میکروسرویس چیست؟

وقتی یک نرم‌افزار به‌صورت یکپارچه و بزرگ طراحی می‌شود، هر تغییر یا توسعه می‌تواند پیچیدگی و ریسک را افزایش دهد. در چنین شرایطی، معماری میکروسرویس به‌عنوان راهکاری مدرن مطرح می‌شود که سیستم را به مجموعه‌ای از سرویس‌های کوچک، مستقل و قابل توسعه جداگانه تقسیم می‌کند.

میکروسرویس با ایجاد استقلال در توسعه و استقرار بخش‌های مختلف، امکان مقیاس‌پذیری بهتر و مدیریت ساده‌تر سیستم‌های پیچیده را فراهم می‌کند. در این مقاله از *نیک آموز*، به بررسی مفهوم میکرو سرویس و مزایا و چالش‌های آن خواهیم پرداخت.

تاریخچه میکروسرویس: دنیا قبل از میکروسرویس به چه صورت بود؟

میکرو سرویس به‌عنوان یک رویکرد مدرن در طراحی نرم‌افزار، ناگهان ظهور نکرده است، بلکه حاصل تکامل تدریجی معماری‌های سرویس‌گرا است.

ریشه‌ها

در دهه ۲۰۰۰، معماری SOA (Service-Oriented Architecture) مطرح شد که سیستم‌ها را به سرویس‌های مستقل تقسیم می‌کرد. با این حال، این سرویس‌ها اغلب بزرگ و پیچیده بودند و انعطاف‌پذیری لازم برای توسعه سریع را فراهم نمی‌کردند.

شکل‌گیری میکرو سرویس

اوایل دهه ۲۰۱۰، شرکت‌هایی مانند Netflix و Amazon برای حل مشکلات مقیاس‌پذیری و توسعه سریع، سیستم‌های یکپارچه خود را به مجموعه‌ای از سرویس‌های کوچک، مستقل و قابل استقرار جداگانه تقسیم کردند. در سال ۲۰۱۴، Martin Fowler و James Lewis با انتشار مقاله‌ای رسمی، اصطلاح «میکرو سرویس» را تثبیت و چارچوب فکری آن را توضیح دادند.

دلیل فراگیری

میکرو سرویس به دلیل نیاز به توسعه و استقرار سریع، مقیاس‌پذیری بهتر و مدیریت ساده‌تر سیستم‌های پیچیده، به‌سرعت در صنعت نرم‌افزار پذیرفته شد.

این نگاه تاریخی نشان می‌دهد که میکرو سرویس نه یک ترند زودگذر، بلکه پاسخی عملی به نیازهای واقعی توسعه نرم‌افزارهای بزرگ و پیچیده است.

چالش‌ها و محدودیت‌های میکروسرویس

از قدیم گفتند: "هیچ ارزانی بی حکمت نیست" اگر بخواهیم به زبان برنامه نویسی ترجمه کنیم می‌شود "هیچ سادگی بدون محدودیت نیست." هر نرم افزار و پروژه‌ای در صورتی که موفق بشود قطعا تمایل به رشد دارد و موفقیت بیشتر پروژه موجب رشد بیشتر پروژه خواهد شد. در نهایت بعد از مدتی پروژه کوچک و ساده ما تبدیل به هیولایی بزرگ و وحشتناک خواهد شد.

اما ممکن است به این فکر کنید که زیاد شدن حجم کدها در طول دوران توسعه نرم افزار و تغییرات آن چه مشکلی می‌تواند داشته باشد؟ در ادامه به چند مورد از این مشکلات خواهیم پرداخت.

مشکل اول: پیچیدگی بیش از حد این برنامه‌ها

با بزرگ و پیچیده شدن نرم افزار تیم توسعه شما با مشکلات فراوانی دست و پنجه نرم خواهد کرد. هر تصمیمی برای تغییر در برنامه یا ایجاد سریع یک ویژگی و ارائه آن احتمالا به بن بست خواهد رسید. بزرگترین مشکل این نرم افزارها پیچیدگی بیش از حد این برنامه‌ها است. معمولا نرم افزارها در طول سالیان آنقدر بزرگ و پیچیده می‌شوند که درک درست و دقیق عملکرد آن‌ها برای یک توسعه دهنده نرم افزار غیر ممکن می‌شود.

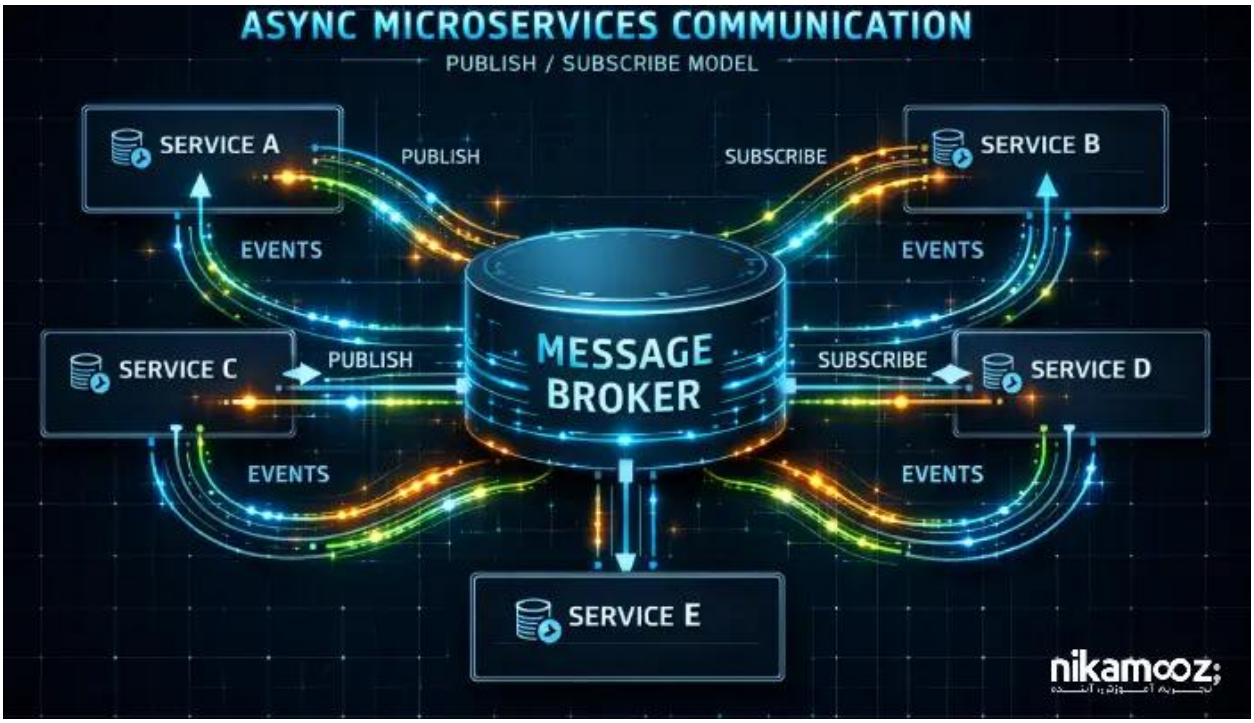
در نتیجه این عدم توانایی شناخت درست و صحیح باعث می‌شود رفع خطاهای موجود یا اضافه کردن یک ویژگی جدید هم بسیار سخت و هم بسیار پیچیده باشد. نکته اصلی در این شرایط این است که با گذشت زمان این مشکل به شکل نمایی بیشتر و بیشتر می‌شود. هر چقدر فهم کد سخت‌تر بشود احتمال اشتباه بیشتر و احتمال پیدا کردن اشتباه‌ها کمتر و توان رفع آنها هم کمتر می‌شود، در نهایت به سورس کدی خواهیم رسید که اصطلاحا به اون big ball of mud می‌گوییم.

