Tìm hiểu quá trình phát triển kiến trúc phần mềm

Họ và tên:	Lớp:	MSSV:	GVHD:
Trần Vĩnh Huy	DCT122C2	3122411072	TS. Đỗ Như Tài

Giới thiệu

Phần báo cáo chính

A. Câu hỏi trắc nghiệm

a) Kiến thức cơ bản về kiến trúc phần mềm

1. Kiến trúc nào dưới đây không thuộc các loại kiến trúc phần mềm phổ biến	**D. Quantum Architecture**
2. Nguyên tắc "YAGNI" đề cập đến điều gì?	**A. Không thêm các tính năng mà bạn sẽ không cần đến**
3. Khi nào nên sử dụng kiến trúc monolithic thay vì microservices?	**B. Khi ứng dụng nhỏ và ít thay đổi**

b) Thiết kế và triển khai microservices

4. Trong kiến trúc microservices, mô hình "Database-per-Service Pattern" đề xuất:	**B. Mỗi dịch vụ có cơ sở dữ liệu riêng của mìn**
5. Lợi ích chính của việc sử dụng gRPC trong microservices là gì?	**B. Hỗ trợ truyền dữ liệu nhanh với mã hóa nhị phân**
6. Pattern nào dùng để xử lý truy vấn dữ liệu và lệnh ghi riêng biệt trong microservices?	**B. CQRS (Command Query Responsibility Segregation)**

c) Xử lý giao tiếp và dữ liệu

7. Pattern nào giúp giảm độ trễ trong giao tiếp giữa các microservices?	**B. Publish/Subscribe Messaging**

8. “Event Sourcing Pattern” hoạt động như thế nào?	**B. Lưu trữ tất cả các sự kiện thay đổi trạng thái hệ thống**
9. Trong mô hình giao dịch phân tán, SAGA Pattern sử dụng:	**B. Các bước nhỏ độc lập với khả năng khôi phục khi thất bại**

d) Resilience và triển khai hệ thống

10. Pattern nào không thuộc các chiến lược resilience phổ biến?	**D. Scale Cube**
11. Chiến lược triển khai nào đảm bảo không có thời gian chết khi cập nhật hệ thống?	**A. Blue-Green Deployment**
12. Công cụ nào không được sử dụng để giám sát và theo dõi microservices?	**C. Kubernetes**

e) Các bài toán thực tế

13. Khi nào nên sử dụng kiến trúc event-driven?	**A. Khi cần xử lý dữ liệu thời gian thực và khối lượng lớn**
14. Giải pháp nào phù hợp nhất để xử lý vấn đề bottleneck ở cơ sở dữ liệu trong microservices?	**B. Triển khai caching phân tán**
15. CQRS giúp ích gì trong kiến trúc microservices?	**B. Tăng hiệu năng khi xử lý truy vấn và ghi dữ liệu**

B. Câu hỏi lý thuyết

a) Kiến thức cơ bản về kiến trúc phần mềm

Câu 1. Các kiến trúc phần mềm chính bao gồm những loại nào? So sánh ưu và nhược điểm của chúng.
Các kiến trúc phần mềm phổ biến hiện nay gồm **Monolithic Architecture, Layered Architecture và Microservices Architecture.** **Monolithic Architecture** là kiến trúc trong đó toàn bộ hệ thống được xây dựng và triển khai như một khối thống nhất, có ưu điểm là đơn giản, dễ phát triển và triển khai ở giai đoạn đầu, tuy nhiên khi hệ thống mở rộng sẽ khó bảo trì và khó mở rộng từng phần riêng biệt. **Layered Architecture** chia hệ thống thành các tầng chức năng như giao diện, xử lý nghiệp vụ và dữ liệu, giúp tổ chức mã nguồn rõ ràng và dễ hiểu, nhưng các tầng phụ thuộc lẫn nhau nên việc thay đổi hoặc mở rộng thường kém linh hoạt. **Microservices Architecture** chia hệ thống thành nhiều dịch vụ nhỏ, độc lập, cho phép mở rộng linh hoạt và phù hợp với hệ thống lớn, tuy nhiên kiến trúc này phức tạp trong triển khai, vận hành và đòi hỏi chi phí cao hơn.

