

**BỘ CÔNG THƯƠNG**

**TRƯỜNG ĐẠI HỌC CÔNG NGHIỆP TP. HỒ CHÍ MINH**

**KHOA CÔNG NGHỆ THÔNG TIN**

**----🙣🕮🙡----**



**BÁO CÁO**

**Môn: Kiến trúc và thiết kế phần mềm**

**ĐỀ TÀI**

**HỆ THỐNG QUẢN LÝ LỚP HỌC**

**TÍN CHỈ**

***Giảng viên hướng dẫn* : GV. Võ Văn Hải**

***Sinh viên thực hiện* : Nguyễn Văn Vũ - 20002865**

**Hà Anh Thảo - 20001575**

**Nguyễn Văn Hiếu - 20000175**

***TP.HCM, ngày 19 tháng 05 năm 2024***

**Mục lục**

[1. Software architecture fundamentals 4](#_Toc167055544)

[**1.1.** **Kiến trúc phần mềm** 4](#_Toc167055545)

[**1.2.** **Xác định kiến trúc phần mềm** 4](#_Toc167055546)

[**1.3.** **Coupling** 4](#_Toc167055547)

[**1.3.1.** **Content coupling** 5](#_Toc167055548)

[**1.3.2.** **Common coupling** 6](#_Toc167055549)

[**1.3.3.** **Control coupling** 6](#_Toc167055550)

[**1.3.4.** **Stamp coupling** 6](#_Toc167055551)

[**1.3.5.** **Data coupling** 6](#_Toc167055552)

[**1.4.** **Cohension** 6](#_Toc167055553)

[**1.4.1.** **Coincidental** 7](#_Toc167055554)

[**1.4.2.** **Logical cohesion** 7](#_Toc167055555)

[**1.4.3.** **Temporal cohesion** 8](#_Toc167055556)

[**1.4.4.** **Procedural cohesion** 8](#_Toc167055557)

[**1.5.** **The Lack of Cohesion in Methods (LCOM)** 8](#_Toc167055558)

[**1.6.** **Distance from the Main Sequence** 9](#_Toc167055559)

[**1.7.** **Connascence** 10](#_Toc167055560)

[**1.7.1.** **Connascense of Name (CoN)** 10](#_Toc167055561)

[**1.7.2.** **Connascence of Type (CoT)** 11](#_Toc167055562)

[**1.7.3.** **Connascence of Meaning (CoM) or Connascence of Convention (CoC)** 12](#_Toc167055563)

[**1.7.5.** **Connascence of Algorithm (CoA)** 13](#_Toc167055564)

[2. Common software architecture styles 20](#_Toc167055565)

[**2.1.** **Layered Architecture Style** 20](#_Toc167055566)

[**2.1.1.** **Ưu điểm** 20](#_Toc167055567)

[**2.2.** **Pipeline Architecture Style** 21](#_Toc167055568)

[**2.2.1.** **Ưu điểm** 21](#_Toc167055569)

[**2.2.2.** **Nhược điểm** 22](#_Toc167055570)

[**2.3.** **Microkernel Architecture Style** 23](#_Toc167055571)

[**2.3.1.** **Ưu điểm** 23](#_Toc167055572)

[**2.3.2.** **Nhược điểm** 23](#_Toc167055573)

[**2.4.** **Service-Based Architecture Style** 25](#_Toc167055574)

[**2.4.1.** **Ưu điểm** 25](#_Toc167055575)

[**2.4.2.** **Nhược điểm** 25](#_Toc167055576)

[**2.5.** **Service-Oriented Architecture Style** 25](#_Toc167055577)

[**2.5.1.** **Ưu điểm** 25](#_Toc167055578)

[**2.6.** **Event-Driven Architecture Style** 28](#_Toc167055579)

[**2.6.1.** **Ưu điểm** 28](#_Toc167055580)

[**2.6.2.** **Nhược điểm** 28](#_Toc167055581)

[**2.7.** **Space-Based Architecture Style** 29](#_Toc167055582)

[**2.7.1.** **Ưu điểm** 29](#_Toc167055583)

[**2.7.2.** **Nhược điểm** 29](#_Toc167055584)

[**2.8.** **Orchestration-Driven Service-Oriented Architecture** 30](#_Toc167055585)

[**2.8.1.** **Ưu điểm** 30](#_Toc167055586)

[**2.9.** **Microservices Architecture** 32](#_Toc167055587)

[**2.9.1.** **Ưu điểm** 32](#_Toc167055588)

[3. Select appropriate software architecture style 34](#_Toc167055589)

[**3.0** **Lựa chọn** 34](#_Toc167055590)

[- Chọn kiến trúc: Event Microservice 34](#_Toc167055591)

[**3.1.** **Vì sao chọn?** 34](#_Toc167055592)

[**3.1.1.** **Phù hợp với tính phức tạp của hệ thống** 34](#_Toc167055593)

[**3.1.2.** **Khả năng mở rộng linh hoạt** 34](#_Toc167055594)

[**3.1.5.** **Tăng cường khả năng kiểm thử và triển khai liên tục** 35](#_Toc167055595)

[**3.2.** **Được gì?** 35](#_Toc167055596)

[**3.3.** **Đánh đổi gì?** 36](#_Toc167055597)

[4. Implement a chosen architecture style 37](#_Toc167055598)

[5. Modelling and presenting a software architecture 37](#_Toc167055599)

[**5.1** **Use case** 38](#_Toc167055600)

[**5.2 UML: package diagram, deployment diagram, sequence diagram, component diagram** 38](#_Toc167055601)

[**5.2.1 Package diagram** 38](#_Toc167055602)

[**5.3 Deployment diagram** 40](#_Toc167055603)

1. **Software architecture fundamentals**
   1. **Kiến trúc phần mềm**

Kiến trúc phần mềm đề cập đến cấu trúc cơ bản của một hệ thống phần mềm và quy tắc của việc tạo ra những cấu trúc và hệ thống như vậy. Mỗi cấu trúc bao gồm sự sắp xếp của các yếu tố phần mềm, mối quan hệ giữa các yếu tố, và tính chất của các yếu tố đó.

