BÀI 7 – CÁC CÔNG NGHỆ LẬP TRÌNH HIỆN ĐẠI

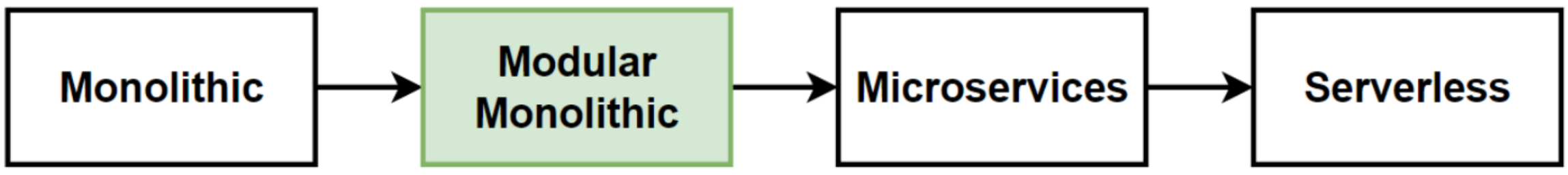
Kiến Trúc Nguyên Khối Module (Modular Monolithic)

Lợi ích và thách thức của kiến trúc nguyên khối Module
Nguyên tắc phụ thuộc của kiến trúc nguyên khối Module
Thiết kế ứng dụng thương mại điện tử với kiến trúc nguyên khối module

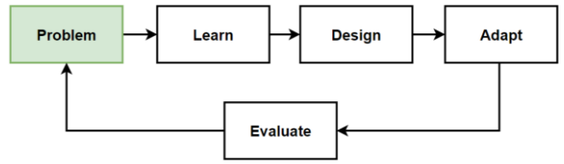
TS. Đỗ Như Tài
Đại Học Sài Gòn
dntai@sgu.edu.vn

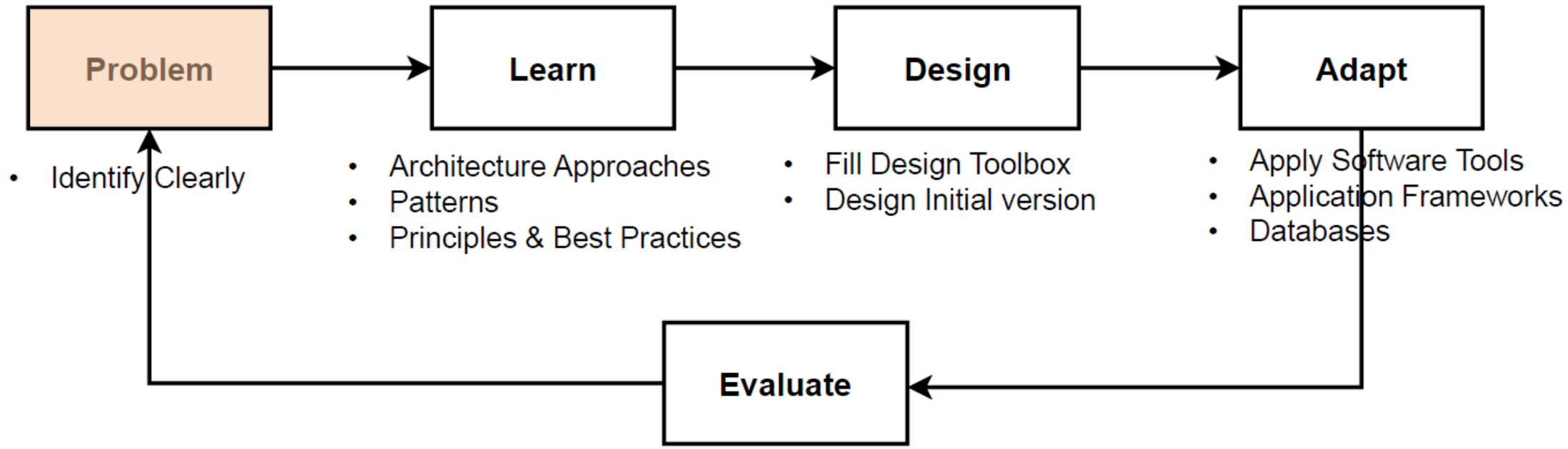
Hành trình Thiết kế Kiến trúc Phần mềm

- **Monolithic** – Kiến trúc nguyên khối truyền thống, mọi thành phần nằm chung một khối.
- **Modular Monolithic** – Phân chia thành các mô-đun độc lập bên trong khối, dễ bảo trì hơn.
- **Microservices** – Chuyển mỗi mô-đun thành một dịch vụ riêng biệt, giao tiếp qua API.
- **Serverless** – Triển khai các chức năng độc lập mà không cần quản lý máy chủ, tối ưu tài nguyên và chi phí.



Quy trình Thiết kế Kiến trúc Phần Mềm

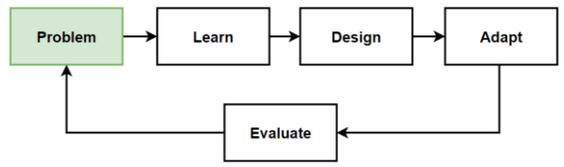




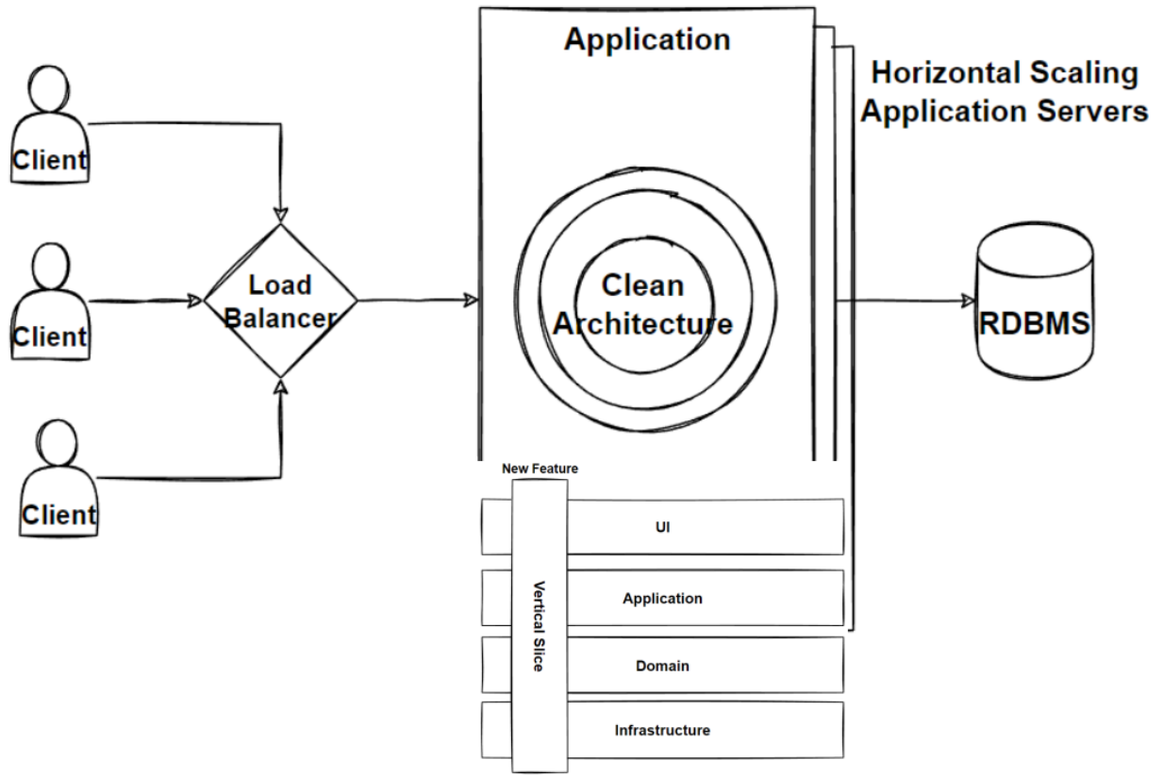
- What's wrong with this architecture ?
- how we can improve this current design ?

Quy trình Thiết kế Kiến trúc Phần Mềm

Vấn đề: Tính linh hoạt khi phát triển tính năng mới, nhóm Agile tách biệt



- **Các vấn đề**:
  - Doanh nghiệp thương mại điện tử đang phát triển.
  - Các nhóm kinh doanh được tách biệt theo phòng ban: Sản phẩm, Bán hàng, Thanh toán...
  - Tất cả các nhóm đều muốn thêm tính năng mới để cạnh tranh trên thị trường.
  - Mã nguồn hiện tại không cho phép quản lý điều đó.
  - Việc chuyển đổi ngữ cảnh và mô hình cắt lớp dọc (Vertical Slices) gây ra vấn đề trong kiến trúc Clean.

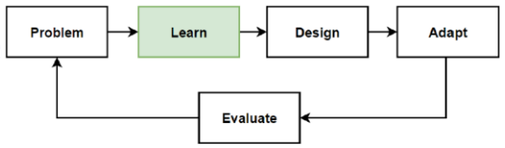


Vấn đề phân cắt dọc trong kiến trúc Clean

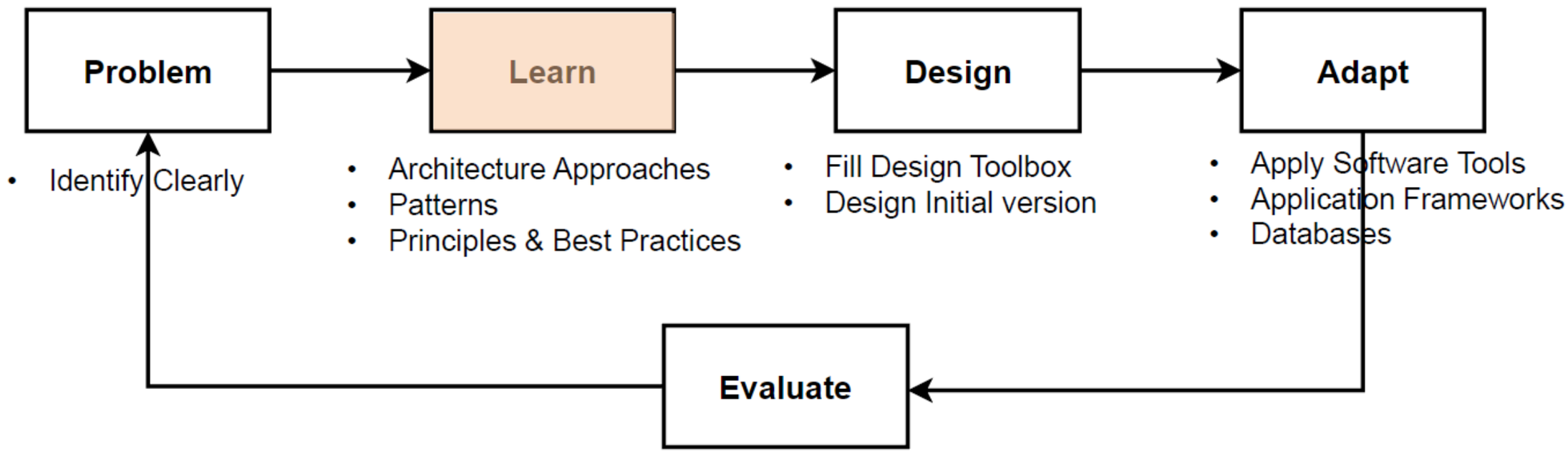
- **Giải pháp**:
  - **Áp dụng kiến trúc nguyên khối theo mô-đun** (Modular Monolithic Architecture)

Do hệ thống đang phát triển và các nhóm nghiệp vụ độc lập cần triển khai tính năng nhanh chóng, kiến trúc Clean Architecture hiện tại gây khó khăn vì độ phức tạp và phụ thuộc giữa các phần. Modular Monolithic giúp chia tách chức năng thành mô-đun riêng biệt trong cùng một khối ứng dụng, giúp các nhóm có thể làm việc song song hiệu quả hơn, giảm xung đột và dễ mở rộng.