مشکل دوم: پایین آمدن سرعت توسعه نرم افزار

سورس کد حجیم می‌تواند باعث پایین آمدن سرعت توسعه نرم افزار بشود. هر وقت تصمیم به بازکردن پروژه بگیرید دقایق زیادی طول خواهد کشید تا IDE شما باز بشود و آماده به کار باشد. در کنار این شرایط احتمالا این سیستم عظیم بهره‌وری کل بستر شما رو هم پایین خواهد آورد.

مشکل سوم: عدم توانایی در انتشار بهبودها و توسعه‌های کوچک

احتمالا با داشتن یک سیستم بزرگ شما با عدم توانایی در انتشار بهبودها و توسعه‌های کوچک روبرو خواهید شد. مسلما سیستمی که باز کردن آن چند دقیقه طول بکشد، Build شدنش بعضا ۱ ساعت طول خواهد کشید و روال نصب راحتی هم نخواهد داشت. همچنین با توجه به اینکه در زمان نصب احتمالا سیستم از دسترس خارج خواهد بود و قاعدتا از دسترس خارج کردن یک سیستم عظیم به خاطر یک تغییر کوچک یا رفع باگ احتمالی جزئی، مقرون به صرفه نیست به همین خاطر نصب نسخه‌های جدید با فاصله‌های زمانی طولانی انجام خواهد شد.



مشکل چهارم: عدم استفاده بهینه از منابع

عدم استفاده بهینه از منابع یکی دیگر از مشکلات اساسی توسعه نرم افزار به این شکل هست. در نرم افزارهای متفاوت نیازهای متفاوتی وجود دارد. ممکن است بخشی از برنامه مصرف RAM زیادی داشته باشه و بخشی دیگر هم مصرف CPU اما در این روش امکان تخصیص یک منبع خاص به بخش خاصی که نیاز به آن منبع دارد، وجود نخواهد داشت. در نتیجه هنگامی که بخشی با مصرف CPU زیاد ارتقا داده بشود این امکان در اختیار همه بخش‌ها قرار خواهد گرفت و برای همه قسمت ها امکان استفاده از این منبع، بی رویه و ناصحیح وجود خواهد داشت.

مشکل پنجم: انتشار مشکلات

مشکلی که توسعه به روش Monolith به همراه داره انتشار مشکلات هست. کل برنامه در یک سیستم و در قالب یک پروسه اجرا خواهد شد. پس اگر ایرادی در هر یک از قسمت‌های برنامه به وجود بیاید، تمامی قسمت‌های برنامه از کار خواهند افتاد و کل برنامه تا زمان رفع مشکل و نصب نسخه جدید از دسترس خارج خواهد بود. البته در این شرایط باید امیدوار بود که نصب نسخه جدید موجب ایجاد مشکلات جدید در سیستم نشود.

مشکل ششم: عدم توانایی در به کارگیری ابزارها و تکنولوژی‌های جدید

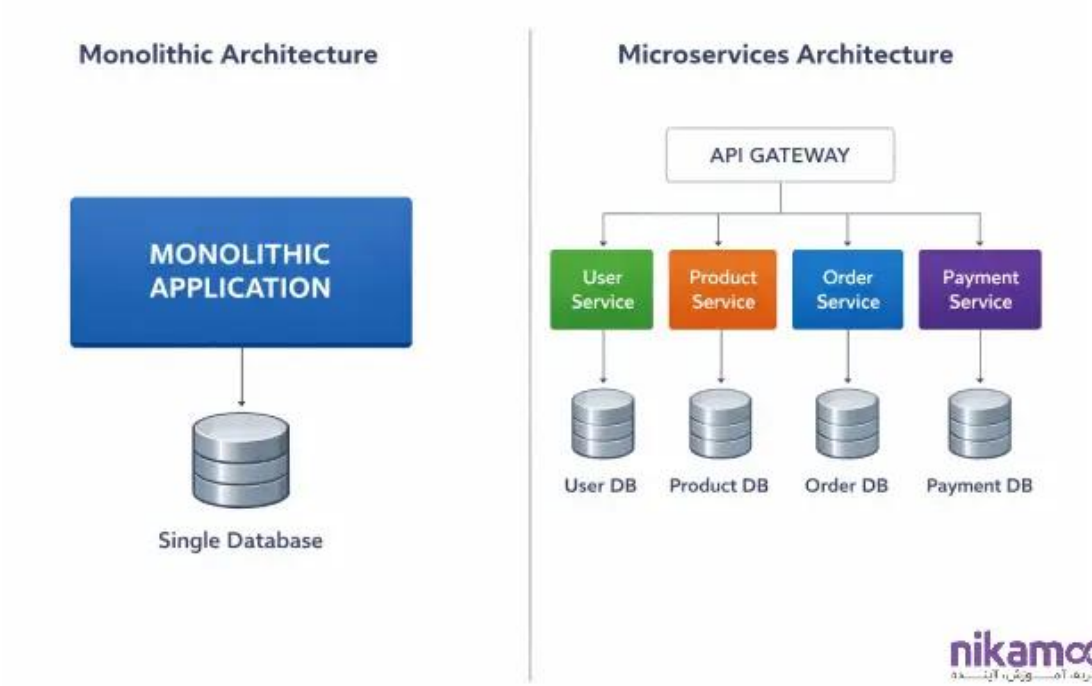
عدم توانایی در به کارگیری ابزارها و تکنولوژی‌های جدید هم یکی دیگر از ایرادات این روش توسعه نرم افزار هست. در شرایطی که شما یک نرم افزار بزرگ و پیچیده را سالها توسعه داده‌اید احتمالا وقتی با یک ابزار،

تکنولوژی یا فریمورک جدید مواجه بشوید به جز حسرت امکانات موجود کار دیگری از دستتان بر نمیاد. معمولا امکان اینکه سالها تلاش تیم دور ریخته بشود نه پذیرفته می‌شود نه قابل اجرا هست.

