1. Monolithic Architecture – Kiến trúc nguyên khối

📌 Bài toán:

Website quản lý khóa học trực tuyến

* Chức năng chính: Đăng ký khóa học, quản lý học viên, theo dõi tiến độ, bài kiểm tra.
* Ban đầu có ít người dùng, không yêu cầu mở rộng nhanh.

🏗 Cách triển khai:

* Sử dụng một codebase duy nhất chứa toàn bộ frontend, backend, database.
* Các thành phần như API, xử lý logic, và giao diện đều nằm chung trong một ứng dụng.

🔧 Công nghệ phù hợp:

* Backend: Spring Boot (Java), Django (Python), Laravel (PHP)
* Frontend: React, Angular, Vue.js (tích hợp trực tiếp vào dự án backend)
* Database: MySQL, PostgreSQL

✅ Ưu điểm:

* Đơn giản, dễ phát triển và triển khai.
* Hiệu suất cao vì không cần giao tiếp giữa nhiều dịch vụ.
* Dễ debug vì tất cả logic nằm trong cùng một ứng dụng.

❌ Nhược điểm:

* Khi ứng dụng lớn, codebase trở nên cồng kềnh, khó bảo trì.
* Khó mở rộng, khi số lượng người dùng tăng đột biến sẽ dễ bị quá tải.
* Nếu có lỗi trong một phần, toàn bộ hệ thống có thể bị ảnh hưởng.

📌 Tình huống thực tế phù hợp:

✅ Doanh nghiệp nhỏ với ứng dụng nội bộ  
✅ Startup đang thử nghiệm sản phẩm

2. Layered (n-tier) Architecture – Kiến trúc phân lớp

📌 Bài toán:

Hệ thống quản lý bệnh viện

* Quản lý bệnh nhân, lịch khám, hồ sơ bệnh án.
* Đòi hỏi tính tổ chức cao, phân chia rõ ràng từng tầng xử lý.

🏗 Cách triển khai:

* Tách ứng dụng thành 3-4 lớp chính:
  1. Presentation Layer: Giao diện người dùng (React, Angular).
  2. Business Layer: Xử lý logic nghiệp vụ (Spring Boot, Node.js).
  3. Data Access Layer: Quản lý database (MySQL, MongoDB).
  4. (Tùy chọn) Integration Layer: Kết nối với hệ thống bên ngoài.

🔧 Công nghệ phù hợp:

* Backend: Spring Boot, .NET Core, Express.js
* Frontend: React, Angular
* Database: PostgreSQL, Oracle

✅ Ưu điểm:

* Tách biệt rõ ràng giúp dễ mở rộng và bảo trì.
* Từng lớp có thể thay đổi mà không ảnh hưởng đến toàn bộ hệ thống.

❌ Nhược điểm:

* Hiệu suất có thể bị ảnh hưởng vì mỗi request phải đi qua nhiều lớp.
* Nếu không thiết kế tốt, có thể xuất hiện nhiều lớp thừa gây cồng kềnh.

📌 Tình huống thực tế phù hợp:

✅ Ứng dụng doanh nghiệp lớn  
✅ Hệ thống có nhiều logic nghiệp vụ phức tạp

3. Client-Server Architecture – Kiến trúc máy khách - máy chủ

📌 Bài toán:

Ứng dụng chat nội bộ

* Nhân viên có thể nhắn tin, gửi file, gọi video trong công ty.
* Server xử lý tin nhắn, file, quản lý danh bạ.

🏗 Cách triển khai:

* Máy khách (Client) gửi request đến máy chủ (Server) để lấy dữ liệu.
* Máy chủ xử lý và gửi phản hồi về cho client.