Quy trình Thiết kế Kiến trúc Phần Mề

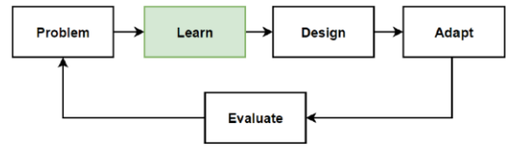


- Kiến trúc nguyên khối module
  - Tình huống sử dụng
  - Lợi ích
  - Thác thức
  - Ưu điểm – Nhược điểm
  - Mẫu tham khảo

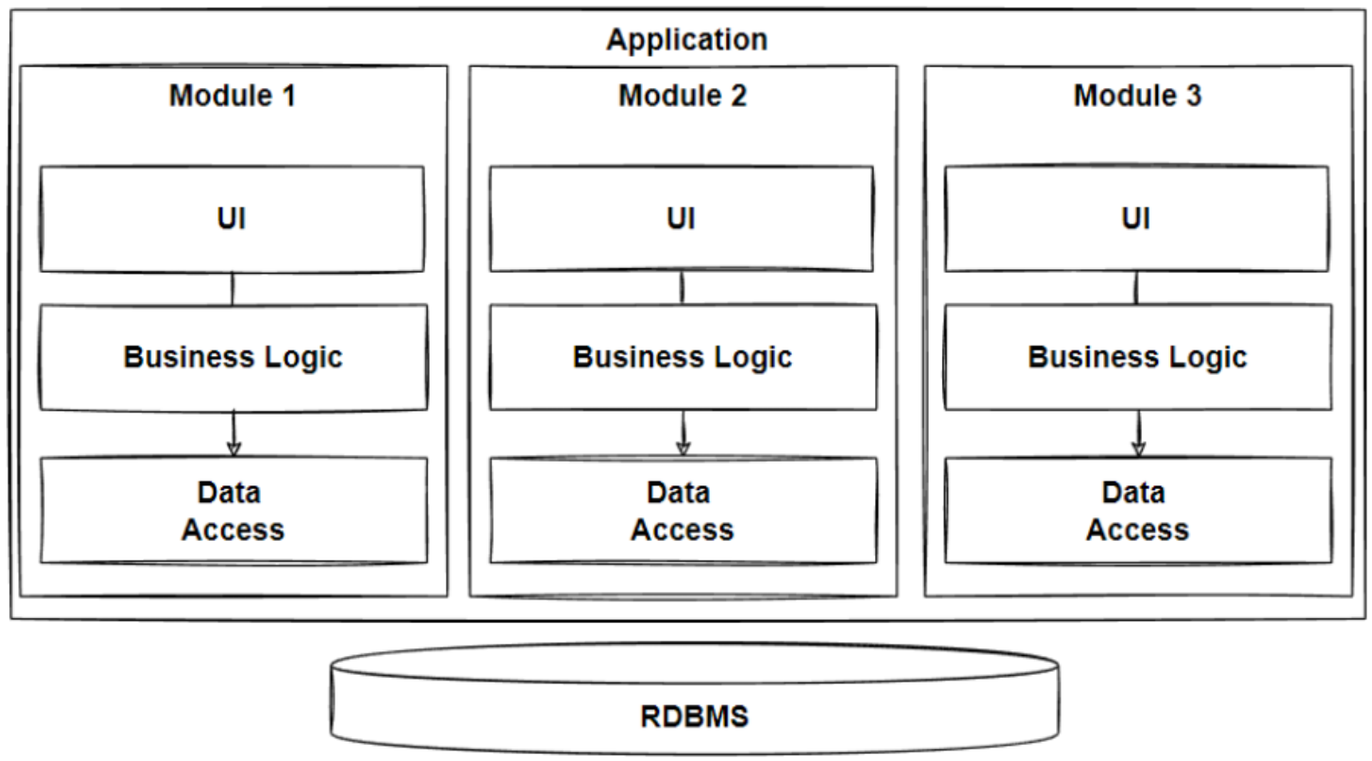


Quy trình Thiết kế Kiến trúc Phần Mềm

Kiến trúc nguyên khối module là gì?

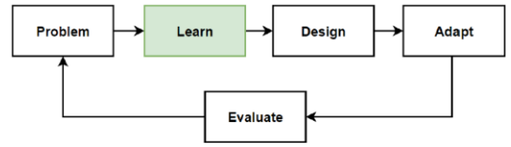


- Kiến trúc nguyên khối module chia logic của ứng dụng thành **các module**, mỗi module sẽ **độc lập**, **tách biệt** và có **logic nghiệp vụ riêng**, lược đồ cơ sở dữ liệu riêng.

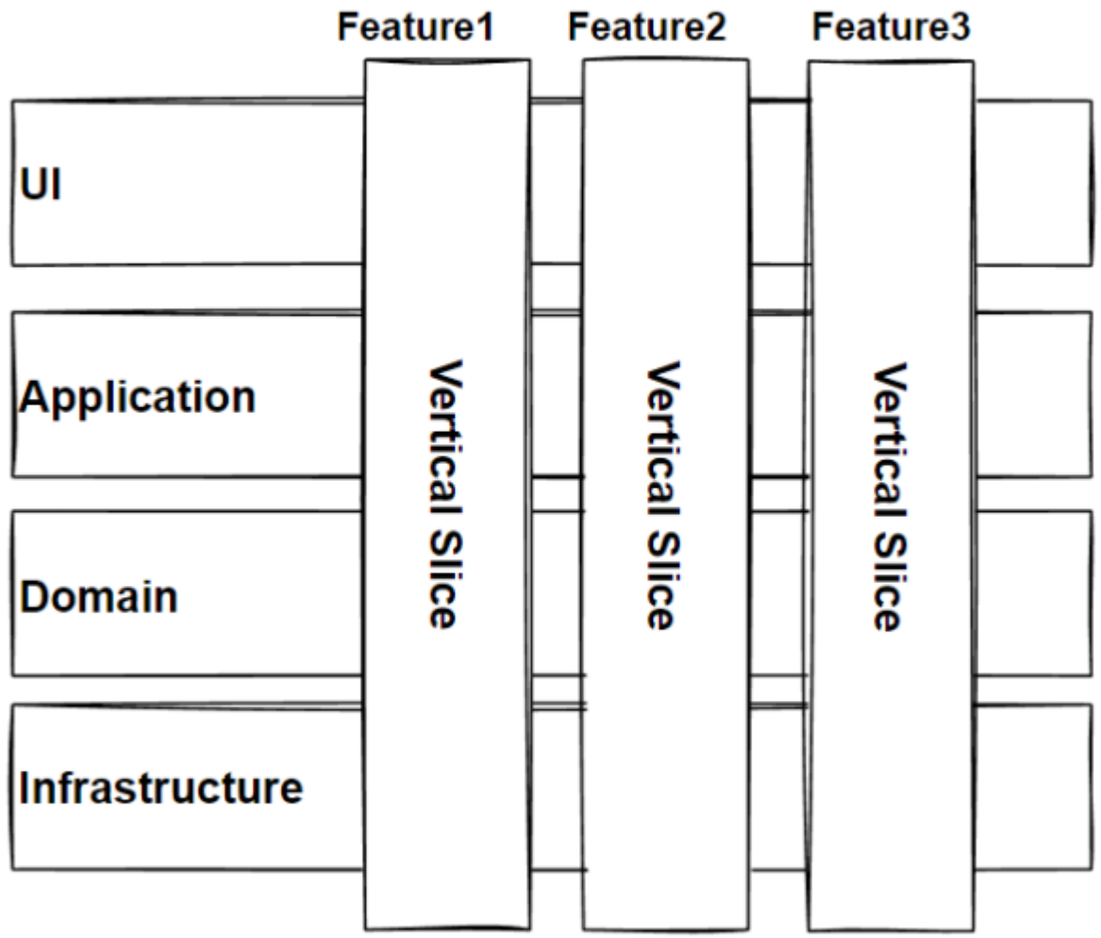


- Mỗi module có thể tuân theo **sự phân tách logic riêng**, có thể là kiến trúc phân lớp hoặc kiến trúc sạch (clean architecture).
- Kiến trúc nguyên khối module chia nhỏ mã nguồn thành **các module độc lập**, mỗi module **đóng gói** các tính năng riêng của nó.
- Module đại diện cho **ngữ cảnh giới hạn (bounded context)** trong miền ứng dụng, và chúng ta **gom nhóm các chức năng theo ngữ cảnh nghiệp vụ** trong module.
- Giảm **phụ thuộc** giữa các module và cho phép phát triển hoặc chỉnh sửa module **mà không ảnh hưởng đến các module khác**.

Cắt dọc kiến trúc nguyên khối module

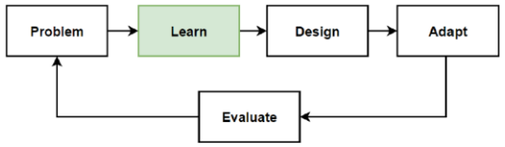


- Thay vì dùng kiến trúc phân lớp với các lớp logic ngang, có thể **tổ chức mã nguồn theo lát cắt dọc** dựa trên **chức năng nghiệp vụ**.
- Những lát cắt này được xác định dựa trên **nhu cầu kinh doanh**, thay vì bị ràng buộc bởi các giới hạn kỹ thuật.
- Khi thêm hoặc thay đổi một tính năng, thay đổi đó được **giới hạn trong phạm vi nghiệp vụ**, không ảnh hưởng đến toàn bộ các lớp logic kỹ thuật.

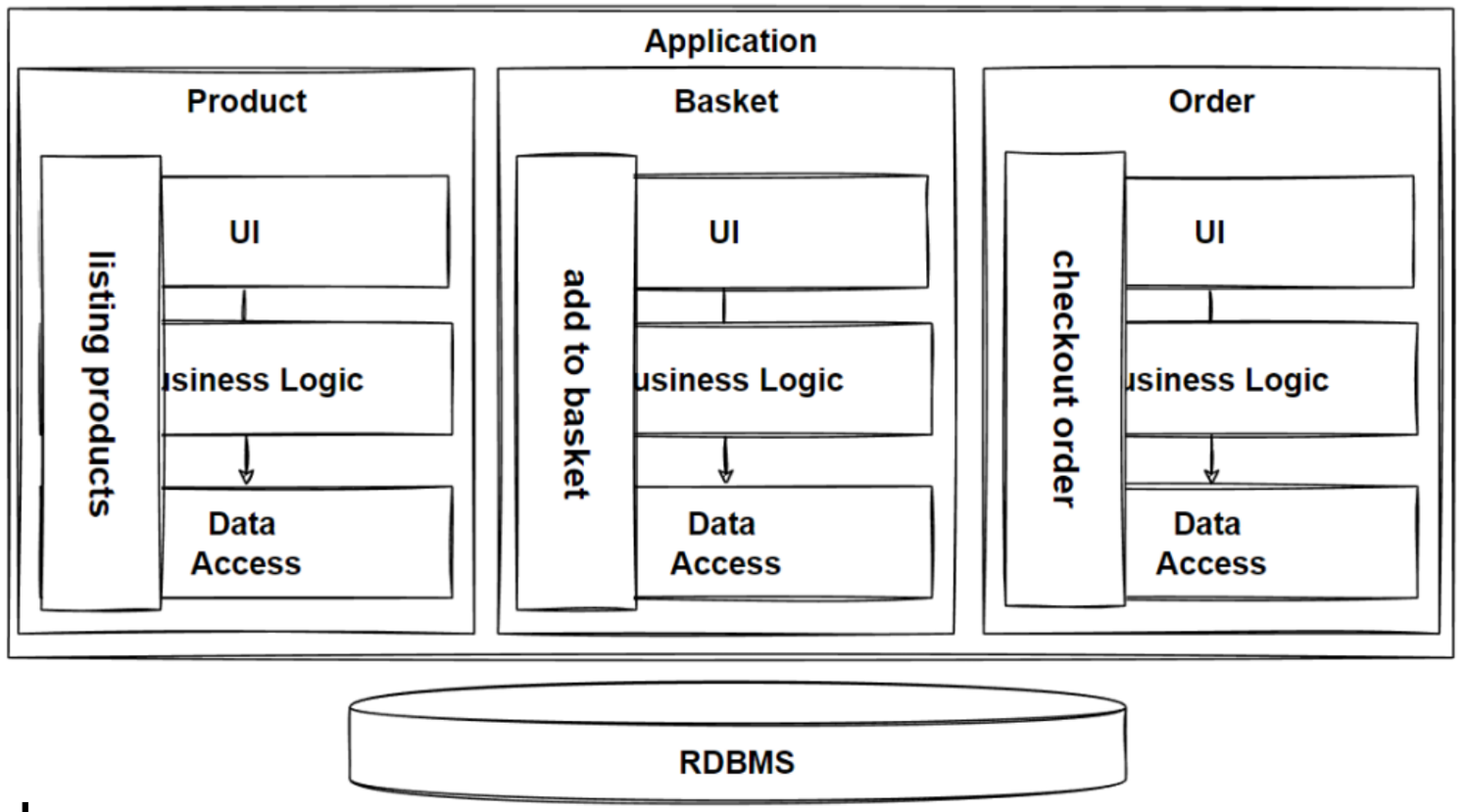


- Với kiến trúc này, tổ chức mã nguồn thành **lát cắt dọc**, và khi hệ thống tiếp tục mở rộng, có thể **tổ chức mã nguồn xung quanh các chức năng nghiệp vụ** thành các module.
- Các **mô-đun có thể trở thành microservice tiềm năng** khi cần triển khai độc lập và mở rộng trong các lần **tái cấu trúc kiến trúc trong tương lai**.

Kiến trúc nguyên khối module trong thương mại điện tử

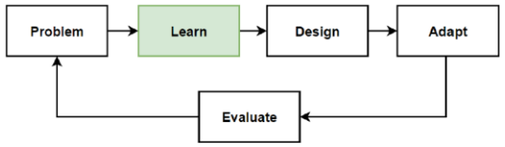


- Tình huống sử dụng
  - **Liệt kê sản phẩm** → thuộc **module Sản phẩm**
  - **Thêm vào giỏ hàng** → thuộc **module Giỏ hang**
  - **Thanh toán đơn hàng** → thuộc **module Đơn hàng**



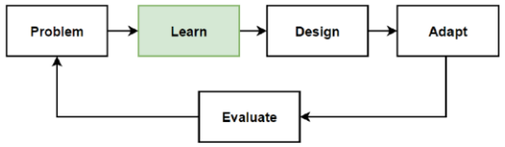
- Lợi ích chính
  - Giảm độ phức tạp
  - Dễ tái cấu trúc mã nguồn
  - Phù hợp hơn với mô hình làm việc theo nhóm

Lợi ích của kiến trúc nguyên khối module



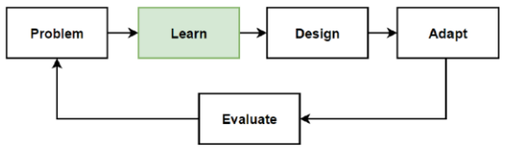
- **Đóng gói logic nghiệp vụ**
  - Logic nghiệp vụ được đóng gói trong các mô-đun, giúp tái sử dụng cao, đồng thời dữ liệu vẫn được nhất quán và mô hình giao tiếp đơn giản.
- **Mã có thể tái sử dụng, dễ tái cấu trúc**
  - Với các nhóm phát triển lớn, việc phát triển các mô-đun riêng biệt giúp tăng khả năng tái sử dụng. Các thành phần mô-đun có thể được tái sử dụng để tạo ra một nguồn chân lý duy nhất cho toàn đội.
- **Phụ thuộc được tổ chức tốt hơn**
  - Trong kiến trúc nguyên khối có mô-đun, các phụ thuộc trong ứng dụng sẽ được tổ chức rõ ràng và dễ nhìn thấy. Điều này giúp lập trình viên dễ xác định phần nào của ứng dụng phụ thuộc vào phần nào.
- **Ít phức tạp hơn so với kiến trúc Microservices**
  - Dễ quản lý hơn so với hàng trăm microservice nhỏ lẻ, bởi vì kiến trúc Modular Monolithic có cơ sở hạ tầng đơn giản và chi phí vận hành thấp.
- **Tốt hơn cho nhóm phát triển**
  - Dễ dàng cho các lập trình viên làm việc trên các phần khác nhau của mã nguồn. Với kiến trúc Modular Monolithic, chúng ta có thể chia nhóm phát triển hiệu quả và triển khai các yêu cầu nghiệp vụ với ảnh hưởng tối thiểu đến nhau.