در حال حاضر در یکی از شرکت‌های بزرگ درگیر مشاوره در یک پروژه ERP هستم که سال‌ها پیش و با تکنولوژی سال ۲۰۰۵ توسعه داده شده است در این ابزار از ۲۰ Net Framework. استفاده شده و برای توسعه وب هم از ASP.NET Web Form استفاده شده است و اینقدر سیستم بزرگ و پیچیده شده که در طی این سال‌ها کسی جرات دست زدن به سیستم و تغییر تکنولوژی را نداشته است.

در صورتی که خودتان جای یکی از مدیران پروژه و شرکت بگذارید چقدر احتمال دارد قبول کنید که ثمره ۱۵ سال توسعه یک تیم برنامه نویسی دور ریخته بشود و یک پروژه جدید استارت زده شود؟ به نظرتان در این ۱۵ سال که یک تیم ۱۵-۲۰ نفره به طور میانگین درگیر این پروژه بوده است چقدر هزینه تولید همچین ابزاری شده است؟ آیا تغییر تکنولوژی با این شرایط امکان پذیر هست؟

خوب به نظر میرسه کم کم داریم به پایان دنیا نزدیک می‌شیم. اما واقعا راه حلی برای این مشکلات نیست؟



مزایای میکروسرویس

موضوع / ویژگی	توضیح
فلسفه میکروسرویس	مشابه روش تقسیم و غلبه، میکروسرویس‌ها برنامه‌های بزرگ (Monolith) را به مینی‌اپلیکیشن‌های کوچک با مدیریت و نگهداری ساده‌تر تقسیم می‌کنند. این کار فهم، توسعه و رفع مشکلات را ساده‌تر می‌کند.
انعطاف‌پذیری (Flexibility)	هر میکروسرویس مسئولیت مشخصی دارد و مستقل توسعه می‌شود. شرکت‌ها می‌توانند از ابزارها و فریم‌ورک‌های مناسب هر سرویس استفاده کنند، مثل انتخاب گراف دیتابیس برای داده‌های گرافی و زبان برنامه‌نویسی خاص برای هر سرویس.
نصب و انتشار مستقل	هر میکروسرویس به‌صورت مستقل قابل نصب و استقرار است؛ رفع باگ در یک سرویس نیازمند توقف کل سیستم نیست.
استفاده بهینه از منابع	هر مینی‌اپلیکیشن منابع خود را دریافت می‌کند و در صورت نیاز، مقیاس‌پذیری عمودی یا افقی برای آن قابل اعمال است.
مقیاس‌پذیری (Scalability)	سیستم می‌تواند منابع سرویس‌ها را افزایش یا کاهش دهد. مقیاس‌پذیری عمودی با افزایش سخت‌افزار و مقیاس‌پذیری افقی با اجرای چندین نسخه از میکروسرویس روی سرورهای مختلف حاصل می‌شود. ابزارهای مدیریت ترافیک مانند Kubernetes و Istio این فرآیند را ساده می‌کنند.
توسعه موازی (Parallel Development)	تیم‌های مستقل می‌توانند همزمان روی میکروسرویس‌های مختلف کار کنند. این کار سرعت توسعه، مدیریت تغییرات و پاسخگویی به نیازهای کسب‌وکار را افزایش می‌دهد.
استقلال میکروسرویس‌ها	هر سرویس دارای دیتابیس، منطق کسب‌وکار و تیم توسعه مستقل خود است. تغییرات و اصلاحات در یک سرویس تأثیر محدودی بر بقیه سیستم دارد.
مدیریت پیچیدگی	سیستم قابل تجزیه است، هر میکروسرویس مسئولیت مشخص دارد و از تکنولوژی‌های متفاوت استفاده می‌کند. مدل‌های داده کوچک و مستقل، مدیریت و اصلاح سیستم را آسان می‌کنند.
مثال‌های کاربردی	**میکروسرویس مدیریت کاربران:** تغییرات در حساب کاربری بدون تأثیر روی سایر سرویس‌ها. **میکروسرویس پرداخت:** اضافه کردن روش پرداخت یا بهبود امنیت

توضیح	موضوع / ویژگی
تراکنش‌ها مستقل از سایر سرویس‌ها. **میکروسرویس ارسال ایمیل:** تغییر قالب یا متن ایمیل بدون تأثیر روی سایر بخش‌ها.	
مینی‌اپلیکیشن‌ها با استفاده از API با هم تعامل می‌کنند. مدیریت تعداد زیاد آدرس‌ها و ارتباطات بین سرویس‌ها با استفاده از API Gateway انجام می‌شود.	ارتباط بین سرویس‌ها

معایب میکروسرویس

توضیح	معایب / چالش
نام «میکروسرویس» روی کوچک بودن سرویس‌ها تاکید دارد، اما تعریف دقیق اندازه سرویس چالش‌برانگیز است. کوچک بودن سرویس وسیله‌ای برای رسیدن به هدف بزرگ است، نه هدف اصلی.	تعیین مقیاس سرویس‌ها
ارتباط بین میکروسرویس‌ها پیچیده‌تر از Monolith است. در Monolith می‌توان از کلاس‌ها و توابع داخلی استفاده کرد، اما در سیستم توزیع‌شده نیاز به API و پروتکل‌های شبکه است.	پیچیدگی ارتباط بین سرویس‌ها
سرویس‌ها روی محیط‌ها و سرورهای مختلف اجرا می‌شوند و هماهنگی بین آن‌ها پیچیده است، به‌خصوص وقتی تغییرات بین سرویس‌ها یا نسخه‌های مختلف داده لازم باشد.	مدیریت توزیع و هماهنگی
تست سیستم توزیع‌شده دشوارتر از سیستم یکپارچه است. شبیه‌سازی تعامل بین سرویس‌ها و مدیریت خطاها پیچیده است.	تست و عیب‌یابی
هر سرویس منابع و دیتابیس مستقل دارد. مدیریت صدها سرویس می‌تواند هزینه عملیاتی و زیرساختی زیادی ایجاد کند.	هزینه زیرساخت
تعامل بین سرویس‌ها از طریق شبکه انجام می‌شود، بنابراین latency و مصرف پهنای باند افزایش می‌یابد.	تأخیر و هزینه ارتباط بین سرویس‌ها