🔧 Công nghệ phù hợp:

* Client: React Native, Flutter
* Server: Node.js (WebSocket), Firebase, Java Spring Boot
* Database: MongoDB, Redis (cache tin nhắn)

📌 Ưu điểm:

✔ Dữ liệu tập trung, dễ bảo mật: Tất cả dữ liệu được lưu trữ và quản lý trên máy chủ, giúp dễ kiểm soát quyền truy cập và bảo mật thông tin.  
✔ Dễ dàng triển khai trên nhiều nền tảng: Client có thể là ứng dụng web, mobile, hoặc desktop, chỉ cần giao tiếp với server thông qua API.  
✔ Quản lý dữ liệu hiệu quả: Do máy chủ là trung tâm xử lý, việc quản lý tin nhắn, file và danh bạ trở nên dễ dàng hơn.  
✔ Hỗ trợ mở rộng: Có thể nâng cấp máy chủ (scale-up) hoặc bổ sung nhiều máy chủ (scale-out) khi lượng người dùng tăng lên.  
✔ Dễ duy trì và cập nhật: Mọi thay đổi (ví dụ: cập nhật giao diện, thêm tính năng) chỉ cần cập nhật trên server, thay vì thay đổi trên từng client.

⚠ Nhược điểm:

❌ Phụ thuộc vào máy chủ: Nếu server gặp sự cố hoặc quá tải, toàn bộ hệ thống có thể bị gián đoạn.  
❌ Độ trễ cao khi server quá tải: Khi có quá nhiều yêu cầu từ client, server có thể bị nghẽn, gây chậm trễ trong việc phản hồi.  
❌ Chi phí duy trì server cao: Cần có hệ thống mạnh mẽ để xử lý lượng lớn yêu cầu từ client, đồng thời cần cơ chế cân bằng tải và cache để tối ưu hiệu suất.  
❌ Rủi ro bảo mật: Nếu server bị tấn công (DDoS, SQL Injection, XSS), toàn bộ hệ thống có thể bị ảnh hưởng nghiêm trọng.  
❌ Khó mở rộng theo chiều ngang: Việc scale-out hệ thống Client-Server truyền thống thường phức tạp hơn so với các kiến trúc phân tán như Microservices hoặc Serverless.

📌 Khi nào nên chọn Client-Server Architecture?

* Khi cần một hệ thống tập trung để quản lý dữ liệu (ví dụ: ứng dụng chat nội bộ, hệ thống CRM, ERP).
* Khi ứng dụng có số lượng người dùng vừa phải và không cần mở rộng quy mô quá lớn ngay từ đầu.
* Khi cần một kiến trúc dễ hiểu, dễ triển khai, phù hợp với các ứng dụng web và mobile thông thường.

📌 Tình huống thực tế phù hợp:

✅ Ứng dụng chat nội bộ  
✅ Ứng dụng SaaS (phần mềm như dịch vụ)

4. Pipeline Architecture – Kiến trúc đường ống

🔹 Bài toán:  
Một hệ thống xử lý log hệ thống theo thời gian thực với các chức năng:

* Thu thập log từ nhiều nguồn khác nhau
* Phân tích dữ liệu log để phát hiện bất thường
* Lưu trữ log để tra cứu sau này

🔹 Công nghệ phù hợp:

* Data Processing: Apache Kafka, Apache Flink, Apache Spark
* Storage: Hadoop HDFS, Amazon S3, Google Cloud Storage
* Database: Cassandra, Elasticsearch
* Machine Learning (nếu cần): TensorFlow, PyTorch
* Triển khai: Kubernetes, Docker

📌 Lý do chọn: Hệ thống cần xử lý dữ liệu theo từng bước, có thể chia pipeline thành nhiều giai đoạn.

📌 Ưu điểm:

✔ Xử lý dữ liệu theo luồng: Dữ liệu được xử lý theo từng giai đoạn (ingestion, processing, storage), giúp tổ chức hệ thống rõ ràng.  
✔ Dễ dàng mở rộng: Mỗi bước trong pipeline có thể được mở rộng độc lập tùy theo nhu cầu xử lý dữ liệu.  
✔ Chạy song song và tối ưu hiệu suất: Dữ liệu có thể được xử lý đồng thời ở nhiều bước khác nhau, tăng tốc độ tổng thể.  
✔ Tính linh hoạt cao: Có thể thay đổi hoặc nâng cấp từng thành phần mà không ảnh hưởng đến toàn bộ hệ thống.  
✔ Dễ dàng tích hợp AI/ML: Có thể bổ sung các mô-đun Machine Learning để phân tích log hoặc phát hiện bất thường.