Cho tới giờ vẫn chưa có một định nghĩa phản ánh đầy đủ nhất về kiến trúc phần mềm bởi hai lý do:

* Sự phát triển nhanh chóng của công nghệ, môi trường phát triển, các loại mô hình kiến trúc phần mềm khác nhau khiến cho các định nghĩa trước đó nhanh chóng trở nên lỗi thời.

- Sự khác biệt về quan điểm giữa các trường phái. Ví dụ: một số software architect coi kiến trúc phần mềm là bản thiết kế chi tiết của hệ thống, trong khi những người khác định nghĩa nó là lộ trình phát triển hệ thống.

* 1. **Xác định kiến trúc phần mềm**

- Kiến trúc phần mềm bao gồm cấu trúc của hệ thống, kết hợp với các đặc điểm kiến trúc mà hệ thống phải hỗ trợ, các quyết định về kiến trúc và cuối cùng là các nguyên tắc thiết kế.

- Cấu trúc đề cập đến kiểu kiến trúc được sử dụng trong hệ thống, chẳng hạn như microservice, layer hoặc microkernel

- Các đặc điểm kiến trúc xác định tiêu chí thành công của hệ thống

- Các quyết định về kiến trúc xác định các quy tắc về cách xây dựng một hệ thống

- Các nguyên tắc thiết kế là những hướng dẫn để xây dựng hệ thống

* 1. **Coupling**

Sự nối kết (coupling) là thước đo chất lượng các mức độ mà một lớp được nối kết với lớp khác. Khi các lớp (và các thành phần) trở nên phụ thuộc lẫn nhau hơn, sự nối kết sẽ gia tăng. Một mục tiêu quan trọng trong thiết kế cấp thành phần là giữ cho sự nối kết càng thấp càng tốt. Ở giai đoạn thiết kế, sự nối kết được dùng để đo mức độ liên kết giữa các đối tượng hoặc giữa các thành phần phần mềm. Đây là một mối kết hợp nhị phầnvà là một khái niệm quan trọng khi đánh giá một thiết kế bởi vì chúng giúp tatập trung đúng vào vấn đề quan trọng của thiết kế. Ví dụ, một thành phần trong hệ thống thay đổi sẽ có một tác động tối thiểu đến các thành phần khác. Sự nối kết trong các đối tượng càng mạnh thì hệ thống càng phức tạp. Lớp nối kết có thể thể hiện chính nó trong nhiều cách khác nhau.

- Afferent coupling: đo số lượng kết nối đến một code artifact. Giá trị này càng cao thì cho thấy độ ổn định của component càng cao, tốt nhất nằm trong khoảng 0 đến 500.

- Efferent coupling: đo số lượng kết nối từ một code artifact khác. Giá trị này phải nằm trong khoảng từ 0 đến 20, nếu cao hơn sẽ gây ra vấn đề trong quá trình phát triển và bảo trì.

- Instability: là tỷ lệ của Efferent coupling với tổng của Afferent coupling và Efferent coupling. Độ đo độ không ổn định xác định tính biến động của cơ sở mã. Một cơ sở mã có mức độ ổn định cao dễ bị phá vỡ hơn khi bị thay đổi do tính liên kết cao. Độ đo độ không ổn định nên nằm trong khoảng từ 0 đến 0.3 hoặc từ 0.7 đến 1. Các component nên tránh trường hợp có độ ổn định trung bình

- Abstractness là tỷ lệ của abstract artifacts so với concrete artifacts. Nó thể hiện độ đo về tính trừu tượng so với việc hiện thực. Một module với tính trừu tượng cao nghĩa là module đó chứa một số lượng đáng kể lớp trừu tượng và interface, ngược lại, nếu module có tính trừu tượng thấp thì khi triển khai cần phải cụ thể hơn với ít lớp trừu tượng và interface hơn. Mục tiêu của tính trừu tượng là đạt được sự cân bằng dựa trên nguyên tắc thiết kế

A diagram of a couple of steps

Description automatically generated

Các mức độ vi phạm coupling và tính nghiêm trọng của nó tăng dần từ dưới lên với high coupling bao gồm content, common và control, loose coupling bao gồm stemp, data và uncoupled.

* + 1. **Content coupling**

Đây là vi phạm couling ở mức cao nhất được định nghĩa là khi mà một component có thể truy cập trực tiếp vào hoạt động bên trong của một component khác ví dụ như trực tiếp truy xuất data, thay đổi data của component. Ví dụ 2 class A và B trong đó một đối tượng class A có một trường dữ liệu có giá trị là 1 đối tượng của class B. Khi đó nếu như bên trong object của class A có thể truy cập thông tin của object class B một cách trực tiếp thông qua B.field\_name thì đó chính là vi phạm về content coupling

Để tránh điều này thì đối tượng lớp A không được trực tiếp truy cập vào nội dung của đối tượng lớp B mà phải thông qua method của đối tượng lớp B bằng cách set private cho field data và sử dụng các getter, setter method để truy xuất dữ liệu.