معایب / چالش	توضیح
پیچیدگی استقرار و نصب	نصب و راه‌اندازی مستقل سرویس‌ها یک مزیت است، اما وقتی سیستم بیش از ۱۰۰ سرویس دارد، نصب، تنظیم ارتباطات و مدیریت نسخه‌ها زمان‌بر و پرخطا خواهد بود.
نیاز به تیم‌های متخصص	مهارت درDevOps، مدیریت ترافیک، ابزارهای ابری و پایگاه‌های داده توزیع‌شده لازم است.
مدیریت داده و تراکنش‌ها	هر سرویس دیتابیس اختصاصی دارد. مدیریت داده‌ها بین سرویس‌ها، تراکنش‌های توزیع‌شده و همگام‌سازی نسخه‌ها چالش‌برانگیز است.
پیچیدگی امنیتی	هر سرویس ممکن است سطوح دسترسی و داده‌های خاص خود را داشته باشد؛ مدیریت احراز هویت و امنیت شبکه گسترده‌تر و پیچیده‌تر است.

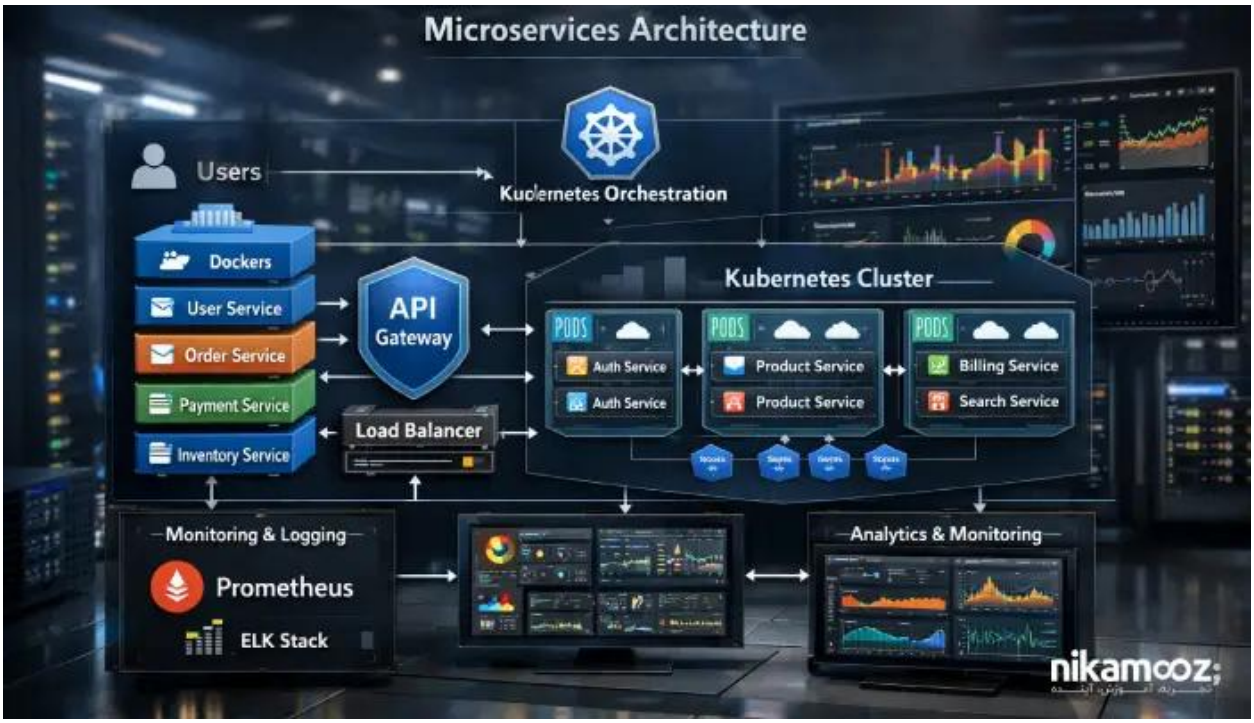
هدف از معماری میکروسرویس چیست؟

معماری میکروسرویس به نوعی پاسخی به چالش‌ها و نیازهای جدید سازمان‌ها در دنیای دیجیتال است. این معماری با ارائه‌ی سرویس‌های کوچک و مستقل که با هم تعامل می‌کنند، سازمان‌ها را قادر می‌سازد تا به صورت موثر و مستقل توسعه و تغییر دهند.

تعامل و استقلال:

یکی از اصلی‌ترین اهداف میکروسرویس، ارائه‌ی سرویس‌هایی است که مستقل و قابل مدیریت هستند. این موضوع باعث می‌شود تیم‌های توسعه بتوانند با سرعت بیشتری واحدهای کد را توسعه، تست و انتشار دهند.

قابلیت اطمینان:



معماری میکروسرویس امکان می‌دهد که هر سرویس به صورت جداگانه نظارت و بهبود یابد. این ویژگی باعث می‌شود در صورت بروز مشکل در یک سرویس، سایر سرویس‌ها تحت تاثیر قرار نگیرند.

مقیاس‌پذیری:

سرویس‌ها می‌توانند به صورت مستقل از یکدیگر مقیاس‌بندی شوند، که این امکان را می‌دهد تا منابع را برای سرویس‌هایی که نیاز به بار بیشتری دارند، اختصاص داد.

آنچه میکروسرویس می‌خواهد ارائه دهد، سازمان‌هایی پویا، قابل تغییر و با قابلیت پاسخ‌گویی بالا به نیازهای مشتری است.

