A diagram of a webpage

AI-generated content may be incorrect.Monolith Architecture

1. **Defination**: **Kiến trúc nguyên khối** là một mô hình truyền thống của một chương trình phần mềm, được xây dựng như một đơn vị **thống nhất, khép kín và độc lập** với các ứng dụng khác**.**
2. **Props (Ưu điểm) and Cons (Nhược điểm):** 
   1. **Props (Ưu điểm)**:
      1. Kiến trúc nguyên khối rất đơn giản và dễ hiểu vì nó liên quan đến việc xây dựng toàn bộ ứng dụng trong một cơ sở mã duy nhất với một bộ ngôn ngữ và khung. ***Ví dụ*:** Tất cả modules (web, service, repository) đều nằm trong cùng một project Spring Boot với file pom.xml duy nhất. Chạy lệnh mvn spring-boot:run là khởi động được toàn bộ ứng dụng.
      2. Kiến trúc nguyên khối thường hoạt động tốt hơn kiến trúc phân tán vì nó không yêu cầu giao tiếp mạng giữa các thành phần khác nhau**. *Ví dụ***: Controller gọi trực tiếp đến Service (@Autowired) trong cùng JVM, không phải gọi REST hay gRPC, nên overhead IO gần như bằng 0.
      3. Việc kiểm tra các ứng dụng nguyên khối dễ dàng hơn vì chỉ có một cơ sở mã để kiểm tra. Bằng cách này, bạn có thể dễ dàng đảm bảo rằng tất cả các thành phần hoạt động chính xác với nhau.***Ví dụ*:** Dùng @SpringBootTest để boot cả application context, có thể kiểm tra flow từ HTTP request đến DB mà không cần mock network hay phải deploy microservice riêng.
      4. Kiến trúc nguyên khối thường ít tốn kém hơn để phát triển và bảo trì vì nó không yêu cầu nhiều công nghệ và công cụ khác nhau, làm giảm độ phức tạp của ứng dụng. **Ví dụ:** Chỉ cần một pipeline CI/CD build–test–deploy cho 1 artifact (JAR), không cần cấu hình nhiều repo, nhiều cluster cho microservices.
      5. Với kiến trúc nguyên khối, việc đảm bảo các biện pháp bảo mật được áp dụng nhất quán trên toàn bộ ứng dụng sẽ dễ tiếp cận hơn***Ví dụ*:** Cấu hình Spring Security trong một class duy nhất (WebSecurityConfigurerAdapter), áp dụng chung cho toàn bộ URL endpoint mà không sợ thiếu service nào “bỏ sót” authentication/authorization.
   2. **Cons** (**Nhược điểm):**
      1. Việc cập nhật bất kỳ phần nào của cơ sở mã sẽ tác động đến kiến trúc nguyên khối, yêu cầu biên dịch lại ứng dụng hoàn chỉnh.***Ví dụ***: Thay đổi logic trong module thanh toán (payment) thì phải rebuild & redeploy cả JAR ~50 MB, dù module khác không đổi.
      2. Bất kỳ lỗi hoặc sự cố máy chủ nào đều ảnh hưởng đến toàn bộ ứng dụng, ảnh hưởng đến độ tin cậy tổng thể của ứng dụng.***Ví dụ*:** Một NullPointerException trong OrderController chưa catch được sẽ crash JVM, kéo theo downtime toàn service, không chỉ mỗi phần order.
      3. Việc sử dụng lại mã bị hạn chế, thường chỉ được hỗ trợ với các thư viện dùng chung (dẫn đến các vấn đề về trùng lặp mã, quá phụ thuộc vào nhau**). *Ví dụ*:** Nếu muốn dùng module “user” trong một project khác, phải copy-paste code hoặc extract thành thư viện riêng, dễ dẫn đến duplicate code và khó đồng bộ.
      4. Những thay đổi đối với bất kỳ phần nào của mã ứng dụng đều trở nên khó khan tốn kém do phụ thuộc**. *Ví dụ*:** Service A gọi Service B trực tiếp qua class, nên sửa interface của B sẽ làm A compile lỗi, phải sửa A cùng lúc, gây tốn thời gian.
      5. Cơ sở mã có thể trở nên cồng kềnh theo thời gian, gây khó khăn cho việc duy trì và bảo trì đồng thời gây khó khắn cho những người mới khi tiếp cận. **Ví dụ:** Sau 2 năm, project có hàng trăm package (com.myapp.user, com.myapp.order, com.myapp.product, …) và >1000 class, new developer mất nhiều tuần để hiểu hết flow.
      6. Kiến trúc nguyên khối **không** **hỗ trợ** **mở rộng theo chiều ngang (chỉ có thể mở rộng theo chiều dọc),** do đó mọi thứ mọi thứ đều phải mở rộng trong khi chỉ một phần của ứng dụng có thể đang phải chịu tải lớn. **Ví dụ:** Khi load cao, chỉ có thể tăng CPU/RAM cho server chạy JAR, không thể scale từng module “order” hay “payment” riêng lẻ mà phải scale toàn bộ.
      7. Trong kiến trúc nguyên khối, chỉ dùng một ngăn xếp công nghệ (**technical stack)**duy nhất cho mọi thứ**. Ví dụ:** Nếu muốn thử Node.js cho real-time chat, phải tạo project tách biệt; trong cùng monolith không thể mix Kotlin, Node, Go mà chỉ toàn Java + Spring.
3. **Khi nào nên chọn Monolith?**
   1. Ứng dụng nhỏ hoặc MVP, tính năng đơn giản, nhóm phát triển gọn (1–3 người).
   2. Yếu tố tốc độ phát triển ban đầu quan trọng hơn khả năng mở rộng dài hạn.
   3. Hệ thống chưa có yêu cầu chịu tải cao hoặc chưa cần linh hoạt chuyển đổi công nghệ.