Thách thức của kiến trúc nguyên khối module



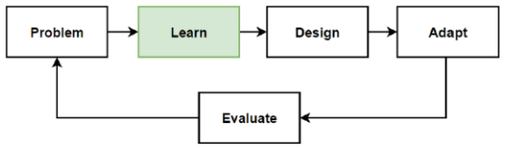
- **Không thể đa dạng hóa công nghệ**
  - Kiến trúc nguyên khối module không mang lại đầy đủ lợi ích như microservices. Nếu bạn cần đa dạng hóa công nghệ hoặc lựa chọn ngôn ngữ khác nhau, bạn không thể làm điều đó với kiến trúc này. Những loại công nghệ hỗn hợp (polyglot technology stacks) không thể sử dụng trong mô hình này.
- **Không thể mở rộng và triển khai độc lập**
  - Vì toàn bộ ứng dụng là một khối đơn lẻ, nên không thể mở rộng từng phần riêng biệt hoặc triển khai độc lập như microservices. Các ứng dụng dạng này thường phải chuyển sang microservices khi gặp giới hạn về khả năng mở rộng hoặc hiệu năng.

Tình huống dùng kiến trúc nguyên khối module

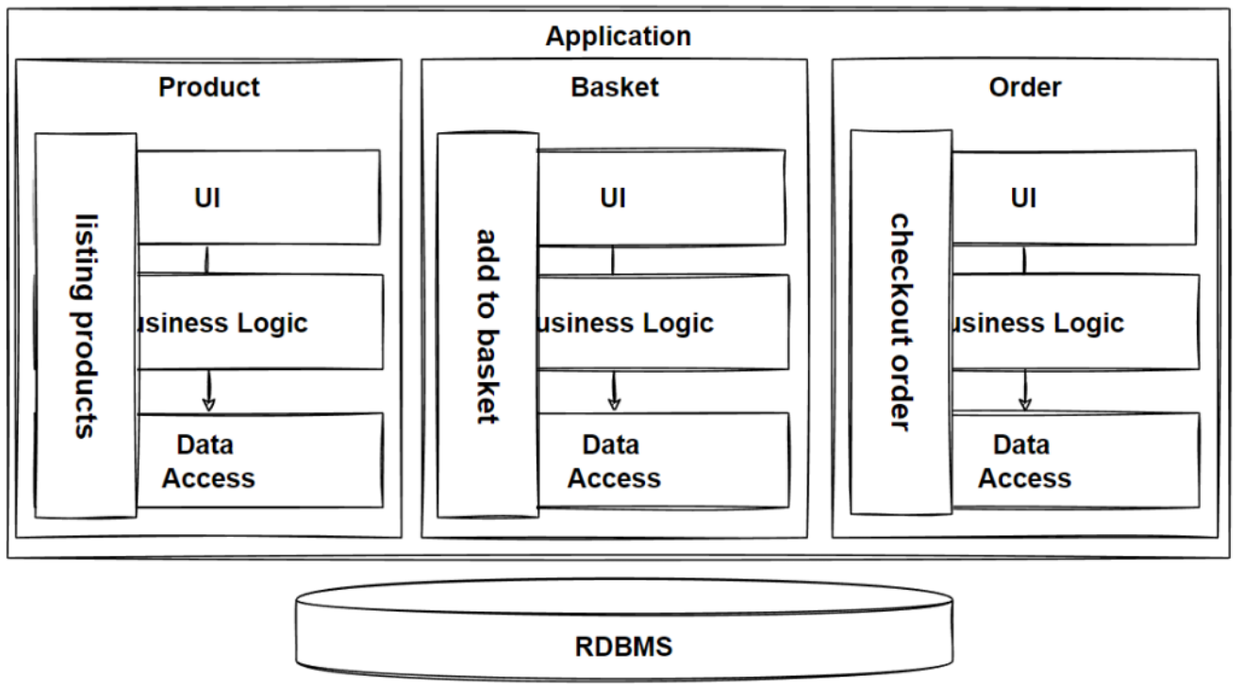


- **Trường hợp yêu cầu tính nhất quán nghiêm ngặt**
  - Với nhiều công ty không thể chuyển sang microservices do hệ thống cơ sở dữ liệu không phù hợp với kiến trúc phân tán, modular monolith trở nên lý tưởng.
  - Ví dụ: nếu ứng dụng của bạn lưu trữ **dữ liệu quan trọng cao** như ghi nợ tài khoản ngân hàng, bạn cần **tính nhất quán dữ liệu mạnh mẽ** – tức dữ liệu phải luôn chính xác. Nếu có lỗi xảy ra, phải khôi phục (rollback) ngay lập tức.
- **Hiện đại hóa hệ thống**
  - Nếu bạn đã có một ứng dụng nguyên khối lớn và phức tạp đang chạy, modular monolith là kiến trúc lý tưởng để hỗ trợ bạn **refactor mã nguồn** và chuẩn bị cho khả năng chuyển đổi sang kiến trúc microservices.
  - Thay vì **nhảy ngay vào microservices**, bạn có thể chuyển dần sang modular monolith để nhận được những lợi ích như hiệu năng cao với một khối kiến trúc được tổ chức tốt.
- **Dự án mới (Green Field Projects)**
  - Modular monolith cho phép bạn tìm hiểu miền nghiệp vụ và xây dựng kiến trúc nhanh hơn nhiều so với microservices.
  - Bạn **không cần lo lắng** về các vấn đề như **Kubernetes** hay **service mesh** ngay từ ngày đầu. Việc triển khai hệ thống sẽ **đơn giản hơn rất nhiều**.

Tiếp cận "Monolith First" - Martin Fowler

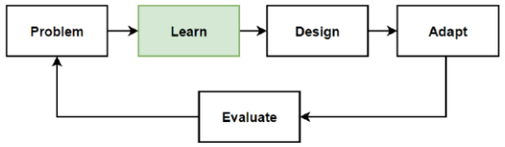


- Nguồn gốc
  - **Bài viết**: Martin Fowler, *Monolithic First*, 2015
  - https://martinfowler.com/bliki/MonolithFirst.html
- **Thông điệp chính**
  - Nhiều hệ thống microservices thành công thực tế đều bắt đầu từ một ứng dụng nguyên khối (monolith).



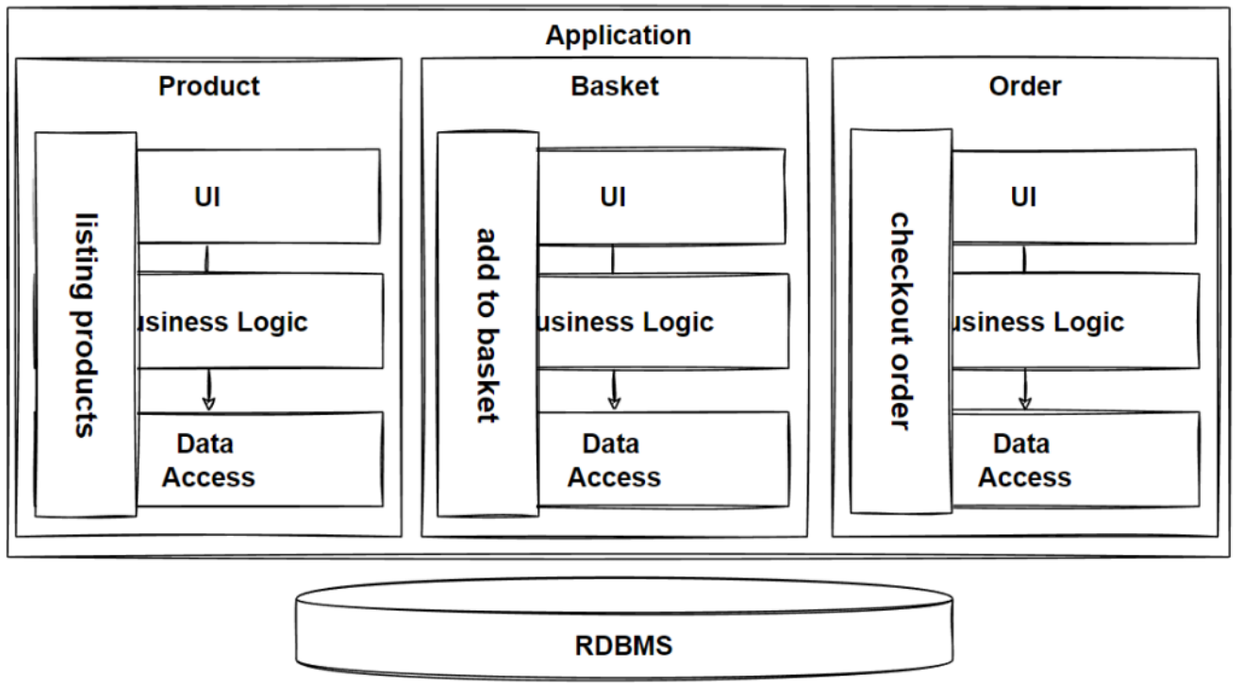
  - Những hệ thống được **xây dựng microservices từ đầu** thường gặp **vấn đề nghiêm trọng** do độ phức tạp tăng cao.
  - **Microservices** là một kiến trúc hữu ích nhưng mang lại **độ phức tạp đáng kể** – chỉ nên áp dụng với **hệ thống lớn và phức tạp**.
  - Hãy cân nhắc nguyên tắc **YAGNI** (*You Aren't Gonna Need It*): Khi bắt đầu một ứng dụng mới, bạn có chắc người dùng thực sự cần ngay microservices?
  - **Chỉ nên bắt đầu với microservices** khi bạn có thể xác định rõ **ranh giới ổn định giữa các dịch vụ**, chính là các **Bounded Contexts** trong Domain-Driven Design.
- Lời khuyên
  - "Hãy bắt đầu với kiến trúc nguyên khối (Modular Monolith), sau đó mới tiến hóa sang Microservices nếu thực sự cần thiết."

Tiếp cận "Monolith First" - Sam Newman



❖ Tác giả & Tài liệu

- ❑ **Sam Newman** – *Building Microservices*, Ấn bản 2
- ❑ https://samnewman.io/books/building_microservices/



❖ Quan điểm chính

- ❑ **Đồng tình với Martin Fowler**: Khuyến nghị **bắt đầu phát triển hệ thống mới** dưới dạng một **khối triển khai đơn lẻ** – tức **monolith**.
- ❑ **Chỉ nên sử dụng microservices** khi bạn **thật sự bị thuyết phục** rằng hệ thống của bạn cần những lợi ích mà microservices mang lại.

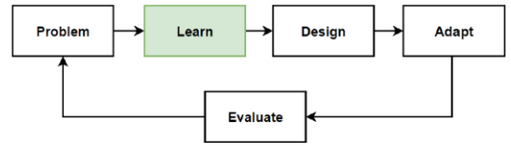
→ Không nên chọn microservices **mặc định** cho mọi dự án.

- ❑ "Monolithic architecture là một lựa chọn hợp lý và chính đáng. Tôi thậm chí còn cho rằng đây là lựa chọn mặc định hợp lý về mặt kiến trúc. Nói cách khác, tôi đang tìm lý do để bị thuyết phục sử dụng microservices, thay vì tìm lý do để không sử dụng chúng."