اطلاعات بیشتر: آموزش معماری میکروسرویس

چه زمانی از معماری میکروسرویس استفاده کنیم؟

در دهه‌ی اخیر، معماری میکروسرویس به عنوان یکی از رویکردهای پرطرفدار در طراحی نرم‌افزار مطرح شده است. اما چگونه می‌توانیم تصمیم بگیریم که این معماری برای پروژه‌ی ما مناسب است یا خیر؟

پیچیدگی سیستم:

اگر سیستم شما پیچیده و گسترده است و شما نیاز به تقسیم‌بندی واحدهای مختلف به صورت مستقل دارید، میکروسرویس می‌تواند گزینه‌ی مناسبی باشد.

نیاز به توسعه موازی:

در پروژه‌هایی که تیم‌های متعدد باید به صورت همزمان روی ویژگی‌های مختلف کار کنند، میکروسرویس‌ها می‌توانند فرآیند را ساده‌تر و سریع‌تر کنند.

زبان‌ها و فناوری‌های مختلف:

اگر نیاز به استفاده از زبان‌ها و فناوری‌های مختلف در یک پروژه دارید، میکروسرویس این امکان را می‌دهد که هر سرویس با زبان و فناوری مناسب خود نوشته شود.

تضمین عملکرد:

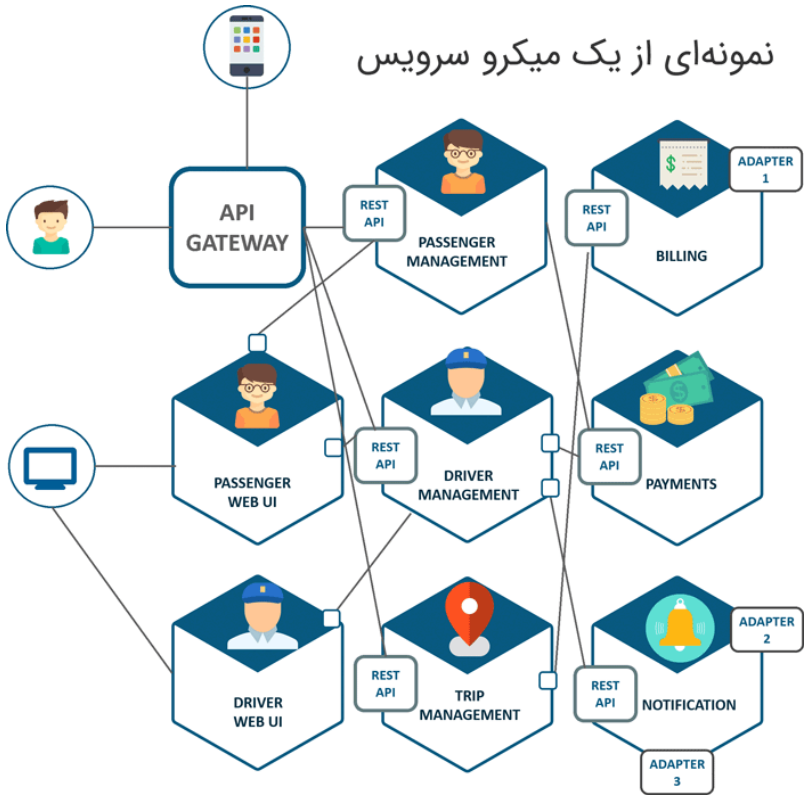
در مواردی که بخش‌های خاصی از برنامه نیاز به منابع بیشتری دارند، می‌توان با استفاده از معماری میکروسرویس منابع را به صورت موازی و مستقل افزایش داد.

توزیع جغرافیایی:

اگر تیم‌های شما در مکان‌های مختلف جغرافیایی واقع شده‌اند، استفاده از میکروسرویس می‌تواند هماهنگی بین تیم‌ها را آسان‌تر کند.

نیاز به ارتقاء مستقل:

اگر می‌خواهید قسمت‌هایی از برنامه خود را بدون تأثیر گذاری بر سایر قسمت‌ها به‌روز کنید، معماری میکروسرویس این امکان را فراهم می‌کند.



بزرگ‌ترین استفاده‌کنندگان معماری میکروسرویس در دنیا

معماری میکروسرویس از آنجا که اجازه می‌دهد سیستم‌ها و اپلیکیشن‌ها را به صورت مستقل و مقیاس‌پذیر طراحی کنیم، در سال‌های اخیر به یکی از محبوب‌ترین معماری‌ها در دنیای نرم‌افزار تبدیل شده است. چندین شرکت بزرگ در جهان از این معماری استفاده می‌کنند تا بهره‌وری، سرعت و کیفیت خدمات خود را به حداکثر برسانند. در زیر به برخی از این شرکت‌ها اشاره می‌کنیم:

- **Netflix**: یکی از اولین و بزرگ‌ترین استفاده‌کنندگان معماری میکروسرویس، Netflix است. با توجه به میلیون‌ها کاربر در سراسر دنیا، این شرکت از میکروسرویس‌ها برای پشتیبانی از تقاضای روزانه خود استفاده می‌کند.
- Amazon: **Amazon** با تجزیه سیستم‌های خود به میکروسرویس‌ها، توانمندی مدیریت ترافیک بالایی را کسب کرده و به کاربران خود خدمات مستمری ارائه می‌دهد.
- **Uber**: با توجه به نیاز به سرعت و پاسخ‌گویی فوری، اوبر سیستم‌های خود را بر مبنای میکروسرویس طراحی کرده است.
- **Spotify**: این شرکت موسیقی با استفاده از معماری میکروسرویس، توانمندی توسعه سریع و مقیاس‌پذیری خود را افزایش داده است.

این شرکت‌ها فقط نمونه‌هایی از بزرگ‌ترین‌ها هستند و بسیاری دیگر نیز در دنیا از معماری میکروسرویس بهره می‌برند. استفاده از این معماری به آن‌ها امکان داده تا به سرعت و با کیفیت به نیازهای کاربران خود پاسخ دهند.