Ví dụ cho tính nghiêm trọng của content coupling đó là Balance cho tài khoản ngân hàng, chúng ta không get, set trực tiếp số tiền được mà phải thay đổi bằng cách gọi các hàm trừ tiền

* + 1. **Common coupling**

Common coupling xảy ra khi mà 2 hay nhiều module cùng đọc và thay đổi 1 dữ liệu dùng chung. Đây cũng là một thiết kế tệ bởi vì nó gây ra sự không rõ ràng về vai trò của dữ liệu, code sẽ khó đọc, khó có thể xác định những thành phần code nào liên quan và ảnh hưởng đến dữ liệu dùng chung. Do đó giảm khả năng bảo trì hay tái sự dụng component. Common coupling thường xảy ra ở các ngôn ngữ lập trình hướng cấu trúc như C bởi việc sử dụng data global là thường xuyên.

* + 1. **Control coupling**

Xảy ra khi mà tham số truyền vào cho module sẽ quyết định luồng xử lý của module theo những cách khác nhau. Điều này cũng rất thường gặp khi chúng ta truyền tham số vào hàm và tham số đó có thể chia thành nhiều nhóm dữ liệu. Do đó bên trong hàm chúng ta sẽ dùng nhiều if else để check tham số truyền vào. Và hãy hình dung xem, với mỗi loại tham số truyền vào thì function đó thực ra chỉ chạy 1 phần code trong 1 block if else thoã mãn điều kiện và đương nhiên rằng những phần code còn lại là không được chạy

* + 1. **Stamp coupling**

Đến đây thì vi phạm coupling đã không còn ở mức high nữa mà đã có thể chấp nhận được. Stamp coupling xảy ra khi tham số truyền vào cho module là thừa, việc xử lý có thể chỉ cần một vài trường dữ liệu của object nhưng chúng ta truyền nguyên object vào.

Ví dụ như function printInvoice() và chúng ta truyền customer trong khi chỉ thông tin tên và địa chỉ khách hàng vậy thì những thông tin khác nằm trong customer là không cần thiết.

* + 1. **Data coupling**

Là coupling ở mức thấp nhất khi mà các modules tương tác với nhau chỉ thông qua tham số truyền vào. Điều này là không thể tránh khỏi do vậy thiết kế này là được cho là mục tiêu hướng đến. Data coupling và stamp coupling đều được xác định bởi tham số truyền vào nhưng khác ở chỗ data coupling không dư thừa dữ liệu tham số truyền vào, tất cả tham số truyền vào đều được sử dụng để xử lý.

* 1. **Cohension**

Sự gắn kết (cohension) phản ánh tính “đơn mục đích” của một thành phần. Trong bối cảnh thiết kế cấp thành phần cho các hệ thống hướng đối tượng, sự gắn kết chỉ ra rằng một thành phần hoặc lớp đóng gói chỉ các thuộc tính và các hoạt động có liên quan chặt chẽ với nhau và với một lớp hoặc thành phần của chính nó. Cohesion thì ngược lại so với Coupling, mức độ cohesion càng thấp thì thiết kế đó càng tệ và ngược lại thì cohesion càng cao thì thiết kế đó càng tốt.

A diagram of a diagram of a person's body

Description automatically generated with medium confidence

* + 1. **Coincidental**

Nghĩa là tình cờ, ngụ ý nói đến những elements nằm trong component một cách lạc lõng và đơn độc, nó chỉ có điểm chung là nằm chung vị trí với các component khác chứ không liên quan đến mục tiêu thể hiện của component. Theo như định nghĩa về cohesion thì những elements này không hề có tính liên kết gì với các elements khác trong component đó là lý do vì sao coincidental cohesion được xếp ở vị trí thấp nhất. Những ví dụ về loại cohesion này cũng rất thường gặp và dễ phát hiện bởi chúng ta dễ dàng nhận ra sự lạc loài khi có sự xuất hiện một method hay thuộc tính nào đó mà nó chẳng phục vụ cho xử lý logic gì liên quan đến chức năng của module. Ví dụ như chúng ta thêm function create random number trong class Customer hoặc cho thuộc tính is\_handsome vào class Customer, nó chẳng được sử dụng ở đâu cả.

* + 1. **Logical cohesion**

Được định nghĩa là khi các components liên quan đến nhau một cách logic chứ không phải là liên quan với nhau theo chức năng. Ví dụ như các functions đọc dữ liệu đầu vào từ tape, disk hay network cùng ở chung 1 module nghe có vẻ hợp lý và vì chúng liên quan đến nhau đó là xử lý dữ liệu đầu vào nhưng rõ ràng chức năng của chúng là khác nhau hoàn toàn. Cách giải quyết vấn đề này là chúng ta tạo 1 interface có method là readInput() để các sub-class implement đến override lại mothod readInput(). Sub-class Tape sẽ đọc từ tape, sub-class disk sẽ đọc từ disk, và tương tự với network mà với những chức năng đọc từ nguồn khác cũng được mở rộng tương tự.

* + 1. **Temporal cohesion**

Những elements liên quan đến nhau theo thời gian chứ không theo chức năng và những elements này được thực thi gần như trong cùng một khoảng thời gian. Để có thể hình dung cohesion này, chúng ta xem xét lúc bắt đầu chạy chương trình, rất nhiều những hành động xảy ra để khởi tạo tất cả các thành phần của hệ thống trong cùng một khoảng thời gian.

* + 1. **Procedural cohesion**

Những elements liên quan đến nhau chỉ để đảm bảo một thứ tự thực thi cụ thể. Những component được thiết kế elements như vậy vẫn có cohesion ở mức yếu và khó để tái sử dụng. Ví dụ như trong class Student sẽ thực hiện chức năng tính điểm tích luỹ trung bình vào cuối học kỳ mà các method được thực hiện một cách tuần tự: Tính GPA trung bình của học kỳ, lưu thông tin GPA học kỳ vào database, tính toán điểm trung bình tích luỹ tất cả các học kỳ, lưu vào database.