⚠ Nhược điểm:

❌ Độ phức tạp cao: Xây dựng và duy trì pipeline cần có kiến thức chuyên sâu về hệ thống phân tán và xử lý dữ liệu.  
❌ Độ trễ có thể xảy ra: Nếu một bước trong pipeline bị tắc nghẽn, toàn bộ quá trình xử lý có thể bị chậm lại.  
❌ Khó debug: Khi dữ liệu di chuyển qua nhiều giai đoạn, việc xác định lỗi trong pipeline có thể mất nhiều thời gian.  
❌ Yêu cầu tài nguyên lớn: Các công cụ như Kafka, Spark, và Flink cần phần cứng mạnh và bộ nhớ lớn để hoạt động hiệu quả.

📌 Khi nào nên chọn Pipeline Architecture?

* Khi cần xử lý luồng dữ liệu lớn theo thời gian thực (ví dụ: hệ thống log, IoT, phân tích dữ liệu tài chính).
* Khi muốn chia nhỏ quy trình xử lý để dễ quản lý và tối ưu từng bước.
* Khi có nhu cầu tích hợp AI/ML vào hệ thống xử lý dữ liệu.

5. Event-driven Architecture – Kiến trúc hướng sự kiện

📌 Bài toán:

Hệ thống đặt vé máy bay

* Người dùng đặt vé, thanh toán, nhận thông báo cập nhật chuyến bay.

🏗 Cách triển khai:

* Sử dụng cơ chế event bus để truyền thông điệp giữa các service.
* Ví dụ: Khi người dùng đặt vé → Gửi event “Đặt vé thành công” → Các service liên quan (thanh toán, gửi email) nhận event này và xử lý.

🔧 Công nghệ phù hợp:

* Event bus: Kafka, RabbitMQ
* Backend: Node.js, Spring Boot
* Database: PostgreSQL, MongoDB

✅ Ưu điểm:

* Hệ thống phản hồi nhanh, hoạt động theo sự kiện.
* Dễ mở rộng bằng cách thêm nhiều event handler.

❌ Nhược điểm:

* Debug khó vì sự kiện có thể diễn ra không đồng bộ.
* Cần cơ chế quản lý tin nhắn để tránh mất dữ liệu.

📌 Tình huống thực tế phù hợp:

✅ Ứng dụng thương mại điện tử  
✅ Hệ thống tài chính, ngân hàng

6. Space-based Architecture – Kiến trúc dựa trên không gian

🔹 Bài toán:  
Một hệ thống giao dịch chứng khoán tốc độ cao, với yêu cầu:

* Xử lý hàng triệu giao dịch mỗi giây
* Đảm bảo tính toàn vẹn dữ liệu, độ trễ thấp
* Chia tải giữa nhiều server khi số lượng giao dịch tăng cao

🔹 Công nghệ phù hợp:

* Caching: Redis, Hazelcast
* Database: NoSQL (Cassandra, MongoDB)
* Backend: Java (Spring Boot), Golang
* Message Queue: Apache Kafka
* Triển khai: Kubernetes, AWS Auto Scaling

📌 Lý do chọn: Kiến trúc này giúp xử lý khối lượng giao dịch lớn bằng cách phân tán dữ liệu hiệu quả.

📌 Ưu điểm:

✔ Hiệu suất cao: Xử lý được hàng triệu giao dịch mỗi giây nhờ phân tán dữ liệu và xử lý song song.  
✔ Giảm tải cho database: Dữ liệu được lưu trong bộ nhớ phân tán (Redis, Hazelcast), hạn chế truy vấn trực tiếp vào database.  
✔ Độ trễ thấp: Giao dịch diễn ra nhanh chóng vì dữ liệu được lưu gần với nơi xử lý thay vì phải truy vấn từ database trung tâm.  
✔ Tính sẵn sàng cao: Nếu một node bị lỗi, hệ thống vẫn hoạt động nhờ khả năng nhân bản dữ liệu giữa các node.  
✔ Tự động mở rộng (Auto Scaling): Khi nhu cầu tăng cao, hệ thống có thể tự động thêm node mà không ảnh hưởng đến hiệu suất chung.