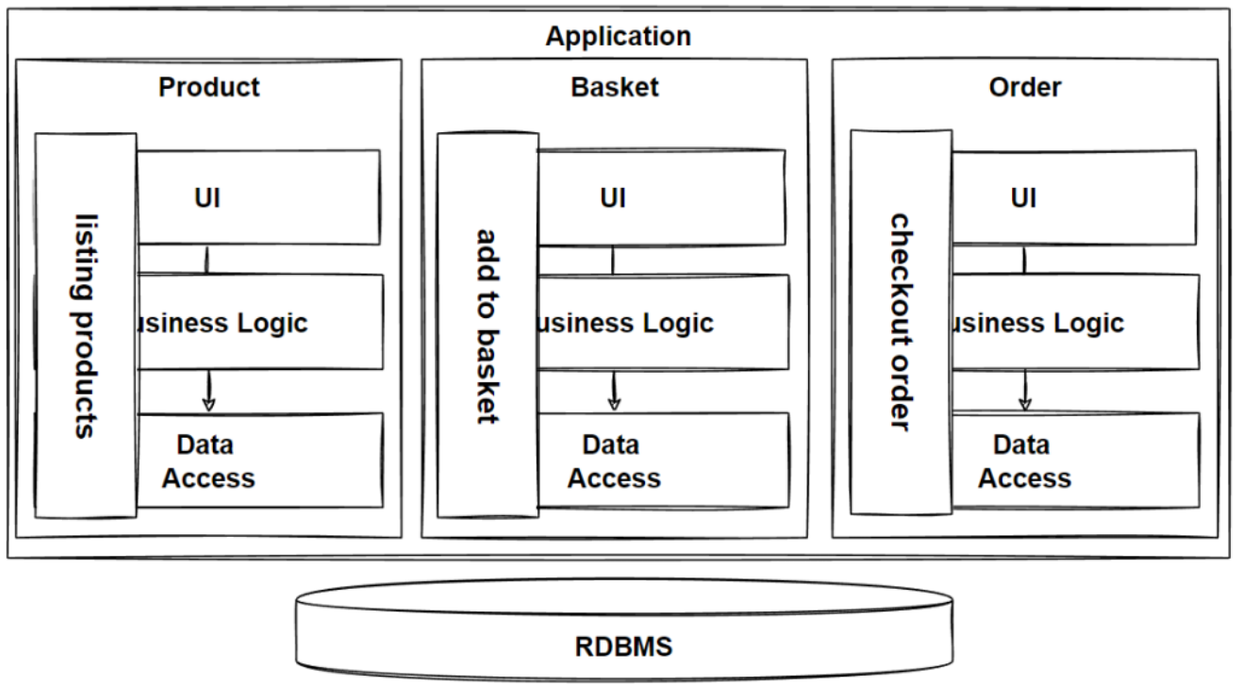
❖ Thông điệp tổng kết

- ❑ Bắt đầu với monolith là lựa chọn hợp lý và hiệu quả. Chỉ nên chia nhỏ thành microservices khi hệ thống đủ phức tạp và bạn hiểu rõ lợi ích cũng như ranh giới chức năng (Bounded Contexts).

Tiếp cận "Monolith First" - Mehmet Ozkaya

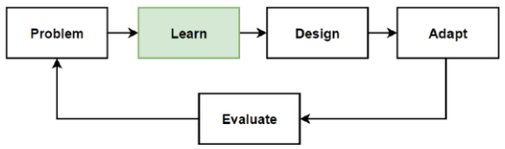


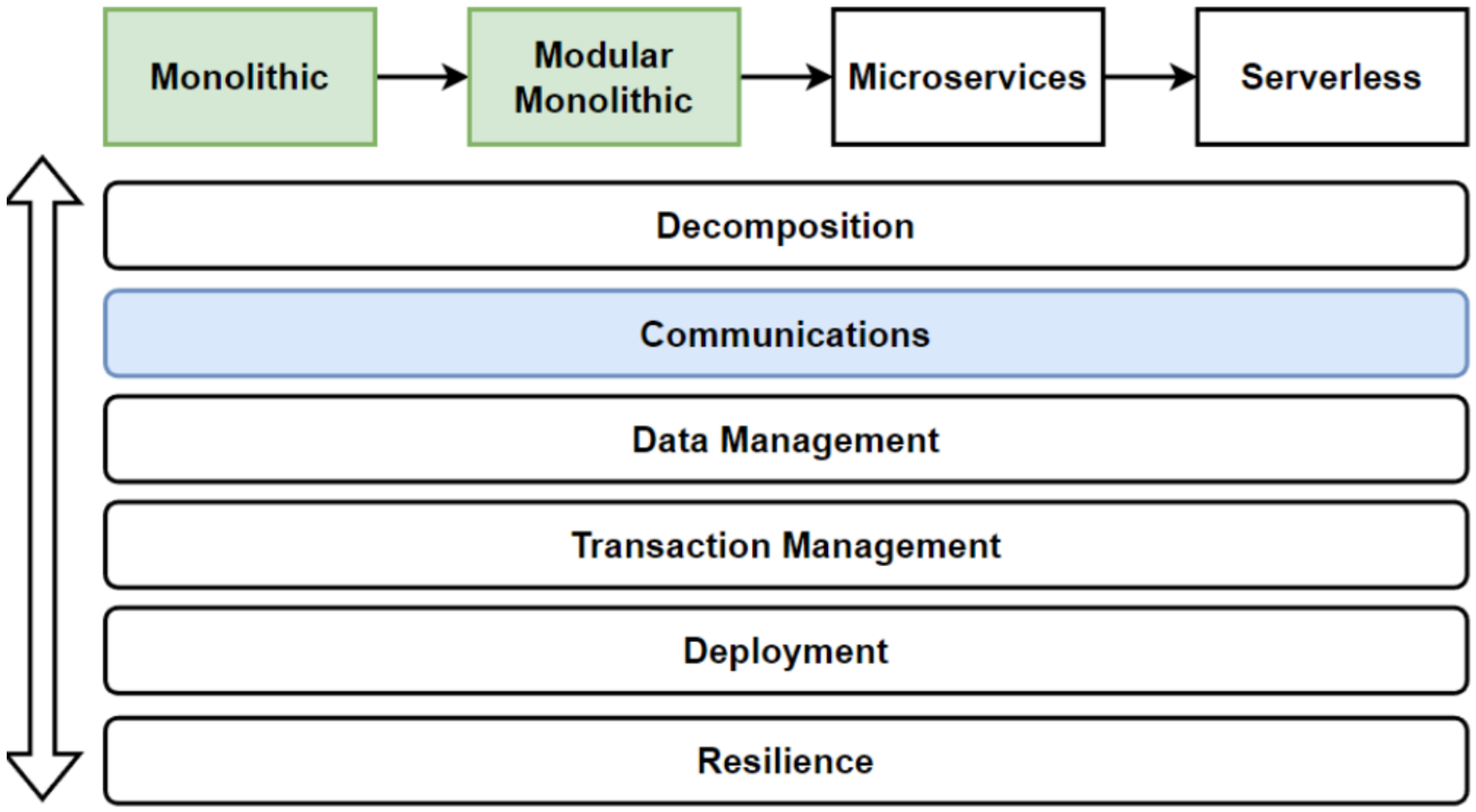
- **Tư duy đúng về Monolith**
  - Kiến trúc nguyên khối không đồng nghĩa với thiết kế kém hay lỗi thời.
  - Ap dụng hướng **Modular Monolithic**, có được nhiều lợi ích tương tự như microservices: ***Tái sử dụng mã nguồn***, ***Dễ dàng tái cấu trúc (refactor)***, **Quản lý phụ thuộc tốt hơn**, ***Tổ chức nhóm phát triển hiệu quả hơn***



- **Vậy có nên luôn bắt đầu với Monolith không?**
  - Câu trả lời là: KHÔNG PHẢI LÚC NÀO CŨNG NÊN.
- Cách ra quyết định giữa Modular Monolithic và Microservices
- **Modular Monolithic**
  - Cần tính nhất quán dữ liệu mạnh (Strong Consistency)
  - Không yêu cầu triển khai độc lập hay mở rộng riêng lẻ
- **Microservices**
  - **Không yêu cầu nhất quán chặt**, chấp nhận nhất quán cuối (Eventual Consistency)
  - Cần mở rộng và triển khai độc lập theo mô-đun

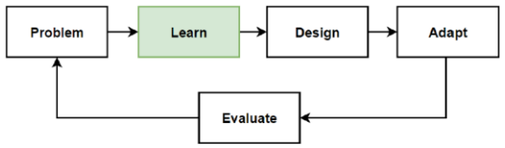
Lát cắt dọc trong kiến trúc nguyên khối: Communications



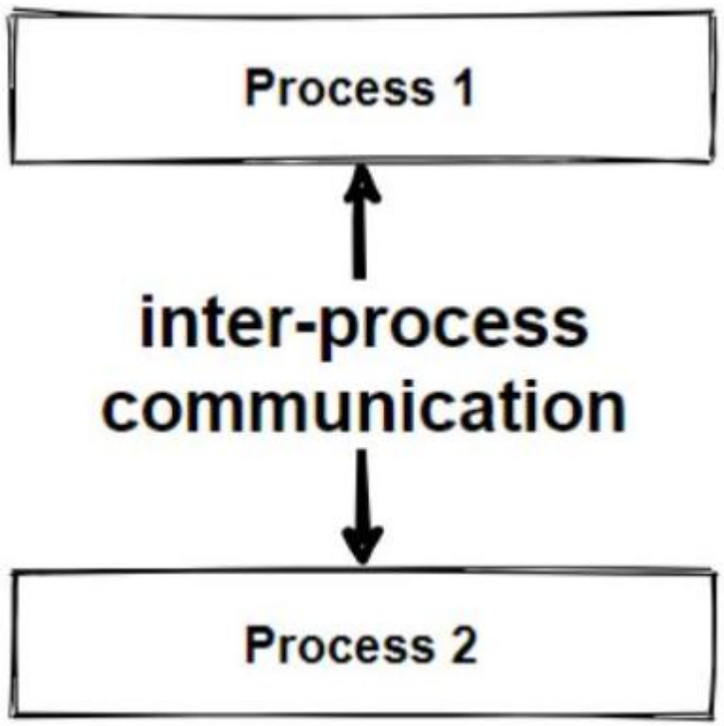


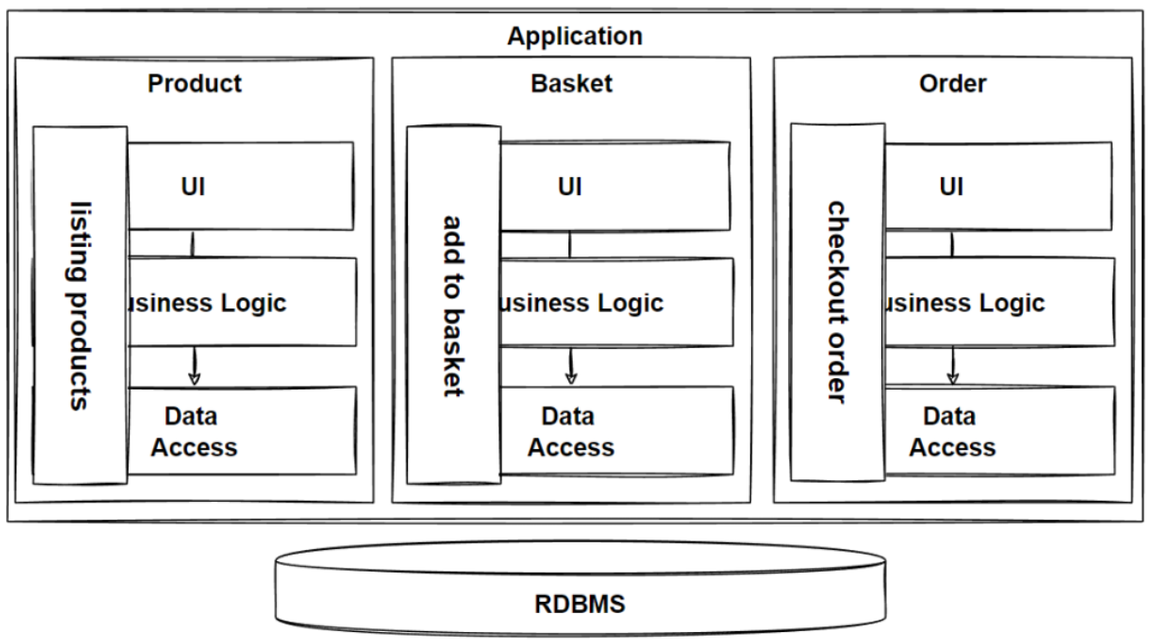
- Các yếu tố cần xem xét trong lộ trình phát triển kiến trúc phần mềm
  - **Decomposition**: Mức độ chia nhỏ hệ thống thành các thành phần, mô-đun hoặc dịch vụ riêng biệt
  - **Communications**: Cách các thành phần/mô-đun giao tiếp với nhau (trong bộ nhớ, HTTP, MQ, v.v.)
  - **Data Management**: Cách quản lý dữ liệu tập trung hay phân tán, tính nhất quán
  - **Transaction Management**: Cách xử lý giao dịch (1 phase hay 2 phase commit, rollback, saga…)
  - **Deployment**: Chiến lược triển khai: đơn lẻ, theo cụm, CI/CD đa module, hay độc lập dịch vụ
  - **Resilience**: Khả năng phục hồi khi một phần hệ thống gặp lỗi hoặc không sẵn sàng