* + 1. **Communicational cohesion**

Là một nhóm các elements của module cùng hoạt động trên cùng một data là dữ liệu I/O của các methods. Ví dụ như chúng ta có một class Invoice trong một ứng dụng quản lý hóa đơn. Class này có thể bao gồm các phương thức như calculateTotal, addItem, và removeItem, tất cả đều làm việc trên dữ liệu hóa đơn. Invoice là một ví dụ về communicational cohesion bởi vì tất cả các phương thức đều làm việc trên cùng một loại dữ liệu, đó là danh sách các mục hóa đơn (items). Điều này giúp cho việc quản lý và bảo trì code trở nên dễ dàng hơn, vì tất cả các chức năng liên quan đến quản lý các mục hóa đơn đều được tập trung trong cùng một nơi

* + 1. **Sequential Cohesion**

Khi output của một element trở thành input của một element khác. Điều này thương xảy ra ở các ngôn ngữ lập trình hướng chức năng(Functional programming languages – scala, SML, Clojure, Erlang,..) . Đây đã được xem như là thiết kế tốt (high cohesion).

* + 1. **Functional cohesion**

Đây là mức cao nhất của cohesion khi mà tất cả các elements trong component đều là cần thiết cho mục đích của component.

* 1. **The Lack of Cohesion in Methods (LCOM)**

The Lack of Cohesion in Methods (LCOM) để xác định mức độ mà các phương thức (methods) trong một lớp (class) liên kết chặt chẽ với nhau. Nói cách khác, LCOM giúp nhận diện sự không liên kết trong việc thực hiện nhiệm vụ giữa các phương thức trong cùng một lớp.

Khi các phương thức của một lớp tương tác và chia sẻ dữ liệu (ví dụ: thông qua các trường (fields) của lớp) một cách chặt chẽ, lớp đó có tính liên kết cao – nghĩa là có một LCOM thấp. Mặt khác, nếu các phương thức trong một lớp ít hoặc không sử dụng chung dữ liệu, thì lớp đó sẽ có tính liên kết thấp – hay một LCOM cao, điều này ngụ ý rằng có thể cần phải cải tiến thiết kế lớp để tăng tính liên kết.

LCOM được tính bằng cách so sánh số lượng các cặp phương thức không chia sẻ dữ liệu (tức là không có các trường chung) với số lượng các cặp phương thức có chia sẻ dữ liệu. Từ đó có thể đặt ra một số phiên bản để tính LCOM, phổ biến nhất là:

- LCOM1 (Lack of Cohesion of Methods version 1): Đây là phiên bản gốc được đề xuất bởi Chidamber và Kemerer, nhằm định lượng mức độ mà các phương thức của một lớp hỗ trợ cùng một chức năng. Nếu các phương thức không sử dụng các biến lớp chung, lớp đó có thể được coi là có tính liên kết thấp.

- LCOM2 và LCOM3: Những cách tiếp cận cải tiến sau LCOM1, nhằm khắc phục nhược điểm của LCOM1, bao gồm việc xử lý chính xác hơn vấn đề của các phương thức truy cập độc lập với các kiểu dữ liệu khác nhau.

- LCOM4: Đánh giá việc phụ thuộc liên kết giữa các phương thức và trường dữ liệu dựa trên cấu trúc của lớp. Nếu một lớp có nhiều hơn một thành phần liên kết (tức là có các nhóm phương thức và trường dữ liệu không liên quan đến nhau), thì nó có tính liên kết thấp và có thể cần được phân chia thành các lớp riêng biệt.

- LCOM5: Không phải là một chỉ số được công nhận rộng rãi hoặc chuẩn mực, và thông tin về nó rất hạn chế. Phỏng đoán rằng LCOM5 có thể là một nỗ lực nhằm cung cấp một định nghĩa cải tiến hơn nữa so với các tiêu chí LCOM trước đó, nhưng việc sử dụng và ý nghĩa cụ thể của nó không rõ ràng.

Nhìn chung, các biến thể của LCOM giúp các nhà phát triển phần mềm xác định được:

- Các vấn đề về thiết kế trong lớp có thể gây khó khăn cho việc hiểu code, bảo trì và mở rộng.

- Các khu vực mà mã có thể cần phải được cải tiến để tăng cường khả năng tái sử dụng và giảm phụ thuộc.

- Cách để cải thiện khả năng đọc và duy trì mã bằng cách tìm kiếm các cơ hội để refactor hoặc chia nhỏ lớp.

* 1. **Distance from the Main Sequence**

Một trong những chỉ số tổng thể ít ỏi mà kiến trúc sư có được cho cấu trúc kiến trúc là khoảng cách so với dãy chính, một chỉ số phái sinh dựa trên sự không ổn định và mức độ trừu tượng.

- Các lớp càng gần vạch thì độ cân bằng của nó càng tốt

- Các lớp nằm ở càng xa dãy chính về phía trên bên phải thì sẽ rơi vào vùng vô dụng: code quá trừu tượng, khó sử dụng

- Ngược lại, nếu rơi về phía dưới bên trái thì sẽ rơi vào vùng khó khăn: code được triển khai quá nhiều, không đủ độ trừu tượng gây khó khăn trong việc bảo trì

**Hạn chế**

- Trong khi ngành công nghiệp có một số độ đo code-level cung cấp cái nhìn sâu sắc có giá trị về các cơ sở mã lệnh, công cụ của chúng ta lại cực kỳ thô sơ so với các công cụ phân tích từ các ngành kỹ thuật khác.

- Thậm chí các độ đo được rút ra trực tiếp từ cấu trúc của code cũng cần được giải thích. Ví dụ, độ phức tạp chu trình đo lường độ phức tạp trong cơ sở mã lệnh nhưng không thể phân biệt được giữa độ phức tạp cần thiết (bởi vì vấn đề cơ bản là phức tạp) hay độ phức tạp ngẫu nhiên (code phức tạp hơn so với mức nó cần).