Câu 2. Nguyên tắc KISS, YAGNI và Separation of Concerns có vai trò gì trong thiết kế phần mềm?
Nguyên tắc **KISS (Keep It Simple, Stupid)** nhấn mạnh việc giữ cho thiết kế hệ thống đơn giản nhất có thể nhằm giảm độ phức tạp và nguy cơ phát sinh lỗi. Nguyên tắc **YAGNI (You Aren't Gonna Need It)** khuyên rằng không nên triển khai những tính năng chưa có nhu cầu thực tế, tránh lãng phí thời gian và tài nguyên. **Separation of Concerns (SoC)**giúp tách biệt các mối quan tâm khác nhau trong hệ thống, từ đó làm cho mã nguồn dễ hiểu, dễ bảo trì và dễ mở rộng. Ba nguyên tắc này đóng vai trò quan trọng trong việc xây dựng các hệ thống phần mềm bền vững và hiệu quả.
Câu 3. Khi nào nên chọn kiến trúc monolithic và khi nào nên chuyển sang microservices?
Kiến trúc monolithic phù hợp với các hệ thống nhỏ, dự án mới hoặc MVP, nơi nhóm phát triển ít người và yêu cầu thay đổi chưa nhiều. Trong khi đó, microservices nên được áp dụng khi hệ thống phát triển lớn, có nhiều nhóm làm việc song song, yêu cầu mở rộng linh hoạt và tần suất triển khai cao. Trên thực tế, nhiều hệ thống được khuyến nghị bắt đầu bằng monolithic để giảm độ phức tạp, sau đó dần chuyển sang microservices khi xuất hiện các yêu cầu mới về quy mô và hiệu năng.

b) Thiết kế và triển khai microservice

Câu 4. Database-per-Service Pattern là gì? Ưu và nhược điểm của mô hình này trong microservices.
Database-per-Service Pattern là mô hình trong đó mỗi microservice sở hữu và quản lý một cơ sở dữ liệu riêng biệt. Mô hình này giúp giảm sự phụ thuộc giữa các dịch vụ, tăng tính độc lập và cho phép mỗi dịch vụ tối ưu cơ sở dữ liệu theo nhu cầu riêng. Tuy nhiên, nhược điểm của nó là việc đảm bảo tính nhất quán dữ liệu trở nên khó khăn hơn và các truy vấn liên quan đến nhiều dịch vụ trở nên phức tạp.
Câu 5. Các cách tiếp cận phân mảnh dữ liệu (Data Partitioning) gồm những loại nào?
Các cách phân mảnh dữ liệu phổ biến bao gồm phân mảnh theo hàng, phân mảnh theo cột và phân mảnh theo chức năng. Phân mảnh theo hàng chia dữ liệu thành nhiều phần dựa trên giá trị khóa, phân mảnh theo cột tách các nhóm thuộc tính khác nhau, còn phân mảnh theo chức năng chia dữ liệu dựa trên từng dịch vụ hoặc nghiệp vụ cụ thể. Các cách tiếp cận này nhằm mục tiêu cải thiện hiệu năng và khả năng mở rộng của hệ thống.

Câu 6. Giải thích các mô hình giao tiếp RESTful API, gRPC và WebSocket.
RESTful API là mô hình giao tiếp phổ biến sử dụng HTTP và định dạng JSON, dễ triển khai và dễ hiểu nhưng có hiệu năng không cao. **gRPC** là mô hình giao tiếp hiệu năng cao sử dụng giao thức nhị phân, phù hợp cho giao tiếp giữa các microservices nhưng khó debug hơn REST. **WebSocket** cho phép giao tiếp hai chiều theo thời gian thực, phù hợp với các ứng dụng cần cập nhật liên tục nhưng việc triển khai và mở rộng phức tạp hơn. Trên thực tế, nhiều hệ thống được khuyến nghị bắt đầu bằng monolithic để giảm độ phức tạp, sau đó dần chuyển sang microservices khi xuất hiện các yêu cầu mới về quy mô và hiệu năng.