**Tóm lại,** monolith rất phù hợp giai đoạn khởi đầu, đơn giản và nhanh chóng, nhưng khi hệ thống lớn lên, nhu cầu scale và phát triển song song tăng, microservices sẽ giúp giải quyết tốt hơn những hạn chế về quy mô và độc lập giữa các thành phần.

A diagram of a shopping service

AI-generated content may be incorrect.Microservices

1. **Defination**:

**Kiến trúc microservices** là một loại kiến trúc ứng dụng trong đó ứng dụng **được phát triển dưới dạng tập hợp các service.**

Nó cung cấp khuôn khổ để phát triển, triển khai và duy trì các sơ đồ kiến trúc kết nối nhưng vẫn **đảm bảo các thành phần service con hoạt động một cách độc lập.**

1. **Nguyên tắc:**
   1. **Độc lập dịch vụ:** Mỗi service có codebase, database, config riêng.

Ví dụ:

Service User dùng PostgreSQL, Service Order dùng MongoDB.

Khi cần thay đổi schema bảng user, chỉ deploy lại User-service, không tác động Order-service.

* 1. **Phân cấp (Bounded Context & Team Ownership):** Chia theo domain, mỗi nhóm tự chịu trách nhiệm end-to-end (dev, test, deploy).**Ví dụ:** Nhóm “Payment” quản lý Payment-service (thiết kế API, CI/CD pipeline, logging). Nhóm “Catalog” quản lý Catalog-service riêng, có sprint, backlog và release cycle độc lập.
  2. **Tính linh hoạt (Polyglot & Autonomous Deployments):** Cho phép dùng tech stack khác nhau và deploy riêng lẻ. **Ví dụ:** Service “Notification” viết bằng Node.js để xử lý WebSocket, deploy trên một Pod Kubernetes khác.

Service “Reporting” viết bằng Python, scale lên khi cần chạy batch job mà không ảnh hưởng dịch vụ khác.

* 1. **Khả năng phục hồi (Resilience & Fault Isolation):**  Service phải chịu lỗi cục bộ, không làm sập cả hệ thống. **Ví dụ:** Service “Recommendation” dùng circuit breaker (Resilience4j). Nếu DB chậm, circuit breaker mở cửa và trả fallback, không kéo gãy flow đặt đơn hàng. Service “Search” timeout sau 500 ms, trả lỗi nhẹ cho client, không block toàn bộ UI.
  2. **Khả năng mở rộng (Scalability):** Scale từng service độc lập theo nhu cầu. **Ví dụ:** Khi traffic đặt hàng tăng, chỉ scale thêm replicas cho Order-service (kubectl scale deployment order-service --replicas=5), User-service vẫn giữ 2 replicas.
  3. **Tích hợp & triển khai liên tục (CI/CD):** Mỗi repo/service có pipeline build–test–deploy riêng, deploy tự động khi code được merge. **Ví dụ:** Dùng GitHub Actions: push code vào main branch của Cart-service tự động chạy unit test, build Docker image và deploy lên staging Kubernetes bằng Helm chart.
  4. **API-first :** Thiết kế API trước, dùng OpenAPI/Swagger để document, mock server. **Ví dụ:** Team định nghĩa spec order-service.yaml (OpenAPI 3.0), generate client SDK Java, Node. Các service khác gọi Order-service qua REST theo spec này, có mock server để front-end dev song song.

1. **Props and Cons (Ưu điểm và nhược điểm):**
   1. **Props:** 
      1. **Do có thể mở rộng các service độc lập với nhau nên hệ thống có tính chịu tải cao, xử lý được lượng truy cập đồng thời cao cũng như những thời điểm tăng đột biến.**  
         Ví dụ: Khi traffic đặt hàng tăng đột biến trong dịp khuyến mãi, bạn chỉ cần scale thêm replicas cho order-service trên Kubernetes:
         1. kubectl scale deployment order-service --replicas=10

trong khi các service khác (user, payment…) vẫn giữ nguyên số replicas.