فناوری‌ها و نرم‌افزارهای مورد نیاز برای پیاده‌سازی میکروسرویس‌ها

دسته فناوری / ابزار	مثال‌ها و توضیح
زبان‌های برنامه‌نویسی	Java, Python, Node.js, Go, .NET Core. انتخاب زبان وابسته به نیازها و تجربیات تیم توسعه است.
کانتینرها (Container)	Docker برای اجرای میکروسرویس‌ها در محیط‌های جداگانه و یکسان.
مدیریت کانتینر	Kubernetes برای مدیریت، مقیاس‌دهی و اجرای سرویس‌ها در محیط‌های کانتینری.
API Gateway	نقطه ورود واحد برای درخواست‌های کاربر به میکروسرویس‌ها، مانند AWS API Gateway یا Kong.
Service Mesh	Istio یا Linkerd برای مدیریت ترافیک بین سرویس‌ها، کشف خدمات و امنیت ارتباطات.
دیتابیس‌ها	هر میکروسرویس می‌تواند از دیتابیس مخصوص خود استفاده کند، مانند PostgreSQL, MongoDB, Cassandra.
ابزارهای مانیتورینگ و لاگ	Prometheus, Grafana, ELK Stack (Elasticsearch, Logstash, Kibana), Zipkin برای جمع‌آوری داده‌ها و آنالیز سیستم‌های میکروسرویسی.
CI/CD	Jenkins, GitLab CI, Travis CI برای اتوماسیون فرآیندهای توسعه و تحویل میکروسرویس‌ها.

API Gateway چیست؟

درادامه به تعریف این سوال میپردازیم API Gateway: یک نقطه ورود واحد برای ارتباط Clientها با میکروسرویس‌ها است که تمام پیچیدگی‌های تعامل بین سرویس‌ها را پشت خود مخفی می‌کند. به جای ارسال چندین درخواست به سرویس‌های مختلف برای نمایش یک صفحه، مثل جزئیات محصول در یک

فروشگاه آنلاین، Client تنها با یک API تعامل دارد و Gateway مسئول مسیریابی، ترکیب نتایج، مدیریت پروتکل‌ها و ارائه API اختصاصی برای هر Client است. این روش بهره‌وری و UX را افزایش می‌دهد، امکان Refactoring ساده‌تر سرویس‌ها را فراهم می‌کند و تعداد درخواست‌ها را کاهش می‌دهد، اما ایجاد و نگهداری Gateway نیازمند منابع، توسعه اختصاصی برای هر Client، مدیریت خطاها، یافتن آدرس سرویس‌ها و اطمینان از مقیاس‌پذیری و عملکرد بالا است. استفاده از مدل برنامه‌نویسی Reactive و پشتیبانی از روش‌های تعامل مختلف (Sync و Async) به بهینه‌سازی عملکرد Gateway کمک می‌کند و در مجموع، API Gateway راهکاری مؤثر برای ساده‌سازی و یکپارچه‌سازی دسترسی به میکروسرویس‌هاست.

ارتباط بین سرویس ها

توضیح	موضوع
Inter-Process Communication (IPC) بین سرویس‌ها به‌صورت یک‌به‌یک یا یک‌به‌چند و همزمان (Sync) یا ناهمزمان(Async)	نوع ارتباط
Request/Response، Notification، Pub/Sub، Request/Async Response	روش‌ها
تعریف دقیق APIها، رعایت API First و Robustness، مدیریت تغییرات به‌صورت backward compatible	اصول طراحی API
استفاده از Timeout، محدود کردن تعداد درخواست‌های بدون پاسخ، Circuit Breaker، Fallback	تحمل خطا (Fault Tolerance)
ارائه داده‌های کش شده یا جایگزین هنگام از دسترس خارج شدن سرویس‌ها	حفظ تجربه کاربری (UX)

تکنولوژی‌های ارتباطی

در میکروسرویس‌ها برای برقراری ارتباط بین سرویس‌ها و کلاینت‌ها تکنولوژی‌های متنوعی وجود دارد که بسته به نوع تعامل Sync یا Async و یک‌به‌یک یا یک‌به‌چند انتخاب می‌شوند؛ روش‌های Async معمولاً از Messaging استفاده می‌کنند که شامل ارسال پیام با Header و بدنه پیام از طریق کانال‌های Point-to-Point یا Publish-Subscribe است و مزایایی مانند جداسازی کامل کلاینت از سرویس، عدم نیاز به همزمانی سرویس و کلاینت و نمایان شدن ماهیت IPC دارد، اما پیچیدگی‌های عملیاتی و پیاده‌سازی و نیاز به نگهداری زیرساخت را نیز به همراه دارد. در روش‌های Sync، Request/Response رایج است که

با پروتکل‌هایی مانند REST یا Thrift پیاده‌سازی می‌شود؛ REST مبتنی بر Resourceها و HTTP است و مزایایی مثل سادگی، تست‌پذیری و سازگاری با زیرساخت‌ها دارد، اما محدودیت‌هایی چون نیاز به همزمانی کلاینت و سرویس، وابستگی به آدرس دقیق سرویس و پشتیبانی صرف ازRequest / Response دارد. انتخاب فرمت پیام‌ها نیز مهم است؛ قالب‌های متنی مانند JSON و XML خوانا اما کم‌بهره‌ور و قالب‌های باینری مثل Avro یا Protocol Buffer سریع و بهینه هستند، بنابراین بسته به نیازمندی‌های سیستم و شرایط عملیاتی باید فناوری و قالب مناسب ارتباط بین سرویس‌ها انتخاب شود.

آشنایی باService Discovery

معایب	مزایا	مثال/ابزار	توضیح	موضوع
نیاز به راهکار داینامیک برای یافتن آدرس‌ها	انعطاف‌پذیری لازم برای سیستم‌های مدرن	-	آدرس‌های استاتیک بی‌فایده‌اند؛ سرویس‌ها در cloud یا محیط توزیع‌شده دائم تغییر آدرس دارند.	**چالش آدرس‌دهی سرویس‌ها**
نیاز به در دسترس بودن دائم و به‌روزرسانی مداوم	ذخیره و مدیریت متمرکز آدرس سرویس‌ها، امکان کش کلاینت	Netflix Eureka, Etcd, Consul, ZooKeeper	دیتابیسی برای نگهداری آدرس تمام نمونه‌های سرویس‌ها به صورت داینامیک	Service Registry
وابستگی کامل کلاینت به Service Registry، نیاز به پیاده‌سازی برای هر کلاینت، پیچیدگی در مقیاس بزرگ	سادگی، امکان پیاده‌سازی الگوریتم توزیع بار اختصاصی	Netflix Eureka + Netflix Ribbon	کلاینت وظیفه query زدن به Service Registry و انتخاب نمونه سرویس با الگوریتم توزیع بار را دارد	Client-side Discovery
اضافه شدن لایه جدید (Proxy) به سیستم، وابستگی به سرور واسط	کلاینت ساده‌تر است، مدیریت توزیع بار متمرکز	AWS ELB, Kubernetes Proxy	کلاینت درخواست را به مسیریاب (Proxy) می‌فرستد، مسیریاب Service Registry را queryکرده و درخواست	Server-side Discovery