Giao tiếp trong kiến trúc nguyên khối

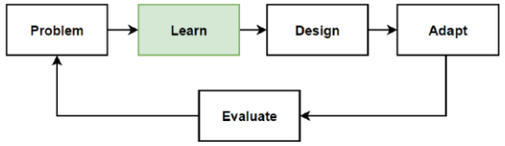


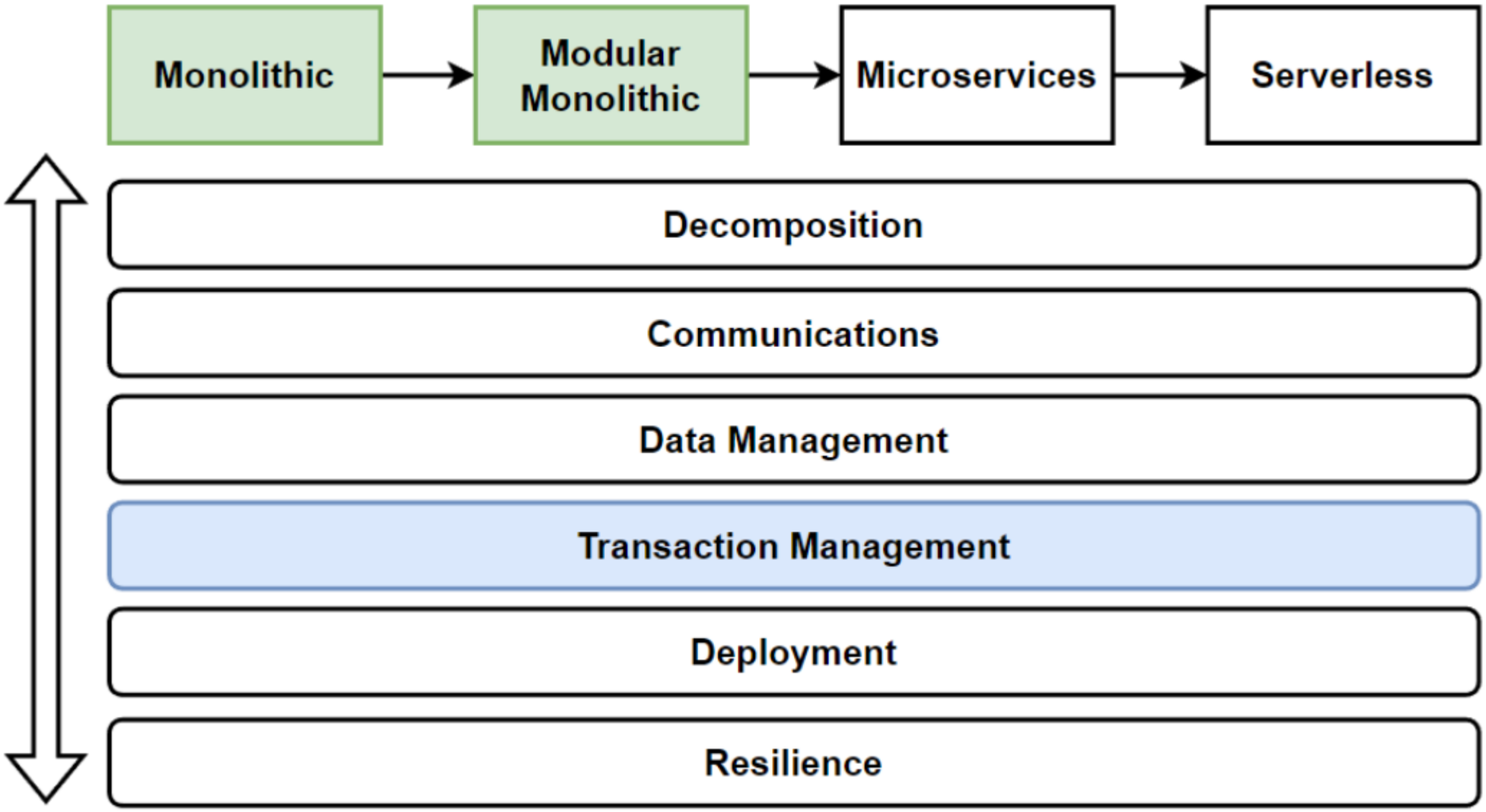
- Cơ chế giao tiếp nội bộ
  - **Ứng dụng monolithic** hoạt động trên **cùng một máy chủ** với tất cả các mô-đun → **Không cần gọi mạng** giữa các phần của hệ thống.
  - Việc giao tiếp giữa các mô-đun là **rất dễ dàng và nhanh chóng**.
- Giao tiếp theo cơ chế gọi hàm nội bộ
  - Giao tiếp trong monolith là dạng **giao tiếp liên tiến trình (Inter-Process Communication)**.
  - Điều này thường được hiểu là: **Gọi hàm trực tiếp trong cùng tiến trình, cùng khối mã**.
  - Hệ điều hành cung cấp cơ chế **cho phép các tiến trình trao đổi thông tin**, nhưng trong monolith, vì tất cả cùng nằm trong một process nên giao tiếp **thường chỉ là gọi phương thức (method call)**.
- Ưu điểm của giao tiếp trong monolith
  - Nhanh chóng (không qua mạng)
  - Dễ debug
  - Không cần xử lý lỗi phân tán
  - Mã gọi hàm rõ ràng, dễ đọc



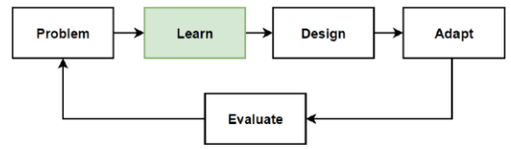


Lát cắt dọc trong kiến trúc nguyên khối: Transaction Management





Quản lý Giao dịch trong kiến trúc nguyên khối



- Đặc điểm nổi bật
  - Giao dịch trong Monolithic **dễ dàng hơn** so với Microservices.
  - Thường sử dụng **một cơ sở dữ liệu duy nhất** cho toàn bộ ứng dụng.
  - Tập trung vào **bối cảnh đơn lẻ (single context)** – phù hợp với kiến trúc nguyên khối.

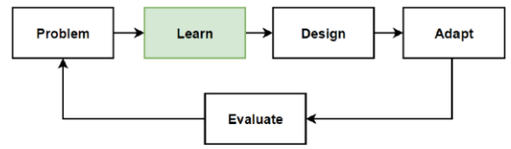
```
function place_order()
    do_payment
    decrease_stock
    send_shipment
    generate_bill
    update_order
```

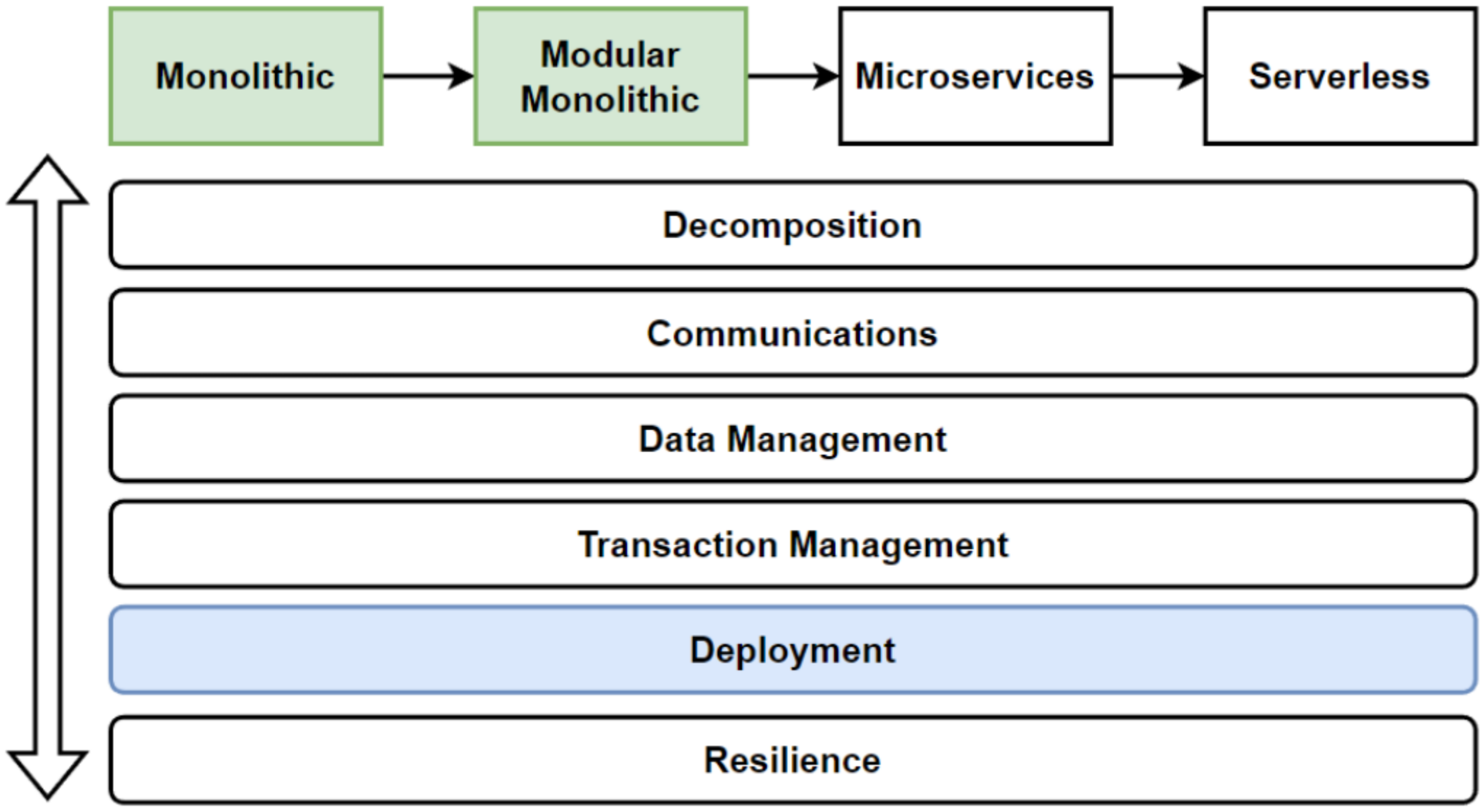
- Cách hoạt động
  - Giao dịch giữ trong **bộ nhớ tạm** cho đến khi **commit**.
  - Nếu có lỗi, **rollback** sẽ xóa toàn bộ thao tác trong vùng giao dịch.
  - Khi **commit ghi xuống CSDL**, giao dịch được coi là **hoàn tất thành công**.

- Ưu điểm
  - **Đơn giản**, dễ triển khai với các framework phổ biến.
  - **Tốc độ nhanh**, ít rủi ro vì không cần xử lý phân tán.
  - **Hỗ trợ đầy đủ commit/rollback** theo cơ chế ACID.

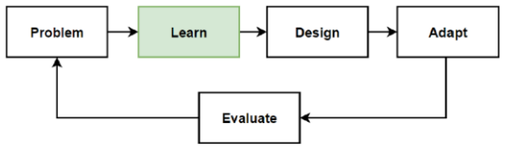
- Cơ chế **ACID** là tập các thuộc tính đảm bảo **giao dịch trong hệ quản trị CSDL** thực hiện **đáng tin cậy và nhất quán**.
  - A – Atomicity (Tính nguyên tử): Giao dịch là một **đơn vị không thể chia nhỏ**. Hoặc toàn bộ các thao tác thành công, hoặc tất cả đều bị hủy (rollback).
  - C – Consistency (Tính nhất quán): Sau mỗi giao dịch, cơ sở dữ liệu phải ở **trạng thái hợp lệ**. Giao dịch không được làm phá vỡ các **ràng buộc toàn vẹn** trong hệ thống.
  - I – Isolation (Tính độc lập): Các giao dịch **diễn ra song song** không được ảnh hưởng lẫn nhau. Kết quả phải **giống như khi chúng được thực hiện tuần tự**.
  - D – Durability (Tính bền vững): Khi giao dịch **đã được commit**, dữ liệu sẽ **được lưu vĩnh viễn**, ngay cả khi hệ thống gặp sự cố.

