سیستم کمک کند.

استفاده از GoLang به پاسخگویی سریع سیستم در پردازش‌های موازی و جنبه‌های شبکه‌ای کمک می‌کند، در حالی‌که Docker و Kubernetes چابکی و کارآمدی مدیریت سیستم‌های متعدد را فراهم می‌آورند. علاوه بر این، تعهد ما به امنیت از طریق استفاده‌ی سیستماتیک از TLS/SSL در همه ارتباطات و تکیه بر روندهای CI/CD برای اتوماسیون بخش‌های طراحی، تست و استقرار، باعث افزایش امنیت و کارایی فرایندها می‌شود.

به طور خلاصه، معماری سیستم ما به گونه‌ای طراحی شده است تا از مقیاس‌پذیری، تحمل‌پذیری خطا، الگوریتم اجماع بی‌رهبر، و سازوکارهای مانیتورینگ پیشرفته به منظور تأمین استقراری بادوام و مستحکم که به طور مداوم خود را بهبود می‌بخشد و در برابر تغییرات آینده پاسخگو است، بهره ببرد. این سیستم طوری ساخته شده تا به ارائه‌ی تجربه‌ای قابل اعتماد و ساده برای کاربران پایانی بپردازد، ضمن اینکه اطمینان می‌دهیم که در هر مرحله، امنیت و کارایی را به‌عنوان اولویت‌های اصلی حفظ کرده‌ایم.