معایب	مزایا	مثال/ابزار	توضیح	موضوع
			را به سرویس مناسب می‌فرستد	
افزایش ترافیک ثبت وضعیت، نیاز به تنظیم مناسب بازه‌ها	اطمینان از زنده بودن سرویس‌ها، حذف خودکار سرویس‌های غیرفعال	Netflix Eureka: هر ۳۰ ثانیهPUT	بررسی وضعیت سرویس‌ها در بازه‌های زمانی مشخص	Heartbeat
سرویس‌ها وظایف اضافی دارند، پیاده‌سازی تکراری برای هر زبان / فریم‌ورک	سادگی، نیازی به ابزار خارجی نیست	Eureka Client	سرویس خودش مسئول ثبت، حذف و ارسال heartbeat است	Self-Registration
اضافه شدن سرویس جدید به سیستم، نیاز به نگهداری و مدیریت سرویس مرکزی	جداسازی سرویس‌ها از وظایف ثبت و مدیریت، خودکارسازی عملیات	Registrator	یک سرویس مرکزی (Service Registrar) مدیریت ثبت و حذف سرویس‌ها را انجام می‌دهد	Third-Party Registration

مدیریت داده‌ها در میکروسرویس

در میکروسرویس‌ها هر سرویس داده‌های خود را به صورت مستقل نگهداری می‌کند و دسترسی به داده‌های سرویس‌های دیگر تنها از طریق API امکان‌پذیر است، که مزایایی مثل استقلال، توزیع‌پذیری و استفاده از DB Engineهای مختلف (Polyglot Persistence) دارد اما چالش‌هایی مثل تراکنش‌های توزیع شده و همزمانی داده‌ها ایجاد می‌کند. برای حل این مشکلات معمولاً از Eventها استفاده می‌شود، به طوری که هر تغییر مهم در داده‌ها یک Event تولید می‌کند و سایر سرویس‌ها با دریافت آن داده‌های داخلی خود را به‌روزرسانی می‌کنند، روشی که eventual consistency، materialized view و تراکنش‌های چندمرحله‌ای را ممکن می‌کند اما پیچیدگی فنی، نیاز به rollback و احتمال دریافت دوباره Eventها از معایب آن است. برای حفظ atomicity می‌توان از جدول Event در تراکنش محلی، Transaction Log یا Event Sourcing استفاده کرد، به طوری که در Event Sourcing وقایع رخداده

روی Entity ها ذخیره می‌شوند و وضعیت نهایی از روی این وقایع بازسازی می‌شود؛ این روش تاریخچه کامل داده‌ها را فراهم می‌کند اما پیچیدگی پیاده‌سازی، زمان‌بر بودن پردازش و دشواری جستجو از محدودیت‌های آن است و معمولاً با CQRS ترکیب می‌شود.

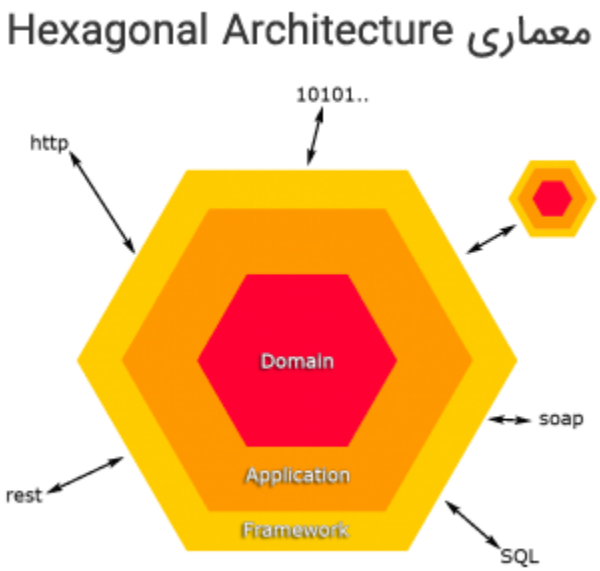
آشنایی با روش‌های انتشار در میکروسرویس‌ها

روش انتشار	توضیح	مزایا	معایب
نصب چند سرویس مختلف به ازای هر میزبان	چندین سرویس روی یک سرور فیزیکی یا مجازی نصب می‌شوند و هر سرویس روی IP و پورت مشخص اجرا می‌شود.	استفاده بهینه از منابع، سرعت بالای نصب و راه‌اندازی، سادگی کپی و اجرای برنامه‌ها	نبود محدودیت منابع، یک سرویس می‌تواند منابع سایر سرویس‌ها را مصرف کند، نیاز به دانش جزئیات نصب و راه‌اندازی برای تیم زیرساخت، افزایش ریسک خطا
نصب یک سرویس به ازای هر ماشین مجازی	هر سرویس روی یک ماشین مجازی مجزا نصب می‌شود و نسخه‌ها از Image ماشین مجازی ایجاد می‌شوند.	ایزوله بودن کامل سرویس‌ها، تخصیص منابع ثابت، ماشین مجازی به عنوان جعبه سیاه، سادگی اجرای نسخه‌ها برای تیم زیرساخت	استفاده بد از منابع زیرساخت، هزینه بالای نصب و نگهداری سیستم‌عامل، زمان‌بر بودن نصب نسخه جدید، راه‌اندازی مجدد کند و زمانگیر، تحمیل کارهای خارج از وظیفه به تیم توسعه
نصب و راه‌اندازی به کمک کانتینرها	هر سرویس با داکرفایل ساخته شده و به سرعت به یک Image تبدیل و اجرا می‌شود.	تمام مزایای ماشین مجازی با سرعت و سادگی بیشتر، عدم نیاز به هزینه نصب سیستم‌عامل و آنتی‌ویروس، مستندسازی مراحل نصب و اجرای سرویس در داکرفایل، اجرای آسان توسط افراد با دانش داکر	نیاز به دانش و تجربه کافی توسعه‌دهندگان و مهندسین زیرساخت در استفاده از کانتینرها، تعداد افراد متخصص محدود

میکروسرویس را انتخاب کنم یا نه؟

سؤال «آیا میکروسرویس را انتخاب کنم یا نه؟» به‌طور مستقیم به نیاز و ظرفیت تیم توسعه بستگی دارد. این معماری که از سال ۲۰۱۰ با تکامل سیستم‌های توزیع شده شکل گرفت، مزایای قابل‌توجهی مانند

مدولار بودن، مقیاس‌پذیری، توسعه مستقل و امکان ادغام تدریجی سرویس‌ها ارائه می‌دهد، اما همزمان چالش‌هایی مانند پیچیدگی ارتباطات بین سرویس‌ها، نیاز به منابع و زیرساخت‌های اضافی، مدیریت داده توزیع‌شده و آزمایش‌های پیچیده را نیز به همراه دارد. بنابراین انتخاب میکروسرویس نیازمند سنجش دقیق مزایا و محدودیت‌ها، توانایی تیم در مدیریت سیستم‌های توزیع‌شده و برنامه‌ریزی مناسب برای برش، مقیاس و طراحی معماری است.