- Hầu như tất cả các độ đo code-level đều cần được giải thích, nhưng việc thiết lập các tiêu chuẩn cơ bản cho các độ do quan trọng như độ phức tạp chu trình vẫn hữu ích để các kiến trúc sư có thể đánh giá loại nào chúng biểu hiện.

* 1. **Connascence**

“Hai component là connascence nếu một thay đổi trong một component sẽ yêu cầu thay đổi component kia để duy trì tính đúng đắn chung của hệ thống” – Meilir Page-Jones

**Các loại connascence**

**- Static**: source-code-level coupling. Static Connascence xảy ra khi mối liên kết hoặc phụ thuộc có thể được xác định và hiểu rõ tại thời điểm biên dịch – nói cách khác, trước khi chương trình thực thi. Đây là những kết nối giữa mã nguồn mà bạn có thể nhìn thấy và định vị mà không cần chạy chương trình.

**- Dynamic**: execution-time coupling. Dynamic Connascence, ngược lại, chỉ có thể được đánh giá hoặc hiểu rõ khi chương trình đang chạy. Các phụ thuộc này không thể xác định chính xác chỉ bằng cách nhìn vào mã nguồn mà cần phải xem xét cả ngữ cảnh thực thi của chương trình.

* + 1. **Connascense of Name (CoN)**

Connascence of Name (CoN) mô tả mức độ phụ thuộc giữa các module hoặc thành phần trong hệ thống thông qua việc chúng chia sẻ cùng một tên. Nói cách khác, CoN đo lường mức độ mà việc thay đổi tên của một phần tử sẽ ảnh hưởng đến các phần tử khác.

Một ví dụ điển hình là việc sử dụng tên biến hoặc tên hàm. Nếu một hàm trong mã nguồn gọi một biến với tên cụ thể, và bất kỳ thay đổi nào đối với tên biến đó cũng đòi hỏi phải thay đổi tương ứng ở tất cả các nơi sử dụng biến đó, thì đó là Connascence of Name.

Thường các conascecce mức độ cao thường được khuyến khích xử lý và đưa về CoN (ví dư như: Connascence of Position, Connascence of Meaning), do CoN có strength thấp nhất và dễ dàng xử lý nhất do hiện nay các IDE hiện đại của các Statically Typed Languages hỗ trợ báo lỗi và hỗ trợ đổi tên khi xảy ra CoN. Với Dynamically Typed Languages thì hiện nay có thể sử dụng các extension như: linters hoặc kiểm thử tự động

Cách giảm thiếu CoN

* + - Với Statically Typed Languages thì hiện nay có các IDE hỗ trợ báo lỗi tên không khớp và hỗ trợ đổi tên (Intellij, Eclipse). Với Dynamically Typed Languages có thể sử dụng extension: Linters (ví dụ: ESLint, JSHint, ...)
* Sử dụng tên có ý nghĩa và duy nhất: Tránh sử dụng các tên chung chung hoặc dễ gây nhầm lẫn.
* Tổ chức mã nguồn hợp lý: Chia mã nguồn thành các module nhỏ và độc lập.
* Sử dụng các công cụ phân tích mã tĩnh: Các công cụ này có thể giúp phát hiện và cảnh báo về các trường hợp CoN cao.
* Áp dụng tốt các nguyên tắc thiết kế: Ví dụ, nguyên tắc Đơn trách nhiệm (Single Responsibility Principle) và nguyên tắc Đóng mở (Open/Closed Principle)
* Sử dụng các kỹ thuật đóng gói để ẩn đi các chi tiết triển khai, giảm bớt sự cần thiết phải biết tên cụ thể.
* Sử dụng các mức độ trừu tượng cao hơn như interfaces hoặc abstract classes để giảm sự phụ thuộc vào tên cụ thể.
  + 1. **Connascence of Type (CoT)**

Connascence of Type (CoT) là một khái niệm trong thiết kế phần mềm, mô tả mức độ phụ thuộc giữa các thành phần trong hệ thống dựa trên việc chúng chia sẻ hoặc tương tác với cùng một kiểu dữ liệu. Nói cách khác, CoT đo lường mức độ mà việc thay đổi kiểu dữ liệu của một thành phần sẽ ảnh hưởng đến các thành phần khác.

Ví dụ: Một function nhận một đối số kiểu int do đó chúng ta không thể truyền giá trị string vào function

Cách giảm thiểu CoT

* + - Với Statically Typed Languages thì các lỗi này thường được bắt bỏi compiler (không phải là luôn luôn). Với Dynamically Typed Languages có thể giảm thiểu bằng việc tuân theo các nguyên tắc của nhóm phát triển (Viết comment, document), viết các automated test.
* Sử dụng interface hoặc abstract class: Thay vì sử dụng trực tiếp một kiểu dữ liệu cụ thể, hãy sử dụng interface hoặc abstract class để định nghĩa các phương thức và thuộc tính mà các thành phần cần.
* Sử dụng dependency injection: Thay vì để các thành phần tự tạo các đối tượng của các kiểu dữ liệu khác, hãy tiêm các đối tượng này từ bên ngoài.
* Sử dụng các công cụ phân tích mã tĩnh: Các công cụ này có thể giúp phát hiện và cảnh báo về các trường hợp CoT cao.
* Đóng gói các chi tiết triển khai để giảm bớt sự cần thiết phải biết kiểu dữ liệu cụ thể.
  + 1. **Connascence of Meaning (CoM) or Connascence of Convention (CoC)**

CoM xảy ra khi hai hoặc nhiều thành phần phải đồng ý về ý nghĩa của một giá trị cụ thể. Nói cách khác, các thành phần này hiểu và diễn giải giá trị đó theo cùng một cách. Nếu ý nghĩa của giá trị thay đổi, tất cả các thành phần liên quan đều cần được cập nhật.