* + 1. **Bất kỳ lỗi hoặc sự cố máy chủ nào đều ảnh hưởng đến toàn bộ ứng dụng, ảnh hưởng đến độ tin cậy tổng thể của ứng dụng.**  
       Ví dụ: Nếu payment-service gặp lỗi NullPointer và crash, chỉ service này bị ảnh hưởng, còn order-service vẫn có thể chấp nhận đơn và trả fallback response — không làm sập toàn bộ hệ thống.
    2. **Việc sử dụng lại mã bị hạn chế, thường chỉ được hỗ trợ với các thư viện dùng chung (dẫn đến các vấn đề về trùng lặp mã).**  
       Ví dụ: Nếu cả user-service và order-service cùng cần validate email, bạn có thể extract thành một “common-lib” trên Maven, nhưng hai team vẫn phải import và cập nhật đồng thời mỗi khi thay đổi, dễ dẫn đến mismatch phiên bản.
    3. **Những thay đổi đối với bất kỳ phần nào của mã ứng dụng đều trở nên khó khăn tốn kém do phụ thuộc.**  
       Ví dụ: Thay đổi API trả về của inventory-service (ví dụ đổi field stockAvailable) thì phải điều chỉnh cả order-service và product-service, rebuild và deploy đồng loạt.
    4. **Cơ sở mã có thể trở nên cồng kềnh theo thời gian, gây khó khăn cho việc duy trì và bảo trì đồng thời gây khó khăn cho những người mới khi tiếp cận.**  
       Ví dụ: Sau 2 năm phát triển, mỗi service có thêm nhiều module (reporting, analytics, notification…), new developer cần xem hàng trăm class và config Kubernetes/Helm chart để hiểu end-to-end flow.
    5. **Kiến trúc nguyên khối không hỗ trợ mở rộng theo chiều ngang (chỉ có thể mở rộng theo chiều dọc), do đó mọi thứ đều phải mở rộng trong khi chỉ một phần của ứng dụng có thể đang phải chịu tải lớn.**  
       Ví dụ: Trong monolith, muốn chịu tải cao cho tính năng chat realtime, bạn phải scale toàn bộ JAR Spring Boot trên nhiều VM, thay vì chỉ scale riêng chat-service như microservice.
    6. **Trong kiến trúc nguyên khối, chỉ dùng một ngăn xếp công nghệ (technical stack) duy nhất cho mọi thứ.**  
       Ví dụ: Monolith Java/Spring Boot khó kết hợp thêm service Python cho AI hoặc Node.js cho WebSocket; nếu muốn, lại phải tách repo, tách pipeline, dễ tạo ra hệ hỗn loạn.
  1. **Cons:**
     1. **Độ phức tạp cao hơn**. **Ví dụ**: Team **User** xây dựng user-service bằng Spring Boot, team **Payment** xây dựng payment-service bằng Node.js. Cần cấu hình Nginx/Traefik làm API Gateway để cân bằng tải, cấu hình SSL, routing cho nhiều domain/service. Cấu hình monitoring với Prometheus + Grafana để theo dõi metrics từ mỗi service, cấu hình alert rules cho từng endpoint.
     2. **Tái sử dụng mã có giới hạn**. Ví dụ: Cả order-service và inventory-service đều cần logic validate email. Phải extract thành Maven artifact common-validation rồi publish lên Nexus/Artifactory. Mỗi khi thay đổi, cả hai team phải update version dependency, build & deploy lại service để nhận thay đổi.
     3. **Tăng thời gian phát triển**. Ví dụ: Khi order-service cần gọi API của inventory-service, developer phải: Chờ team Inventory viết spec OpenAPI/Swagger. Cấu hình generator (OpenAPI Generator) để tạo client stub. Viết mock server hoặc stub tests cho local development.
     4. Tất cả bước này làm chậm tiến độ feature của cả hai bên.
     5. **Sự phụ thuộc vào DevOps** .**Ví dụ:** Mỗi khi tạo thêm notification-service, DevOps phải: Viết GitLab CI pipeline (build Docker image, chạy test, push registry). Tạo Helm chart, values.yaml riêng. Cấu hình namespace, RBAC, ingress mới trên Kubernetes. Khi số service tăng lên, workload cho đội DevOps (CI/CD, infra, config) cũng tăng theo.
     6. **Phức tạp trong việc kiểm tra và gỡ lỗi toàn bộ ứng dụng**  
        Ví dụ: Muốn debug flow “khách đặt hàng thành công”, ta phải: Bật distributed tracing (OpenTelemetry + Jaeger) cho API Gateway, Authentication, Order, Inventory, Payment. Chạy end-to-end test trên môi trường staging. Dùng Kibana để search log traceId xuyên suốt 5 service. Cả quy trình rất tốn công, dễ bỏ sót lỗi ở service phụ.
  2. **Khi nào nên chuyển sang microservices?**
     1. **Codebase & release chậm:** Monolith quá lớn, build/test/deploy mất nhiều phút.
     2. **Scale không đồng đều:** Chỉ một vài tính năng chịu tải cao nhưng phải scale toàn bộ monolith.
     3. **Nhiều team & CI/CD độc lập:** Các nhóm cần deploy riêng mà không chờ nhau.
     4. **Polyglot & công nghệ khác nhau:** Muốn mix Java, Node, Python… trong cùng hệ thống.
     5. **Độ sẵn sàng & fault isolation:** Lỗi của service này không làm sập toàn bộ hệ thống.