تفاوت های ساختاری

Monolithic  
یک برنامه به‌صورت یک واحد واحدِ deployable است UI، منطق تجاری و لایهٔ داده معمولاً در یک کدبیس و یک فرآیند اجرا قرار دارند. توسعه و deploy معمولاً به‌صورت یک بستهٔ کامل انجام می‌شود که ساده‌تر است اما با رشد برنامه بزرگ و درهم‌تنیده می‌شود.

Microservices

برنامه به مجموعه‌ای از سرویس‌های مستقل تقسیم می‌شود؛ هر سرویس مسئول یک قابلیت تجاری است، می‌تواند جداگانه توسعه، تست، deploy و مقیاس‌پذیر شود و معمولاً با پروتکل‌های سبک (HTTP/REST, gRPC, messaging) با هم صحبت می‌کنند. این سبک، استقلال تیمی و مقیاس‌پذیری جزئی را تقویت می‌کند اما پیچیدگی توزیع‌شده (networking, eventual consistency, observability) را وارد می‌کند.

* تک‌بسته vs مجموعهٔ سرویس‌های مستقل.
* DB معمولا مشترک در مونولیت (یا مدیریت مرکزی)؛ در میکروسرویس‌ها معمول است که هر سرویس DB/پرسپکتیو دادهٔ خود را داشته باشد (bounded context).
* استقرار و چرخهٔ انتشار: یک بسته‌ی بزرگ در مونولیت، چندین واحد مستقل در میکروسرویس.

روش‌های احراز هویت و مدیریت امنیت در هر معماری

Monolith

* معمولاً احراز هویت و مجوز (authn/authz) در یک نقطهٔ مرکزی (مثلاً لایهٔ وب یا یک ماژول) انجام می‌شود: session-based auth، cookie، یا OAuth2/OpenID Connect برای SSO. کنترل دسترسی و سیاست‌ها هم‌درون برنامه قابل پیاده‌سازی و ساده‌تر است (یک کدبیس، یک نقطهٔ ورود).
* مزیت: پیاده‌سازی ساده و ردیابی جریان درخواست آسان است. نقطه‌ضعف: اگر نقطهٔ مرکزی آسیب‌پذیر شود، کل برنامه در خطر است.

Microservices

* **Edge / Gateway-based auth:** API Gateway یا BFF (Backend-for-Frontend) احراز هویت را برای تمام درخواست‌ها انجام می‌دهد (مثلاً اعتبارسنجی JWT/OIDC) و درخواست‌های معتبر را به سرویس‌های داخلی هدایت می‌کند. ساده‌سازی کلاینت و متمرکزسازی اما ریسک “گذشتن از گیت‌وی” را دارد — باید دفاع در عمق برقرار شود (مثلاً mutual TLS بین سرویس‌ها).
* **Token propagation / Service-level auth:** توکن (JWT یا reference token) از لبه منتقل و توسط هر سرویس بررسی یا از یک IAM/Policy Decision Point پرسیده می‌شود. مزیت: کنترل‌های دقیق‌تر؛ معایب: پیچیدگی توکن‌ها، مسائل بازخوانی/انقضا، و نیاز به مدیریت مرکزی برای ریوک و حقوق.
* **Centralized IAM (OAuth2/OIDC) + mTLS + mutual auth برای سرویس-به-سرویس:** در محیط‌های حساس توصیه می‌شود سرویس‌ها یکدیگر را با گواهی و کنترل سطح پایین شناسایی کنند و از سرویس‌های IAM برای تصمیم‌گیری دربارهٔ دسترسی استفاده شود.

نکات عملی امنیتی برای میکروسرویس‌ها

* لاگینگ و ترَیسینگ توزیع‌شده (distributed tracing) برای بررسی حملات و ریشه‌یابی ضروری است.
* مدیریت توکن—ریووک، زمان انقضا، و استفاده از reference tokens به‌جای توکن‌های خیلی بزرگ (در صورت نیاز) را در نظر بگیرید.
* سیاست «نپذیرفتن دسترسی مستقیم به سرویس‌ها» (only via gateway) و استفاده از شبکهٔ داخلی امن.

مقایسهٔ قابلیت توسعه‌پذیری (Scalability)

Monolith

معمولاً نیاز به **Vertical scaling** (بزرگ‌تر کردن سرور) یا کپی‌برداری کامل اپلیکیشن (replicate whole app) برای افزایش ظرفیت. توسعهٔ مقیاس‌پذیری برای بخش‌های خاص هزینه‌بر و ناکارآمد است چون کل اپ را باید کپی کنید. مناسب برای بارهای نسبتاً یکنواخت و سیستم‌های کوچک تا متوسط.