CoC là một dạng đặc biệt của CoM, xảy ra khi ý nghĩa của một giá trị không được xác định rõ ràng, mà dựa trên một quy ước ngầm giữa các thành phần.

Ví dụ, mã lỗi 404 có ý nghĩa là “Không tìm thấy”, nếu ai đó sửa đổi ý nghĩa của nó thành “Lỗi xác thực” thì chúng ta cần phải cập nhật lại tất cả các component có sử dụng mã lỗi đó

Cách giảm thiểu CoM và CoC

* Sử dụng tên có ý nghĩa và rõ ràng: Đặt tên biến, hàm, lớp, v.v. sao cho chúng thể hiện rõ ý nghĩa của mình, tránh sử dụng các giá trị "ma thuật" (magic values) không có giải thích.
* Tài liệu hóa các quy ước: Nếu phải sử dụng quy ước ngầm, hãy ghi chép rõ ràng trong tài liệu để các thành viên khác trong nhóm hiểu và tuân thủ.
* Sử dụng kiểu dữ liệu mạnh (strong types): Thay vì sử dụng các kiểu dữ liệu cơ bản (như số nguyên, chuỗi), hãy tạo các kiểu dữ liệu mới (Enum) để thể hiện rõ ràng ý nghĩa của dữ liệu.
* Giới hạn phạm vi của các quy ước: Cố gắng giới hạn phạm vi mà các quy ước được áp dụng, ví dụ chỉ sử dụng chúng trong một module hoặc lớp cụ thể.
  + 1. **Connascence of Position (CoP)**

Connascence of Position chỉ sự phụ thuộc giữa các thành phần của mã nguồn dựa trên vị trí hoặc thứ tự của chúng. Khi có Connascence of Position, các thành phần phải được sắp xếp hoặc xuất hiện theo một thứ tự cụ thể để hệ thống hoạt động đúng. Nếu thứ tự này bị thay đổi, hệ thống có thể gặp lỗi hoặc hoạt động không chính xác.

Ví dụ nếu muốn tính khoảng cách của 2 điểm, ta phải truyền toạ độ của các điểm đúng thứ tự của nó. Nếu thứ tự của các tham số được thay đổi, thì lời gọi hàm cũng cần được cập nhật.

Cách giảm thiểu CoP

* + - Nếu function có danh sách dài các đối hay số có kích thước lớn và chịu trách nhiệm cho quá nhiều thứ thì việc refactoring thành nhiều functions không chỉ giảm connascence mà còn cải thiện khả năng dễ đọc và bảo trì code.
    - Nhóm các đối số thành 1 kiểu dữ liệu (DTO) có thể là một giải pháp tốt giúp giảm level hiện tại (CoP) xuống thành Connascence of type
* Sử dụng các cấu trúc dữ liệu linh hoạt: Thay vì sử dụng các cấu trúc dữ liệu có thứ tự cố định (như mảng), hãy sử dụng các cấu trúc dữ liệu cho phép thay đổi vị trí của các phần tử một cách dễ dàng (như danh sách liên kết hoặc cây) hoặc sử dụng kiểu dữ liệu không quan
* Sử dụng tên có ý nghĩa cho các tham số: Thay vì dựa vào vị trí của các tham số trong lời gọi hàm, hãy đặt tên cho chúng để thể hiện rõ ý nghĩa của chúng.
* Sử dụng các công cụ phân tích mã tĩnh: Các công cụ này có thể giúp phát hiện và cảnh báo về các trường hợp CoP cao.
* Sử dụng các hàm hỗ trợ: Chia nhỏ một hàm lớn, có nhiều tham số phụ thuộc vào vị trí, thành các hàm nhỏ hơn, mỗi hàm chỉ tập trung vào một nhiệm vụ cụ thể
  + 1. **Connascence of Algorithm (CoA)**

Connascence of Algorithm là một khái niệm trong thiết kế phần mềm, đặc biệt liên quan đến sự phụ thuộc giữa các thành phần của mã nguồn dựa trên thuật toán cụ thể mà chúng thực hiện. Khi có Connascence of Algorithm, các thành phần phải sử dụng cùng một thuật toán hoặc phải hiểu và tương thích với thuật toán được sử dụng ở một phần khác của hệ thống. Nếu thuật toán thay đổi, các phần liên quan của hệ thống cũng phải thay đổi tương ứng để duy trì tính đúng đắn và hiệu quả.

Ví dụ trong các hệ thống mã hóa và giải mã, cả hai bên phải sử dụng cùng một thuật toán và khóa để đảm bảo dữ liệu được mã hóa có thể được giải mã chính xác.

Cách giảm thiểu CoA

* Cô lập thuật toán: Tách biệt các thuật toán thành các thành phần (component) riêng biệt, độc lập với phần còn lại của hệ thống. Điều này giúp dễ dàng thay đổi hoặc thay thế thuật toán mà không ảnh hưởng đến các thành phần khác.
* Sử dụng các giao diện (interfaces) hoặc lớp trừu tượng (abstract classes): Định nghĩa các giao diện hoặc lớp trừu tượng để trừu tượng hóa các thuật toán. Các thành phần khác có thể tương tác với các thuật toán thông qua các giao diện hoặc lớp trừu tượng này, giúp giảm sự phụ thuộc trực tiếp vào việc triển khai cụ thể của thuật toán.
* Sử dụng các thư viện hoặc framework: Sử dụng các thư viện hoặc framework cung cấp các thuật toán đã được triển khai và kiểm thử kỹ lưỡng. Điều này giúp tiết kiệm thời gian và công sức, đồng thời đảm bảo tính chính xác và hiệu quả của các thuật toán.
  + 1. **Connascence of Execution (CoE)**

Connascence of Execution (CoE) mô tả mức độ phụ thuộc giữa các thành phần trong hệ thống dựa trên thứ tự thực thi của chúng. Nói cách khác, CoE đo lường mức độ mà việc thay đổi thứ tự thực thi của một thành phần sẽ ảnh hưởng đến các thành phần khác.

