**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.
* Phù hợp với ứng dụng web và mobile.

**❌ Nhược điểm:**

* Nếu server bị lỗi, toàn bộ hệ thống bị ngừng hoạt động.
* Khi có quá nhiều kết nối, cần tối ưu server (cân bằng tải, cache).

**📌 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 dịch vụ có thể được triển khai độc lập.

**❌ Nhược điểm:**

* Quản lý phức tạp hơn so với monolithic.
* Cần DevOps mạnh để quản lý container.

**📌 Tình huống thực tế phù hợp:**

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