c) Xử lý dữ liệu và giao dịch

Câu 7. CQRS là gì? Khi nào nên áp dụng mô hình này?
CQRS là mô hình thiết kế tách biệt việc lệnh ghi (Command) và truy vấn đọc (Query) thành hai phần riêng biệt. Mô hình này phù hợp với các hệ thống có số lượng truy vấn đọc lớn và yêu cầu tối ưu hiệu năng, tuy nhiên không nên áp dụng cho hệ thống nhỏ vì làm tăng độ phức tạp.
Câu 8. So sánh SAGA Pattern với giao dịch truyền thống trong hệ thống phân tán.
Giao dịch truyền thống đảm bảo tính nguyên tử và nhất quán mạnh mẽ nhưng không phù hợp với kiến trúc microservices do làm giảm khả năng mở rộng. **SAGA Pattern** chia giao dịch thành nhiều bước nhỏ, mỗi bước có cơ chế hoàn tác khi xảy ra lỗi, giúp hệ thống phân tán duy trì tính nhất quán ở mức chấp nhận được.
Câu 9. Sự khác biệt giữa Event Sourcing Pattern và Transactional Outbox Pattern là gì?
Event Sourcing Pattern lưu trữ toàn bộ các sự kiện thay đổi trạng thái của hệ thống và tái tạo trạng thái hiện tại từ các sự kiện đó. **Transactional Outbox Pattern** tập trung vào việc đảm bảo dữ liệu và sự kiện được ghi đồng bộ để tránh mất mát sự kiện khi xảy ra lỗi.

d) Resilience và triển khai hệ thống

Câu 10. Mô hình triển khai nào được sử dụng để giảm thiểu thời gian chết khi triển khai microservices?
Các mô hình triển khai như Blue–Green, Rolling và Canary đều được sử dụng để giảm downtime. Trong đó, Blue–Green Deployment giúp đảm bảo không có thời gian chết rõ ràng nhất bằng cách chuyển traffic giữa hai môi trường.
Câu 11. Các resilience patterns phổ biến trong microservices bao gồm những gì?
Các resilience patterns phổ biến trong microservices bao gồm Circuit Breaker, Retry, Timeout và Bulkhead, nhằm mục tiêu ngăn lỗi lan rộng, tăng khả năng chịu lỗi và đảm bảo hệ thống hoạt động ổn định.

Câu 12. Làm thế nào để giám sát microservices bằng Prometheus, Grafana và Elastic Stack?
Prometheus được sử dụng để thu thập các chỉ số hệ thống, **Grafana** hiển thị dữ liệu dưới dạng biểu đồ trực quan, trong khi **Elastic Stack** hỗ trợ lưu trữ và phân tích log, giúp theo dõi và phát hiện sự cố hiệu quả.

e) Các bài toán thực tế

Câu 13. Giải pháp nào giúp tăng hiệu năng khi giao tiếp giữa các microservices?
Hiệu năng giao tiếp giữa các microservices có thể được cải thiện bằng cách sử dụng cache phân tán, giảm số lượng API call, áp dụng giao tiếp bất đồng bộ và sử dụng các giao thức hiệu năng cao như gRPC.
Câu 14. Làm thế nào để xử lý bài toán hiệu suất cho ứng dụng xử lý hàng triệu sự kiện mỗi giây?
Để xử lý khối lượng lớn sự kiện, hệ thống nên áp dụng kiến trúc event-driven, sử dụng message broker và mở rộng theo chiều ngang thông qua cơ chế phân vùng và nhóm xử lý song song.
Câu 15. Khi nào cần sử dụng kiến trúc event-driven thay vì request–response?
Kiến trúc event-driven phù hợp khi hệ thống cần xử lý bất đồng bộ, có nhiều dịch vụ liên quan và không yêu cầu phản hồi ngay lập tức, chẳng hạn như xử lý log, gửi thông báo hoặc phân tích dữ liệu.

f) Tư duy thiết kế

Câu 16. Tại sao mỗi quyết định thiết kế cần được biện minh bởi yêu cầu kinh doanh?
Mỗi quyết định thiết kế cần gắn với yêu cầu kinh doanh nhằm đảm bảo hệ thống mang lại giá trị thực tế, tránh lãng phí tài nguyên và hạn chế tình trạng thiết kế quá phức tạp so với nhu cầu.
Câu 17. Over-engineering là gì và làm thế nào để tránh?
Over-engineering là việc thiết kế hệ thống phức tạp hơn mức cần thiết, thường dẫn đến chi phí cao và khó bảo trì. Có thể tránh tình trạng này bằng cách áp dụng các nguyên tắc KISS, YAGNI và phát triển hệ thống theo từng giai đoạn dựa trên nhu cầu thực tế.