⚠ Nhược điểm:

❌ Độ phức tạp cao: Việc quản lý dữ liệu phân tán đòi hỏi kiến thức sâu về caching, replication và consistency.  
❌ Khó đảm bảo tính nhất quán dữ liệu: Vì dữ liệu được phân tán giữa nhiều node, cần có cơ chế đồng bộ và giải quyết xung đột dữ liệu.  
❌ Cần phần cứng mạnh: Bộ nhớ RAM lớn và CPU mạnh để xử lý dữ liệu nhanh chóng.  
❌ Khó debug và giám sát: Vì dữ liệu không lưu cố định trong một database duy nhất, việc theo dõi lỗi và log có thể phức tạp.

📌 Khi nào nên chọn Space-based Architecture?

* Khi cần xử lý lượng lớn dữ liệu trong thời gian thực (real-time).
* Khi hệ thống có lưu lượng giao dịch cao và yêu cầu độ trễ cực thấp (ví dụ: chứng khoán, thương mại điện tử quy mô lớn).
* Khi muốn giảm tải cho database và tăng tính sẵn sàng của hệ thống.

7. Service-oriented Architecture (SOA) – Kiến trúc hướng dịch vụ

🔹 Bài toán:  
Một hệ thống quản lý bệnh viện với các dịch vụ:

* Đặt lịch hẹn với bác sĩ
* Quản lý hồ sơ bệnh nhân
* Quản lý kho thuốc

🔹 Công nghệ phù hợp:

* Backend: Java (Spring Boot), .NET Core
* Database: PostgreSQL, MySQL
* API Communication: SOAP, RESTful API
* Authentication: OAuth2, SAML
* Triển khai: Kubernetes, Docker

📌 Lý do chọn: SOA phù hợp với hệ thống có nhiều module riêng lẻ nhưng cần giao tiếp với nhau.

📌 Ưu điểm:

✔ Tái sử dụng cao: Các dịch vụ có thể tái sử dụng cho nhiều ứng dụng khác nhau, giúp giảm chi phí phát triển.  
✔ Tính linh hoạt: Dễ dàng tích hợp với các hệ thống bên thứ ba và hỗ trợ nhiều giao thức (SOAP, REST, AMQP…).  
✔ Bảo mật tốt: Có thể áp dụng các cơ chế bảo mật tập trung như OAuth2, SAML để bảo vệ dữ liệu giữa các dịch vụ.  
✔ Dễ mở rộng: Khi có yêu cầu mới, chỉ cần bổ sung dịch vụ mới mà không ảnh hưởng đến toàn bộ hệ thống.  
✔ Khả năng chịu lỗi cao: Nếu một dịch vụ gặp sự cố, các dịch vụ khác vẫn có thể hoạt động bình thường.

⚠ Nhược điểm:

❌ Độ phức tạp cao: Cần nhiều thành phần hỗ trợ như Service Bus (ESB), API Gateway để điều phối dịch vụ.  
❌ Hiệu suất có thể bị ảnh hưởng: Vì các dịch vụ giao tiếp qua mạng (HTTP, SOAP), tốc độ có thể chậm hơn so với kiến trúc monolithic.  
❌ Khó quản lý: Cần có cơ chế logging, monitoring (như ELK Stack, Prometheus) để theo dõi hoạt động của từng dịch vụ.  
❌ Chi phí triển khai cao: Yêu cầu nhiều công nghệ và tài nguyên hơn so với các kiến trúc đơn giản như Monolithic.

📌 Khi nào nên chọn SOA?

* Khi hệ thống có nhiều module hoạt động độc lập nhưng vẫn cần giao tiếp với nhau.
* Khi cần tích hợp với hệ thống bên ngoài (ví dụ: API của bệnh viện khác, bảo hiểm y tế…).
* Khi yêu cầu bảo mật và quản lý truy cập chặt chẽ.