Ví dụ một đối tượng phải được khởi tạo trước khi các phương thức của nó có thể được gọi. Nếu thứ tự này bị đảo ngược, chương trình sẽ gặp lỗi.

Cách giảm thiểu CoE

* Thiết kế hướng sự kiện (Event-driven design): Thay vì thực thi các thành phần theo một thứ tự cố định, hãy để chúng phản ứng với các sự kiện. Điều này giúp giảm sự phụ thuộc vào thứ tự thực thi.
* Sử dụng các công cụ đồng bộ hóa: Trong các hệ thống bất đồng bộ, hãy sử dụng các công cụ đồng bộ hóa như mutex, semaphore, hoặc monitor để đảm bảo các tác vụ được thực thi theo đúng thứ tự.
* Viết mã nguồn rõ ràng và dễ hiểu: Sử dụng các tên biến và hàm có ý nghĩa, viết tài liệu rõ ràng, và tuân thủ các quy ước về mã nguồn để giúp người khác dễ dàng hiểu được luồng thực thi của chương trình.
* Tách biệt trách nhiệm (separation of concerns): Phân chia chức năng rõ ràng để giảm sự phụ thuộc chéo.
* Quản lý phụ thuộc bằng Dependency Injection: Sử dụng các mẫu thiết kế như Dependency Injection để quản lý và kiểm soát phụ thuộc giữa các thành phần.
  + 1. **Connascence of Timing (CoT)**

Connascence of Timing xảy ra khi hai hoặc nhiều thành phần của hệ thống phần mềm cần phải được thực thi trong một khung thời gian nhất định hoặc phải tuân theo một số ràng buộc thời gian cụ thể để hệ thống hoạt động đúng. Loại connascence này đặc biệt quan trọng trong các hệ thống thời gian thực hoặc các hệ thống có yêu cầu về hiệu suất và đồng bộ hóa cao.

Trong các hệ thống xử lý đồng thời, nhiều luồng thực thi có thể truy cập vào cùng một tài nguyên được chia sẻ. Nếu không có cơ chế đồng bộ hóa thích hợp, thứ tự truy cập vào tài nguyên có thể ảnh hưởng đến kết quả của chương trình.

Cách giảm thiểu CoT

* Thiết kế đồng bộ: Thiết kế hệ thống sao cho các thành phần không cần phải hoạt động đồng thời hoặc có sự tương tác theo thời gian.
* Sử dụng các công cụ đồng bộ hóa: Trong các hệ thống bất đồng bộ hoặc xử lý đồng thời, hãy sử dụng các công cụ đồng bộ hóa như mutex, semaphore, hoặc monitor để đảm bảo các thành phần truy cập vào tài nguyên được chia sẻ một cách an toàn và có thứ tự.
* Giảm thiểu sự phụ thuộc thời gian: Cố gắng thiết kế các thành phần sao cho kết quả của chúng ít phụ thuộc vào thời điểm chúng được thực thi. Ví dụ, bạn có thể sử dụng các kỹ thuật như buffering hoặc caching để giảm thiểu sự phụ thuộc vào thời gian của các hoạt động nhập xuất.
* Sử dụng các kỹ thuật xử lý bất đồng bộ: Thay vì chờ đợi các phần mã khác hoàn thành, sử dụng các kỹ thuật xử lý bất đồng bộ để cho phép các phần mã thực thi độc lập và trao đổi thông tin thông qua các kênh giao tiếp như hàng đợi, sự kiện hoặc các kỹ thuật tương tự.
  + 1. **Connascence of Values (CoV)**

Connascence of Values (CoV) mô tả mức độ phụ thuộc giữa các thành phần trong hệ thống dựa trên việc chúng chia sẻ hoặc ràng buộc với cùng một giá trị hoặc một tập hợp các giá trị. Nói cách khác, CoV đo lường mức độ mà việc thay đổi một giá trị sẽ ảnh hưởng đến các giá trị khác hoặc các thành phần khác trong hệ thống.

Nếu hai đoạn code cùng sử dụng một biến toàn cục, sự thay đổi giá trị của biến này trong một đoạn code có thể ảnh hưởng đến hoạt động của đoạn code còn lại. Ví dụ, nếu một đoạn code đặt giá trị của biến toàn cục là true và phần mã khác kiểm tra giá trị này, sự thay đổi giá trị này trong phần mã đầu tiên có thể dẫn đến kết quả không mong muốn trong đoạn code thứ hai.

Cách giảm thiểu CoV

* Sử dụng các ràng buộc toàn vẹn (integrity constraints) trong cơ sở dữ liệu: Các ràng buộc toàn vẹn có thể giúp tự động duy trì tính nhất quán của dữ liệu.
* Sử dụng các đối tượng giá trị (value objects): Các đối tượng giá trị là các đối tượng không thay đổi (immutable) đại diện cho một giá trị hoặc một tập hợp các giá trị. Việc sử dụng các đối tượng giá trị có thể giúp giảm thiểu CoV bằng cách đóng gói các giá trị liên quan vào một đối tượng duy nhất.
* Sử dụng các dịch vụ miền (domain services): Các dịch vụ miền có thể được sử dụng để thực hiện các quy tắc nghiệp vụ và đảm bảo tính nhất quán của dữ liệu.
* Sử dụng dependency injection: Truyền các giá trị phụ thuộc vào các thành phần thay vì sử dụng các giá trị cố định toàn cục.
  + 1. **Connascence of Identity (CoI)**

Connascence of Identity (CoI) là một khái niệm trong thiết kế phần mềm, mô tả mức độ phụ thuộc giữa các thành phần trong hệ thống dựa trên việc chúng tham chiếu đến cùng một thực thể (entity) hoặc đối tượng (object). Nói cách khác, CoI đo lường mức độ mà việc thay đổi một thực thể sẽ ảnh hưởng đến các thành phần khác tham chiếu đến nó.