Microservices

**Horizontal & Fine-grained scaling: هر سرویس را می‌توان جداگانه مقیاس کرد (مثلاً سرویس پرداخت یا جستجو را بیشتر کنید بدون افزایش نمونه‌های سرویس گزارش‌دهی). این به‌خصوص وقتی مفید است که بار بین قابلیت‌ها ناهمگون باشد. اما مدیریت و هماهنگی بین سرویس‌ها (load balancing، service discovery، شبکه) پیچیده‌تر می‌شود.**

**خلاصه: اگر نیاز به مقیاس‌پذیری دقیق و ناهمگن (hotspots) دارید و تیم عملیات/SRE و زیرساخت را دارید، میکروسرویس برتری دارد. برای MVPها و سیستم‌های با بار همگن، مونولیت ممکن است کاراتر و ارزان‌تر باشد.**

**هزینه‌های پیاده‌سازی، نگهداری و توسعه**

**Monolith**

**پیاده‌سازی اولیه:** گران‌تر و کندتر — نیاز به طراحی دامنه‌ها، infra برای سرویس‌دیسکاوری، شبکهٔ امن، CI/CD پیچیده، ابزار مانیتورینگ و ترَیسینگ، تیم SRE/DevOps.

**نگهداری در طول زمان:** با رشد برنامه ممکن است هزینه‌های نگهداری افزایش یابد (کندی توسعه، زمان‌های تست و deploy طولانی، وابستگی‌های درهم). بازتوسعه یا مهاجرت ممکن است پرهزینه شود.

Microservices

**پیاده‌سازی اولیه:** گران‌تر و کندتر — نیاز به طراحی دامنه‌ها، infra برای سرویس‌دیسکاوری، شبکهٔ امن، CI/CD پیچیده، ابزار مانیتورینگ و ترَیسینگ، تیم SRE/DevOps.

**هزینهٔ عملیاتی: هزینهٔ زیرساخت (بیشتر instances، بیشتر شبکه، بیشتر ذخیره‌سازی)، هزینهٔ پیچیدگی (debugging توزیع‌شده)، و هزینهٔ مهارتی (تیم‌های تخصصی). در مقیاس بزرگ اما می‌تواند باعث بهینه‌سازی منابع و سرعت توسعهٔ مستقل تیم‌ها شود که در بلندمدت منافع اقتصادی دارد.**

**نکتهٔ تجاری:** به‌طور کلی مونولیت برای پایین آوردن هزینهٔ شروع مناسب است؛ میکروسرویس برای سازمان‌هایی که مقیاس بالایی دارند یا نیاز به چرخه‌های مستقل توسعه برای تیم‌های متعدد دارند، اقتصادی‌تر است — اما با شرط داشتن سرمایه‌گذاری اولیه در infra و عملیات.

سایر نکات مهم

* **پیچیدگی عملیاتی:** میکروسرویس‌ها پیچیدگی شبکه/توزیع‌شده، مسئلهٔ نسخه‌ساز‌ی API بین سرویس‌ها، و مشکلات eventual consistency را وارد می‌کنند. مونولیت ساده‌تر اما مستعد «کد بی‌قواره» (spaghetti) در طول زمان است.
* **تست:** در مونولیت، تست end-to-end ساده‌تر است؛ در میکروسرویس‌ها باید ترکیبی از unit, contract, integration, and consumer-driven tests و همچنین تست‌های سیستمی توزیع‌شده را اجرا کنید.
* **استقرار (Deployment):** میکروسرویس‌ها نیاز به CI/CD اتوماتیک برای هر سرویس، مدیریت نسخه‌ها و rollout‌های کانتینر (یا serverless) دارند؛ مونولیت تنها یک pipeline ساده‌تر نیاز دارد.
* **خطایابی و Observability:** در میکروسرویس‌ها لازم است tracing (مثلاً OpenTelemetry)، مرکزی‌سازی لاگ، متریک‌ها و alerting قوی را پیاده کنید تا ریشه‌یابی سریع ممکن شود؛ در مونولیت ردیابی ساده‌تر اما وقتی خطا در لایهٔ داده/بیزنس است، بازتولید مشکل ممکن است دشوار باشد.
* **توسعهٔ تیمی و سازمانی:** میکروسرویس‌ها با تیم‌های کوچکِ دارای مالکیت سرویس (bounded context) همخوانی خوبی دارند؛ مونولیت می‌تواند برای تیم‌های کوچک یا وقتی که هماهنگی تیمی کمتر موردنظر است، مناسب باشد.
* **وابستگی‌ها و داده‌ها:** در میکروسرویس‌ها مسائل ACID و تراکنش چندسرویسی (saga pattern، eventual consistency) مطرح می‌شود؛ مونولیت راحت‌تر تراکنش‌های ACID را حفظ می‌کند چون معمولاً یک DB مشترک است.

جمع‌بندی کلی و توصیه‌ها

**Monolithic = ساده‌تر برای شروع، کم‌هزینه‌تر در آغاز، توسعه/تست/دیباگ راحت‌تر؛ اما در مقیاس بزرگ می‌تواند به بن‌بست تبدیل شود.**

**Microservices = انعطاف‌پذیر و قابل مقیاس برای سازمان‌ها و سیستم‌های بزرگ با نیاز به توسعهٔ موازی؛ اما نیاز به سرمایه‌گذاری قابل توجه در infra، امنیت، و عملیات دارد.**