8. Microservice Architecture – Kiến trúc Microservice

📌 Bài toán:

Sàn thương mại điện tử giống Shopee

* Hàng triệu người dùng, nhiều tính năng như giỏ hàng, thanh toán, theo dõi đơn hàng.

🏗 Cách triển khai:

* Chia hệ thống thành nhiều service nhỏ:
  + UserService: Quản lý tài khoản.
  + ProductService: Quản lý sản phẩm.
  + OrderService: Quản lý đơn hàng.
  + PaymentService: Xử lý thanh toán.
  + NotificationService: Gửi email, SMS.

🔧 Công nghệ phù hợp:

* Backend: Spring Boot (Java), NestJS (Node.js)
* API Gateway: Kong, Nginx
* Database: MongoDB (sản phẩm), MySQL (người dùng, đơn hàng)
* Containerization: Docker, Kubernetes

📌 Ưu điểm:

✔ Dễ mở rộng từng phần của hệ thống: Mỗi microservice có thể được mở rộng độc lập tùy theo nhu cầu thực tế (ví dụ: OrderService có thể scale nhiều hơn UserService khi có nhiều giao dịch).  
✔ Mỗi dịch vụ có thể triển khai độc lập: Các nhóm phát triển có thể làm việc trên từng service riêng biệt mà không ảnh hưởng đến toàn bộ hệ thống.  
✔ Cải thiện độ tin cậy và khả năng chịu lỗi: Nếu một service gặp sự cố (ví dụ: PaymentService bị lỗi), các service khác vẫn có thể hoạt động bình thường.  
✔ Công nghệ linh hoạt: Mỗi service có thể sử dụng công nghệ, ngôn ngữ lập trình và database phù hợp nhất với yêu cầu của nó (ví dụ: MongoDB cho sản phẩm, MySQL cho đơn hàng).  
✔ Tối ưu hiệu suất: Có thể tối ưu riêng từng microservice theo nhu cầu, chẳng hạn như caching cho ProductService hoặc message queue cho NotificationService.  
✔ Dễ tích hợp với hệ thống bên thứ ba: Microservices có thể giao tiếp với các API bên ngoài dễ dàng (ví dụ: cổng thanh toán, dịch vụ vận chuyển).

⚠ Nhược điểm:

❌ Quản lý phức tạp hơn so với Monolithic: Vì có nhiều service riêng biệt, cần có hệ thống giám sát, logging và tracing để theo dõi hoạt động của từng service.  
❌ Tăng độ trễ khi giao tiếp giữa các service: Các microservices thường giao tiếp qua HTTP, gRPC hoặc message queue, có thể làm tăng độ trễ so với kiến trúc Monolithic.  
❌ Yêu cầu DevOps mạnh để quản lý container: Cần có CI/CD, Kubernetes, Docker, API Gateway để triển khai và quản lý toàn bộ hệ thống.  
❌ Dữ liệu bị phân tán, khó nhất quán: Mỗi service có database riêng nên cần các kỹ thuật như Saga Pattern hoặc Eventual Consistency để đảm bảo tính nhất quán dữ liệu.  
❌ Tốn nhiều tài nguyên hơn: Mỗi microservice cần chạy trên một container riêng, tiêu tốn nhiều tài nguyên hơn so với một hệ thống Monolithic duy nhất.

📌 Khi nào nên chọn Microservice Architecture?

* Khi xây dựng ứng dụng lớn với nhiều tính năng (ví dụ: sàn thương mại điện tử, nền tảng mạng xã hội, ứng dụng SaaS).
* Khi có đội ngũ phát triển lớn, làm việc theo nhiều nhóm và cần triển khai nhanh chóng các tính năng mới.
* Khi cần mở rộng linh hoạt, có thể scale từng phần mà không ảnh hưởng đến toàn hệ thống.
* Khi muốn tích hợp nhiều công nghệ khác nhau, sử dụng ngôn ngữ và database phù hợp với từng dịch vụ cụ thể.

📌 Tình huống thực tế phù hợp:

✅ Sàn thương mại điện tử lớn  
✅ Ứng dụng có nhiều user và yêu cầu mở rộng linh hoạt