Lát cắt dọc trong kiến trúc nguyên khối: Depoloyment

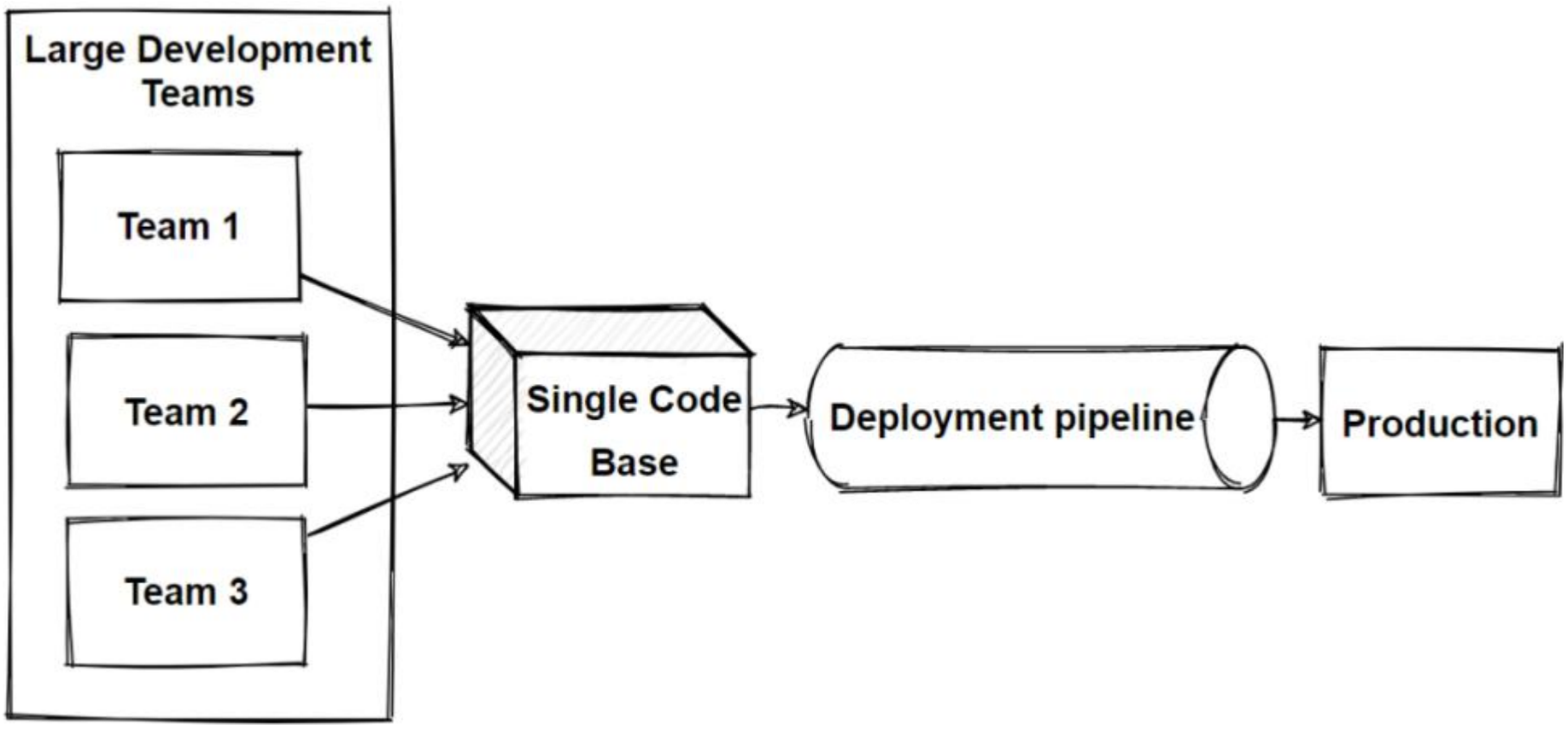




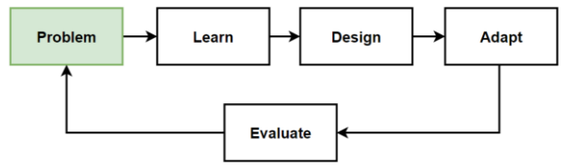
Triển khai trong kiến trúc nguyên mẫu



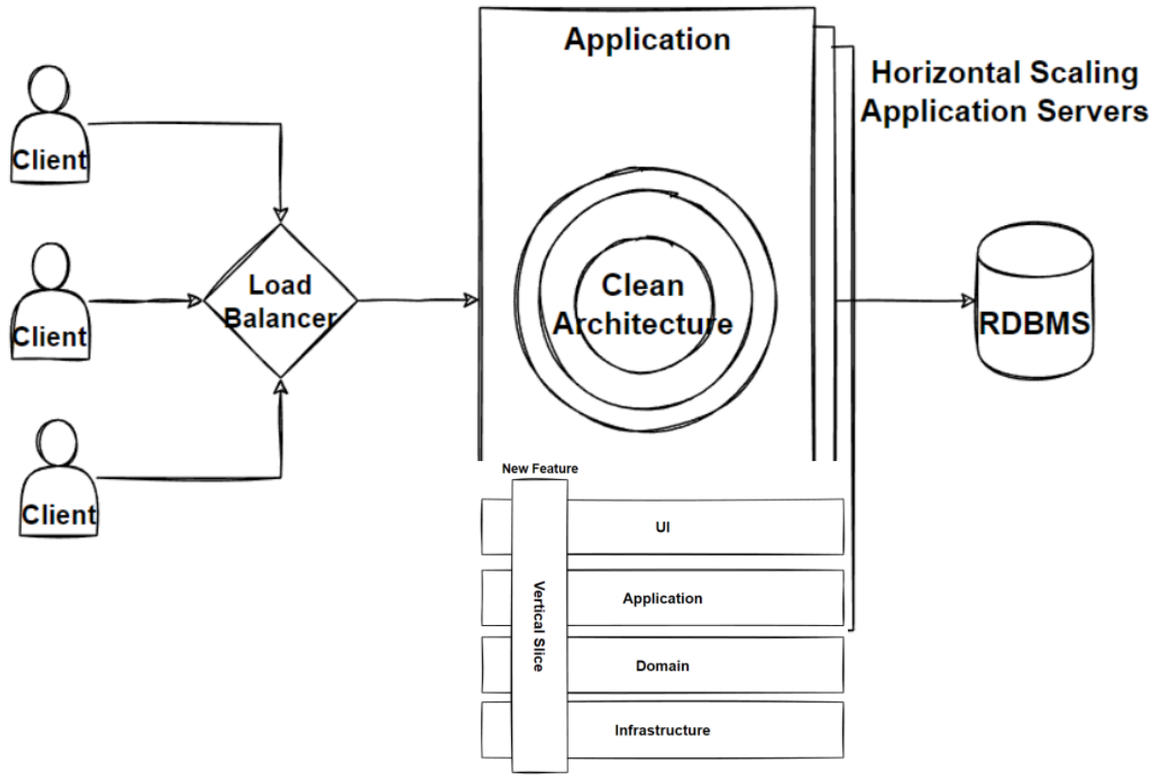
- Vấn đề chính
  - **Một khối mã nguồn (single code base)** khiến việc cập nhật trở nên khó khăn, nhất là với hệ thống lớn.
  - **Thay đổi nhỏ cũng ảnh hưởng toàn hệ thống** → buộc phải triển khai lại toàn bộ ứng dụng.
  - **Một lỗi (bug)** trong bất kỳ mô-đun nào **có thể làm sập cả hệ thống.**
- Hệ quả
  - Dễ xảy ra **xung đột khi nhiều nhóm cùng phát triển**.
  - Quy trình triển khai **phức tạp, rủi ro cao, thiếu linh hoạt**.



Vấn đề: Tính linh hoạt khi phát triển tính năng mới, nhóm Agile tách biệt



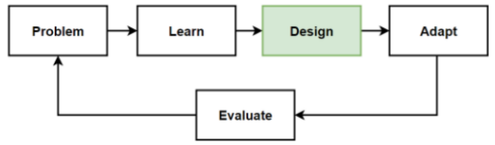
- **Các vấn đề**:
  - Doanh nghiệp thương mại điện tử đang phát triển.
  - Các nhóm kinh doanh được tách biệt theo phòng ban: Sản phẩm, Bán hàng, Thanh toán...
  - Tất cả các nhóm đều muốn thêm tính năng mới để cạnh tranh trên thị trường.
  - Mã nguồn hiện tại không cho phép quản lý điều đó.
  - Việc chuyển đổi ngữ cảnh và mô hình cắt lớp dọc (Vertical Slices) gây ra vấn đề trong kiến trúc Clean.

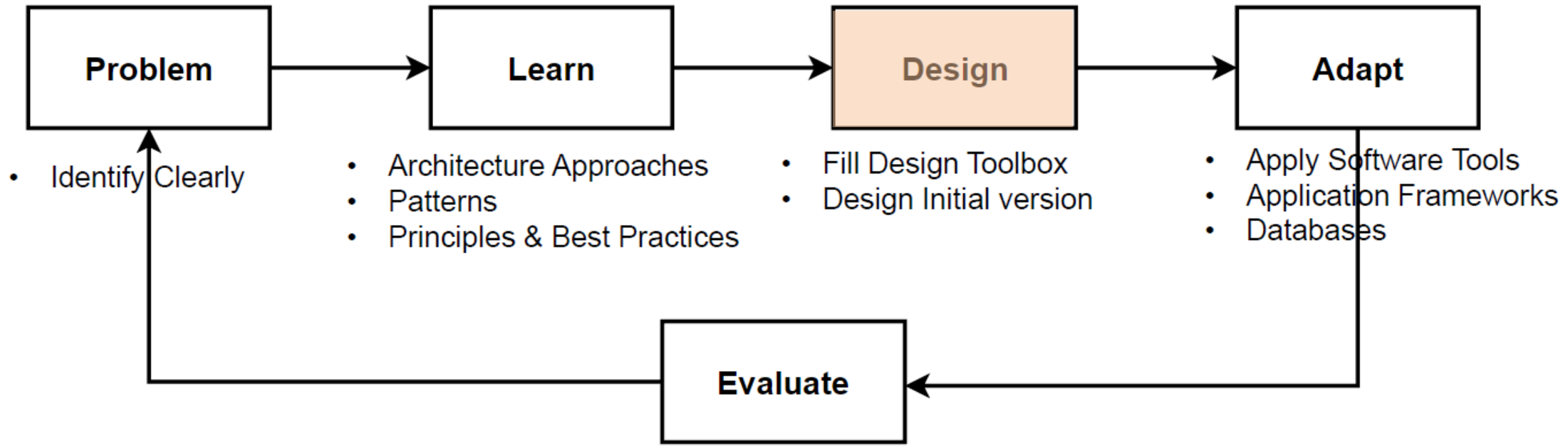


Vấn đề phân cắt dọc trong kiến trúc Clean

- **Giải pháp**:
  - **Áp dụng kiến trúc nguyên khối theo mô-đun** (Modular Monolithic Architecture)

Quy trình Thiết kế Kiến trúc Phần Mềm

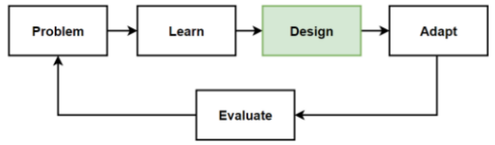




- What's wrong with this architecture ?
- how we can improve this current design ?

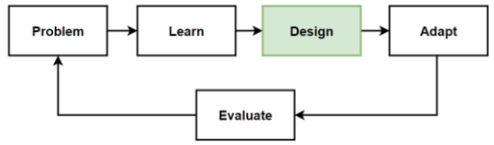
Quy trình Thiết kế Kiến trúc Phần Mềm

Trước khi thiết kế – Chúng ta có gì trong hộp công cụ thiết kế?



Kiến trúc	Mẫu & Nguyên tắc	Yêu cầu phi chức năng	Yêu cầu chức năng
• Kiến trúc nguyên khối • Kiến trúc phân lớp • Kiến trúc Clean • Kiến trúc nguyên mẫu module	• DRY (Không lặp lại chính mình) • KISS (Giữ đơn giản, đừng phức tạp) • YAGNI (Bạn sẽ không cần nó) • Seperation of Concerns (SoC) • SOLID • Quy tắc phụ thuộc • **Ưu tiên tiếp cận nguyên mẫu**	• Độ sẵn sàng • Số lượng người dùng đồng thời nhỏ • Khả năng bảo trì • Tính linh hoạt • Dễ kiểm thử • Tính khả mở • **Độ tin cậy** • **Tính tái sử dụng**	• Liệt kê sản phẩm • Lọc sản phẩm theo thương hiệu và danh mục • Thêm sản phẩm vào giỏ hàng • Áp dụng mã giảm giá • Thanh toán giỏ hàng và tạo đơn hàng • Xem danh sách đơn hàng cũ và lịch sử các mặt hàng đã đặt

Thiết kế Kiến trúc nguyên khối module

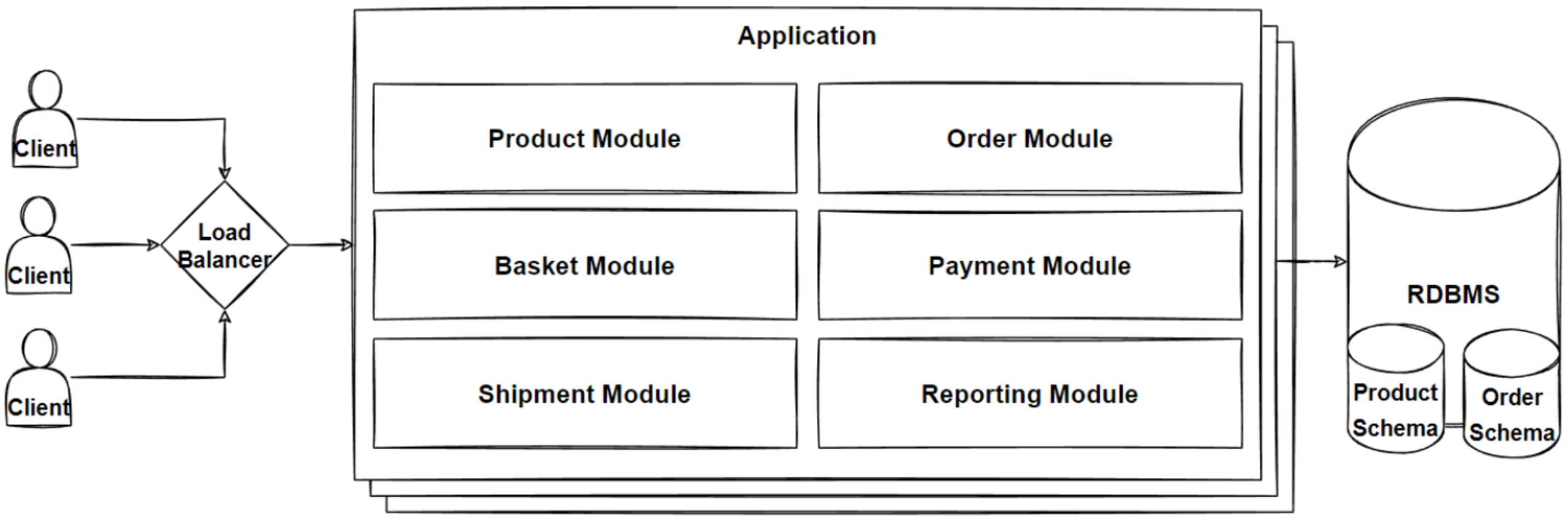


❖ Cấu trúc hệ thống

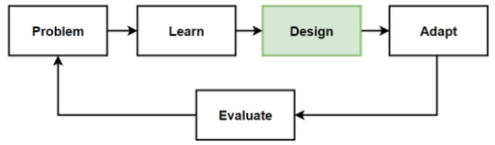
- ❑ Ứng dụng gồm nhiều mô-đun nghiệp vụ độc lập
- ❑ **Product**, **Order**
- ❑ **Basket**, **Payment**
- ❑ **Shipment**, **Reporting**
- ❑ Tất cả mô-đun chạy trong **một khối ứng dụng duy nhất**, sử dụng **Load Balancer** để chia tải cho nhiều người dùng.
- ❑ Dữ liệu được quản lý tập trung qua **RDBMS**, nhưng có thể tách riêng theo **schema từng mô-đun** (ví dụ: Product Schema, Order Schema).