جمع بندی میکروسرویس چیست؟

در نهایت، معماری میکروسرویس نه یک معجزه است و نه یک مد زودگذر؛ یک انتخاب استراتژیک است که اگر بدون شناخت ظرفیت تیم، بلوغ فنی سازمان و پیچیدگی واقعی مسئله انجام شود، بیشتر دردسر می‌سازد تا ارزش. میکروسرویس زمانی معنا پیدا می‌کند که سیستم شما واقعاً به مقیاس‌پذیری مستقل، توسعه موازی، انعطاف فناوری و استقرار سریع نیاز داشته باشد؛ در غیر این صورت یک Monolith تمیز و خوش‌طراحی می‌تواند سال‌ها بدون بحران کار کند. انتخاب درست، نتیجه تحلیل دقیق است نه هیجان تکنولوژی. در نیک آموز تلاش کردیم تصویر کامل، واقع‌بینانه و بدون اغراق از این معماری ارائه دهیم تا تصمیم شما بر پایه منطق و نیاز واقعی باشد، نه صرفاً جذابیت نام «میکروسرویس.»

میکروسرویس را «درست» پیاده‌سازی کن، نه پرهزینه
بیشتر شکست‌های معماری میکروسرویس به دلیل تصمیم‌های اشتباه طراحی است، نه خود تکنولوژی. با شناخت چالش‌ها و راه‌حل‌های واقعی، سیستم‌هایی بساز که مقیاس‌پذیر، پایدار و قابل توسعه باشند.
۵ راز ساخت سیستم قدرتمند با پیاده سازی معماری میکروسرویس : چالش ها و راه حل ها

سوالات متداول میکروسرویس چیست؟

۱. میکروسرویس چیست و چه تفاوتی با معماری Monolithic دارد؟

میکروسرویس یک روش معماری است که نرم‌افزار را به سرویس‌های کوچک، مستقل و قابل توسعه تقسیم می‌کند، در حالی که Monolithic تمام بخش‌ها را در یک برنامه یکپارچه اجرا می‌کند.

۲. مزایای استفاده از میکروسرویس‌ها چیست؟

مزایا شامل توسعه مستقل سرویس‌ها، مقیاس‌پذیری بهتر، مدیریت ساده‌تر منابع، توسعه موازی توسط تیم‌های مستقل، و کاهش تأثیر تغییرات یک سرویس روی کل سیستم است.

۳. چالش‌ها و محدودیت‌های میکروسرویس‌ها کدامند؟

چالش‌ها شامل پیچیدگی ارتباط بین سرویس‌ها، مدیریت داده‌های توزیع‌شده، تست و عیب‌یابی دشوار، نیاز به تیم‌های متخصص و افزایش هزینه‌های زیرساخت است.

٤. چه زمانی معماری میکروسرویس مناسب است؟

زمانی که سیستم پیچیده است، نیاز به توسعه موازی تیم‌ها وجود دارد، منابع باید به صورت مستقل مدیریت شوند، یا تیم‌ها در مکان‌های مختلف جغرافیایی کار می‌کنند، میکروسرویس مناسب است.

٥. API Gateway چیست و چرا استفاده می‌شود؟

API Gatewayیک نقطه ورود واحد برای Clientها است که پیچیدگی تعامل بین میکروسرویس‌ها را مخفی می‌کند، مسیریابی و ترکیب نتایج را انجام می‌دهد و بهره‌وری و تجربه کاربری را افزایش می‌دهد.

٦. ارتباط بین سرویس‌ها در میکروسرویس چگونه انجام می‌شود؟

سرویس‌ها از طریق Inter-Process Communication (IPC) با هم ارتباط دارند، همزمان (Sync) یا ناهمزمان (Async)، با روش‌هایی مانند: Request/Response، Notification، Pub/Sub و Request/Async Response.

۷. Service Discovery چه نقشی در میکروسرویس‌ها دارد؟

Service Discovery مسئول یافتن و مدیریت آدرس سرویس‌ها در محیط توزیع‌شده است و می‌تواند به روش client-side یا server-side پیاده‌سازی شود تا سرویس‌ها همیشه در دسترس باشند.

۸. چگونه داده‌ها بین میکروسرویس‌ها مدیریت می‌شوند؟

هر سرویس دیتابیس مستقل خود را دارد و تعامل با داده‌های دیگر سرویس‌ها از طریق API یا Event انجام می‌شود. روش‌هایی مانند eventual consistency، Event Sourcing و CQRS برای هماهنگی داده‌ها استفاده می‌شوند.

۹. روش‌های انتشار میکروسرویس‌ها کدامند و مزایا و معایب آنها چیست؟

روش‌ها شامل نصب چند سرویس روی یک میزبان، نصب یک سرویس روی هر ماشین مجازی، و استفاده از کانتینرها است. کانتینرها معمولاً سریع‌تر، ساده‌تر و با مستندسازی بهتر هستند، اما نیازمند دانش تخصصی تیم توسعه و زیرساخت است.

۱۰. چه فناوری‌ها و ابزارهایی برای پیاده‌سازی میکروسرویس‌ها استفاده می‌شوند؟

زبان‌ها: Java, Python, Node.js, Go, .NET Core
کانتینر: Docker
مدیریت کانتینر: Kubernetes
API Gateway: AWS API Gateway, Kong
Service Mesh: Istio, Linkerd
دیتابیس‌ها: PostgreSQL, MongoDB, Cassandra
مانیتورینگ: Prometheus, Grafana, ELK Stack
CI/CD: Jenkins, GitLab CI, Travis CI

بیشتر بخوانید:

- طراحی معماری میکروسرویس با الگوی SAGA
- تفاوت DDD ، میکروسرویس(Microservice) ، الگوهای طراحی (Design pattern) و معماری تمیز(Clean Architecture)