❖ Lợi ích chính

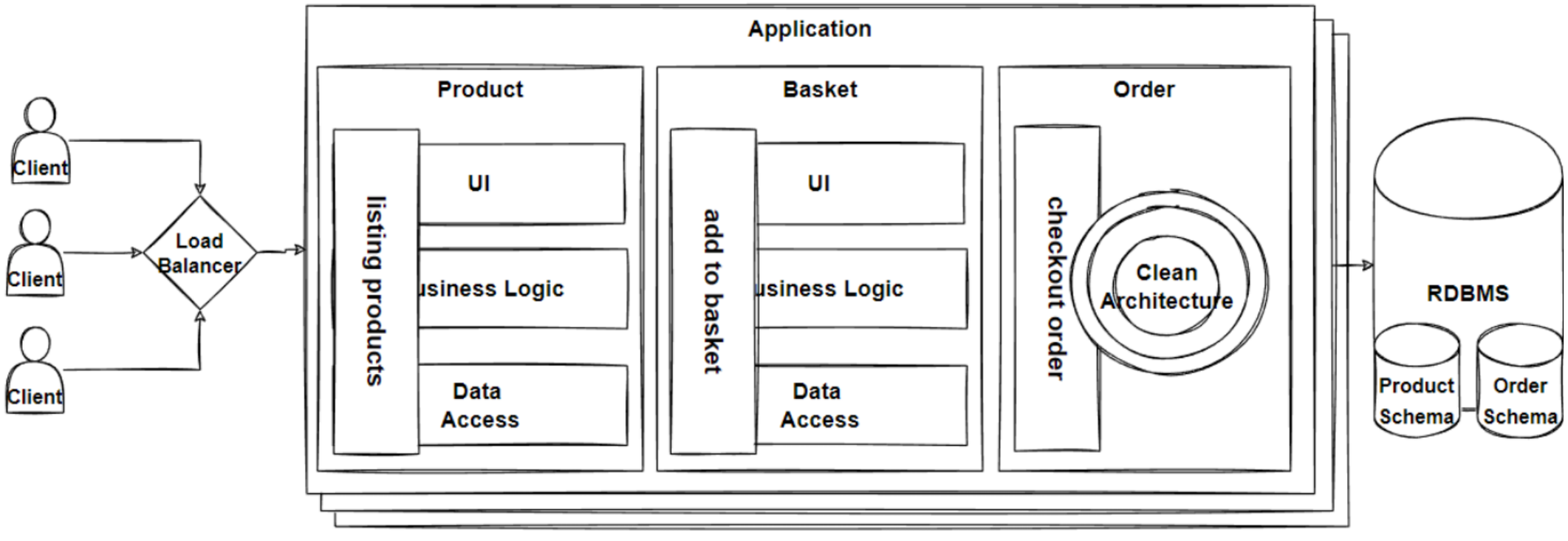
- ❑ Dễ tổ chức nhóm phát triển theo từng chức năng.
- ❑ Tăng khả năng bảo trì, mở rộng dần theo mô-đun.
- ❑ Chuẩn bị tốt cho việc chuyển hóa sang microservices khi cần.



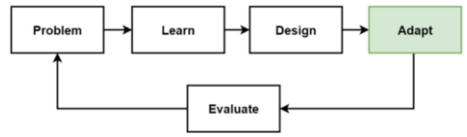
Thiết kế bên trong của kiến trúc nguyên khối

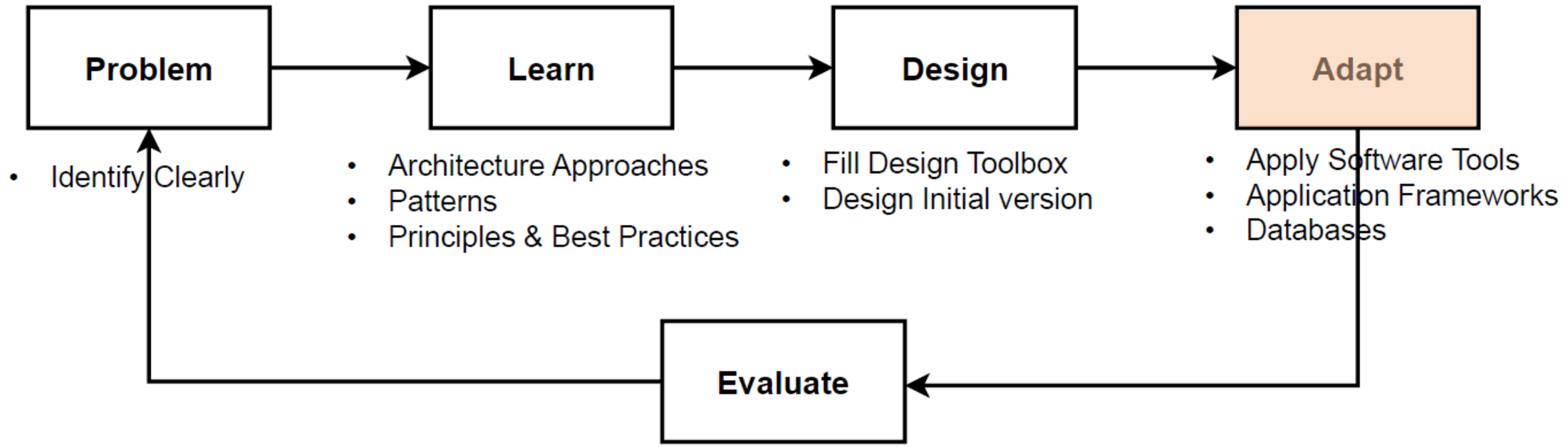


- **Cấu trúc module**
  - Ứng dụng chia thành các mô-đun theo chức năng
  - **Product** – Liệt kê sản phẩm, **Basket** – Thêm vào giỏ hang, **Order** – Thanh toán đơn hàng (áp dụng **Clean Architecture**)
- Luồng xử lý
  - Người dùng truy cập qua **Load Balancer** → chuyển đến ứng dụng chính.
  - Mỗi mô-đun gồm: UI, Business Logic, Data Acces
- Quản lý dữ liệu
  - Dùng **RDBMS** tập trung
  - Schema riêng theo mô-đun: **Product Schema**, **Order Schema**
- **Lợi ích**
  - Dễ mở rộng, phân chia nhóm theo chức năng
  - Áp dụng Clean Architecture cho mô-đun quan trọng
  - Tăng khả năng bảo trì và chuẩn bị tốt cho microservices



Quy trình Thiết kế Kiến trúc Phần Mềm

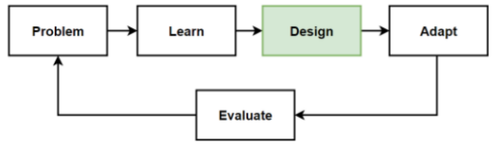




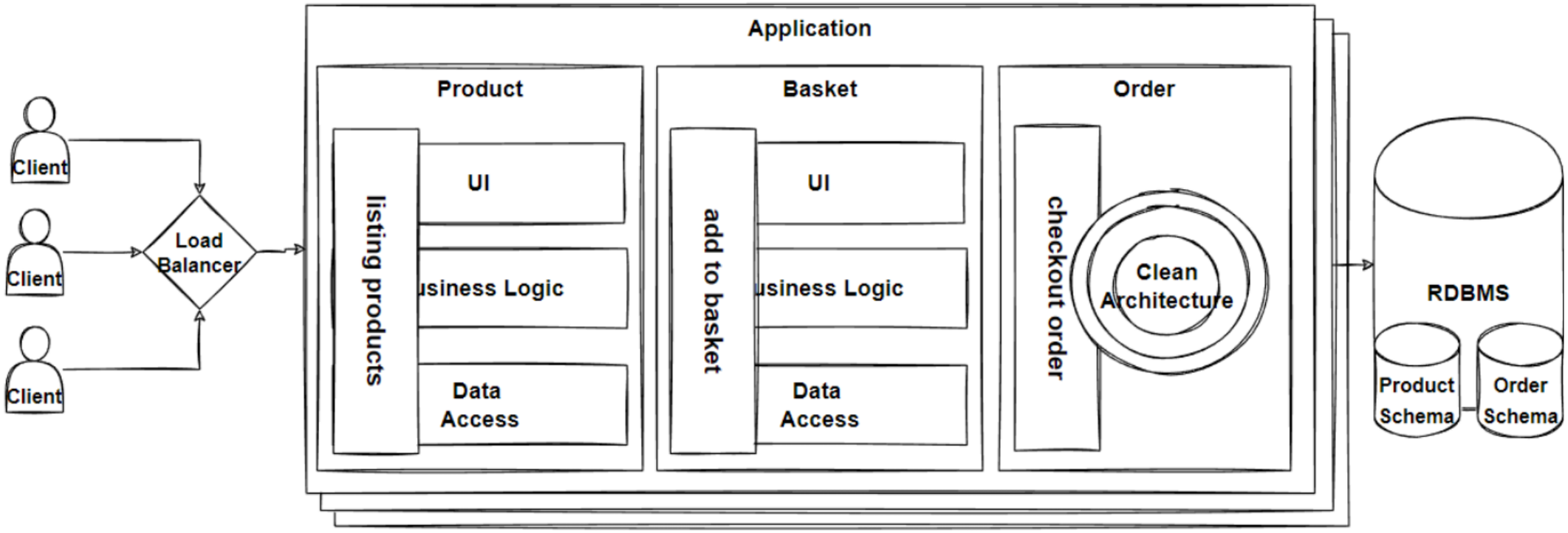
- What's wrong with this architecture ?
- how we can improve this current design ?

Quy trình Thiết kế Kiến trúc Phần Mềm

Minh họa: Đánh giá mã nguồn của Kiến trúc nguyên khố module trong .NET

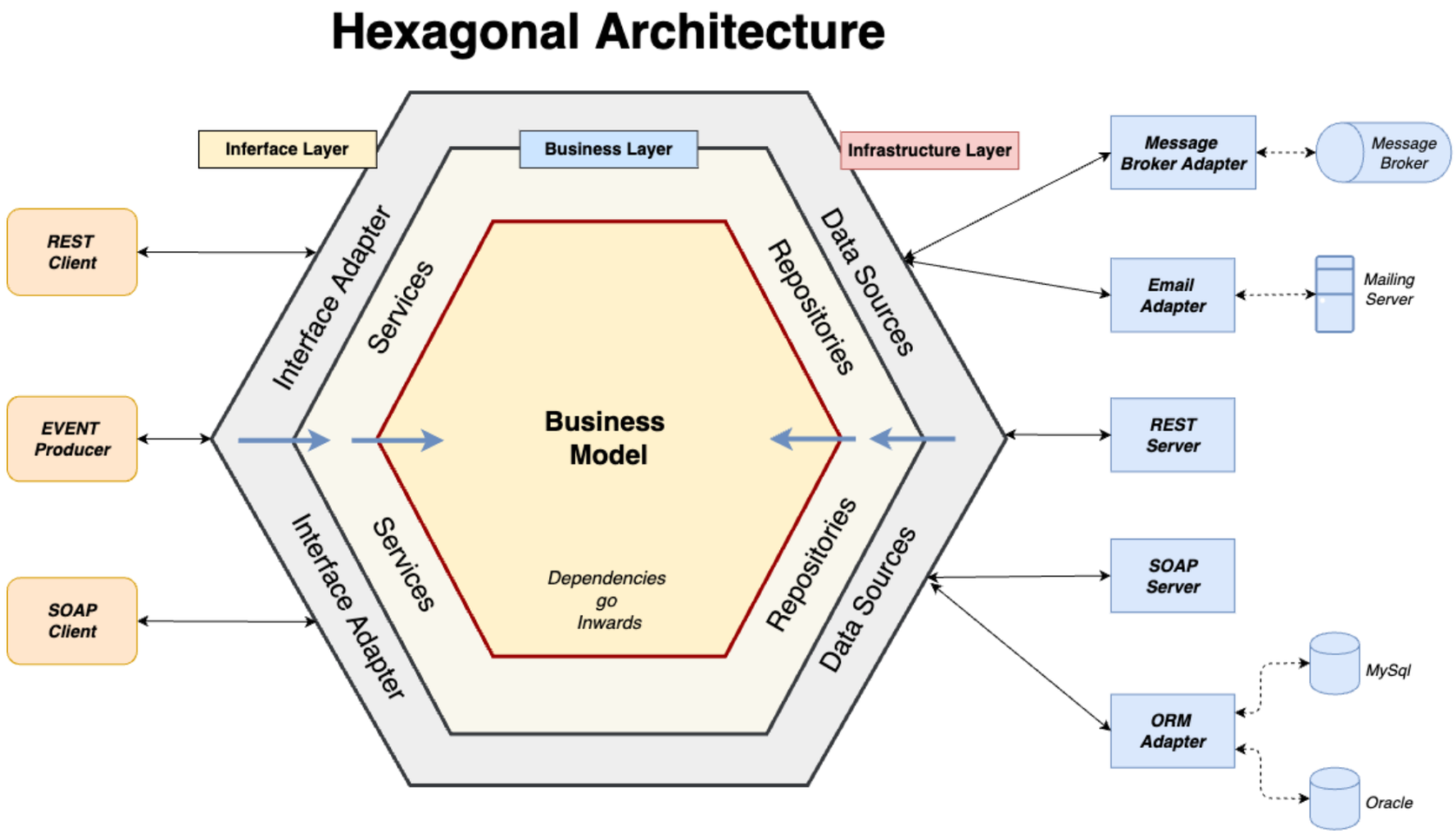


- **DEMO: Đánh giá mã nguồn của kiến trúc nguyên khối module trong .NET**
- Ứng dụng thực tế
  - https://github.com/kgrzybek/modular-monolith-with-ddd
  - https://github.com/meysamhadeli/booking-modular-monolith
  - https://github.com/arawn/building-modular-monoliths-using-spring



Triển khai kiến trúc nguyên khối module trong .NET

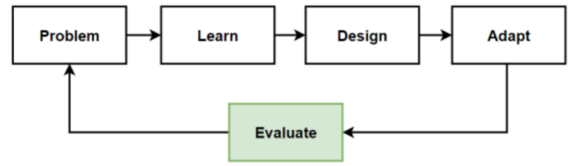
Minh họa: Hexagonal vs Clean Archtecture

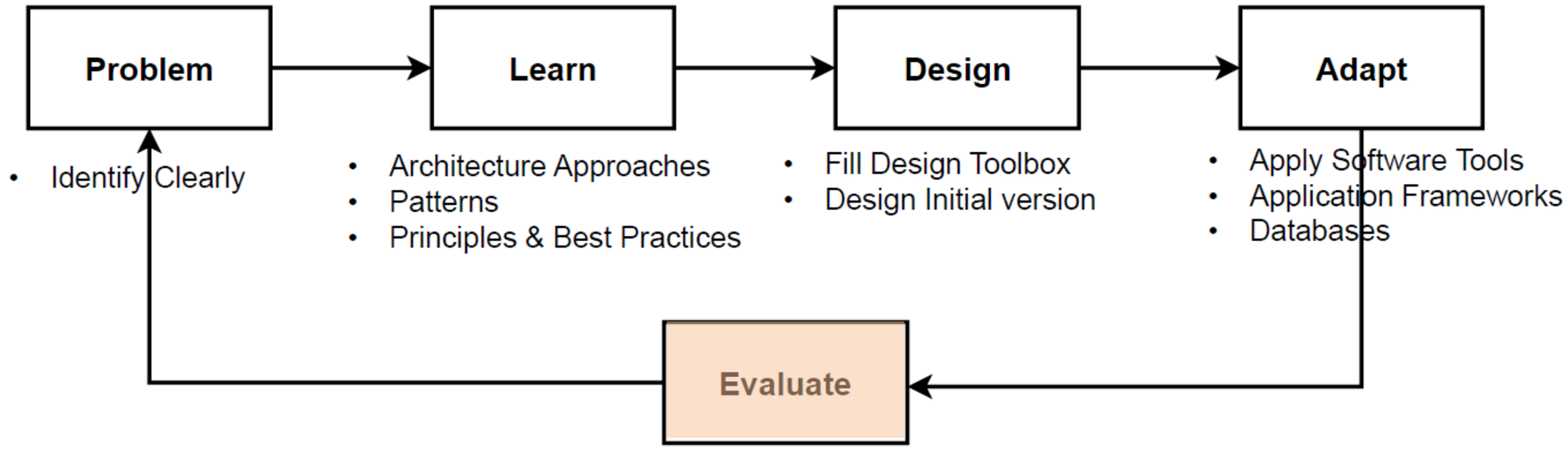


https://www.linkedin.com/pulse/whats-hexagonal-architecture-luis-soares-m-sc-/

https://github.com/AlicanAkkus/Modular-Architecture-Hexagonal-Demo-Project

Quy trình Thiết kế Kiến trúc Phần Mềm

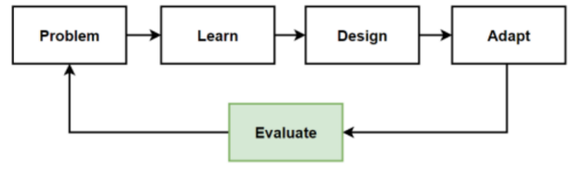




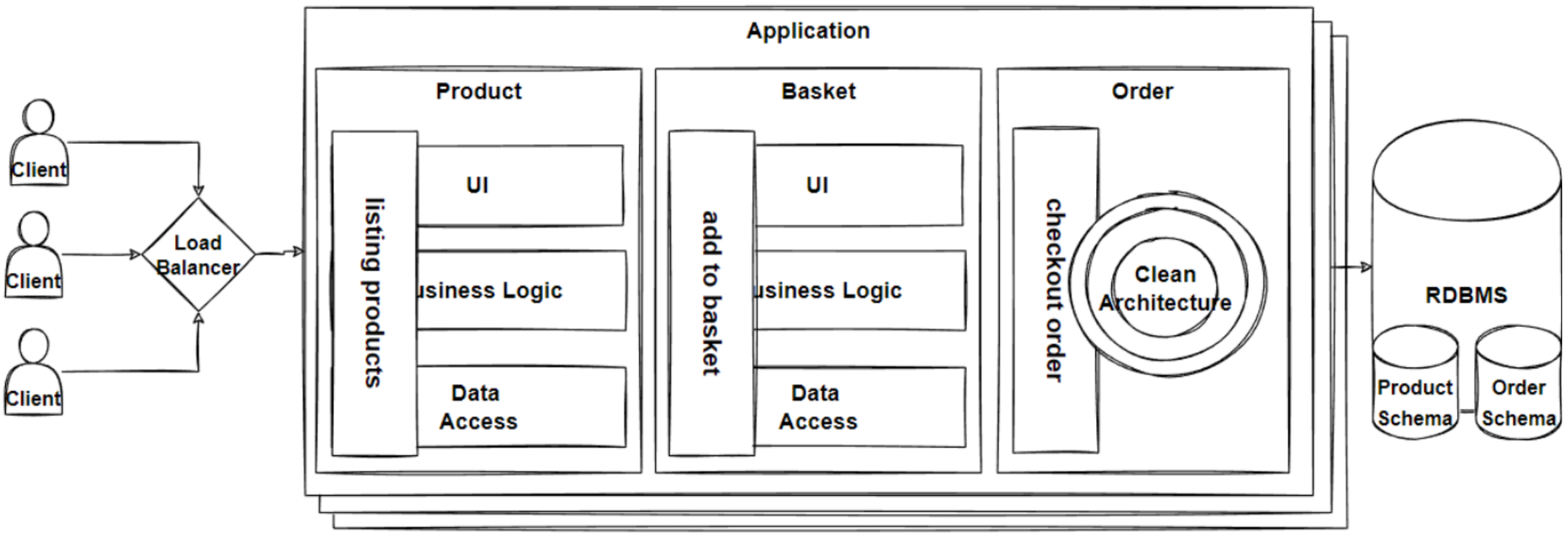
- What's wrong with this architecture ?
- how we can improve this current design ?

Quy trình Thiết kế Kiến trúc Phần Mềm

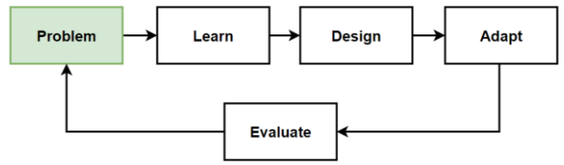
Đánh giá: Độ phức tạp của lớp giao diện người dùng (UI)

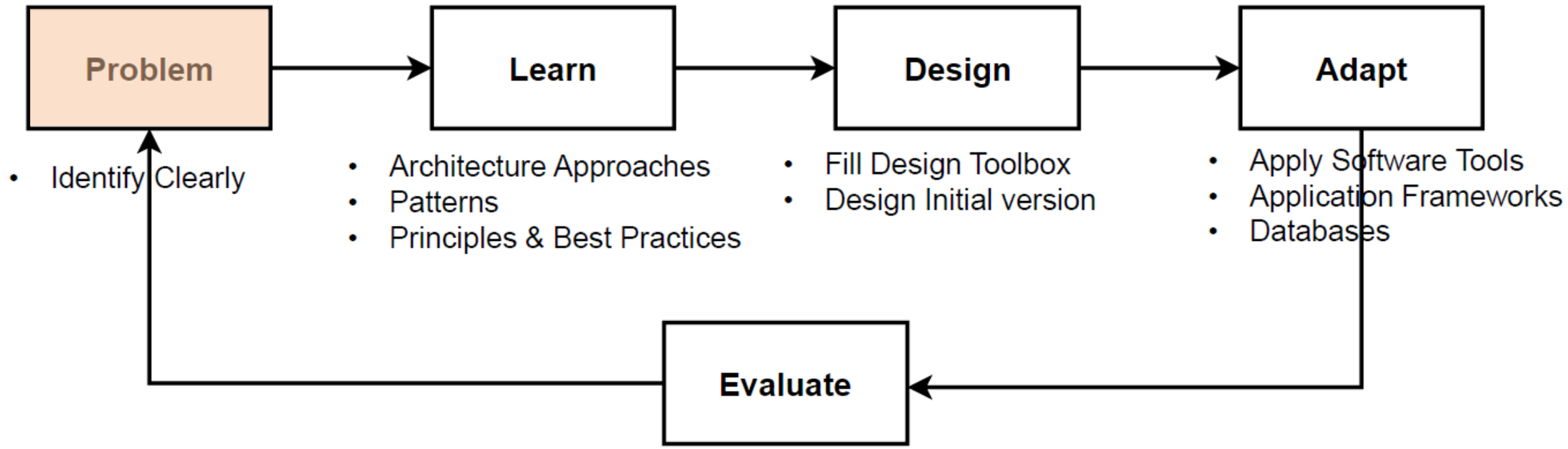


- Lợi ích
  - Phát triển, gỡ lỗi và triển khai dễ dàng
  - Mỗi mô-đun giao diện được đóng gói riêng theo **vertical slice**
- Hạn chế
  - Giao diện được xử lý tập trung trong một ứng dụng monolith lớn
  - Càng thêm tính năng mới, **UI càng trở nên phức tạp, khó kiểm soát**
  - Mỗi mô-đun đều sinh giao diện phía server → dễ xung đột
  - Các yêu cầu nhỏ của UI cũng đòi hỏi **triển khai lại toàn bộ ứng dụng**
  - **Các nhóm nghiệp vụ liên tục yêu cầu tùy biến UI**, gây áp lực lớn cho backend
- Giao diện trong monolith ban đầu dễ phát triển, nhưng về lâu dài **trở thành điểm nghẽn về tính linh hoạt và khả năng mở rộng**.



Quy trình Thiết kế Kiến trúc Phần Mềm

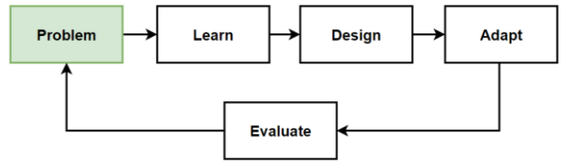




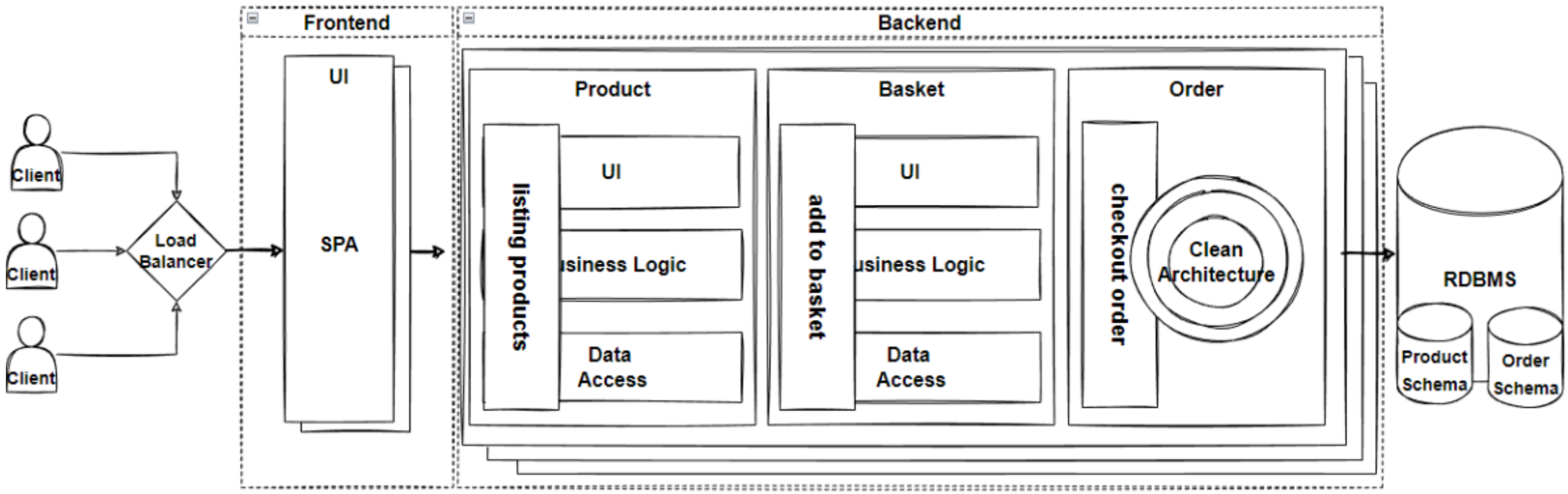
- What's wrong with this architecture ?
- how we can improve this current design ?

Quy trình Thiết kế Kiến trúc Phần Mềm

Vấn đề: Cải thiện trải nghiệm khách hàng với SPA



- Vấn đề hiện tại
  - Doanh nghiệp TMĐT đang mở rộng
  - Cần **trải nghiệm tốt hơn** với giao diện tách biệt và hỗ trợ đa kênh (Omnichannel)
  - Giao diện hiện tại không phản hồi tốt (non-responsive)
  - Kỳ vọng từ khách hàng đa kênh ngày càng cao
- Giải pháp đề xuất
  - Tách riêng phần giao diện bằng **SPA (Single Page Application)**
  - Phân tách **Frontend – Backend** rõ rang
  - Hướng đến **Kiến trúc Headless** (không phụ thuộc UI tích hợp sẵn trong backend)
- Để đáp ứng nhu cầu người dùng hiện đại, hệ thống cần chuyển từ UI tích hợp sang giao diện SPA độc lập, dễ mở rộng và hỗ trợ đa nền tảng (web, mobile, kiosk...).



CÁM ƠN ĐÃ CHÚ Ý LANG NGHE!