Ví dụ mẫu thiết kế Singleton đảm bảo rằng chỉ có một phiên bản của một lớp tồn tại trong hệ thống. Tất cả các thành phần cần sử dụng lớp đó phải tham chiếu đến cùng một phiên bản duy nhất, tạo ra CoI.

Cách giảm thiểu CoI

* Hạn chế sử dụng Singleton và đối tượng toàn cục: Thay vì sử dụng Singleton và đối tượng toàn cục, hãy xem xét các phương pháp khác như Dependency Injection để quản lý sự phụ thuộc giữa các thành phần.
* Truyền giá trị (Pass-by-value): Khi có thể, hãy truyền các đối tượng vào hàm bằng giá trị để tránh chia sẻ đối tượng giữa các thành phần.
* Sử dụng các interface hoặc abstract class: Thay vì tham chiếu trực tiếp đến một lớp cụ thể, hãy sử dụng các interface hoặc abstract class để giảm sự phụ thuộc giữa các thành phần.

**1.8 Apply fundamentals to Project**

**1.8.1 Instability - Coupling (Efferent Coupling, Afferent Coupling)**

1.8.1.1 Student Service

* Afferent coupling của các package dao động từ 0-24 nằm trong khoảng tốt (CA nên nằm trong khoảng 0-500)
* Efferent coupling của các package dao động từ 0-24, chỉ có 1 package nằm ngoài vùng an toàn (0-20 vùng ổn định, > 20 sẽ không ổn định) còn các package còn lại <= 20 nằm trong vùng ổn định giá trị càng thấp thì dễ phát triển và bảo trì.
* Instability: Loại bỏ các package không có Effent Coupling, Afferent Coupling, Có 2 package không ổn định là: (convert: 0.92, service: 0.91), Không có package nào nằm trong giá trị nên tránh (giá trị nên tránh là từ 0.4-0.6), các package có I càng tiến đến 0 cho thấy các package càng ổn định

A screenshot of a computer

Description automatically generated 1.8.1.2 Lecturer Service

- Afferent coupling của các package dao động từ 0-25 nằm trong khoảng (0-500) cho thấy các component có độ ổn định cao

- Efferent Coupling của các package dao động từ 0-47, có 1 package có CE cao hơn 20 là: convert (47) cho thấy package này không ổn định, còn các package còn lại đều có CE <= 20, CE càng thấp càng dễ phát triển và bảo trì

- Instability: Loại bỏ các Afferent Coupling và Efferent Coupling = 0, Có 2 package không ổn định là: service : 0.91 và convert: 0.94. Có 1 package có I nên tránh là 0.5 (giá trị nên tránh là: 0.4-0.6), các package còn lại tương đối ổn định

A screenshot of a computer

Description automatically generated 1.8.1.3 Subject Service

- Afferent coupling của các package dao động từ 0-71 nằm trong khoảng (0-500) cho thấy các component có độ ổn định cao

- Efferent Coupling của các package trong service này tương đối cao, có đến 3 package nằm ngoài vùng ổn định (consumer, converters, repositories) nếu có thể cần refactor lại các package trên

- Instability: Bỏ đi những package có Afferent coupling và Efferent Coupling là 0, có khá nhiều package không ổn định, có 1 package nên tránh và 1 package nằm trong vùng ổn định

A screenshot of a computer

Description automatically generated

**1.8.2 Apply Connascence to Project**

**1.8.2.1 Connascence of Name and Connascence of Type**

- Với Java và các Static Typed Lanaguage đã có IDEs hỗ trợ báo lỗi và giúp renaming

- Với Javascript (ReactJS) thì nhóm có sử dụng một số Linters extensions như: ESLint để giải quyết Connascence of Name và Connascence of Type ở Dynamicly Typed Language

1.8.2.2 Fix Connnascence of Meaning (Convert to Connascence of Name)

- Với các giá trị mang ý nghĩa, được sử dụng để xử lý ở nhiều class thì trong dự án đưa các giá trị dùng ở nhiều class đó về Enum, khi thay đổi giá trị chỉ cần thay đổi ở Enum, sau đây là một vài demo giải quyết Connascence of Meaning bằng cách đưa về Connascence có mức độ mạnh thấp hơn:

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

1.8.2.3 Fix Connascence of Position (Convert to Connascence of Name)

* + Với các function có nhiều hơn 2 function có khả năng xảy ra Connascence of Position, trong dự án nhóm fix vấn đề này bằng cách tạo ra các DTO như là các đối số cho function đó, giải quyết vấn đề bằng cách đưa nó về Connascence of Name, sau đây là demo:

A screenshot of a computer

Description automatically generated

1. **Common software architecture styles**
   1. **Layered Architecture Style**
      1. **Ưu điểm**

- Phát triển phần mềm nhanh do kiến trúc đơn giản, dễ dàng tìm hiểu và hiện thực. Ví dụ, nhóm phát triển giao diện người dùng (UI/UX) chỉ cần tập trung vào Presentation Layer mà không cần quan tâm đến logic nghiệp vụ hoặc truy xuất dữ liệu.

- Giảm phụ thuộc giữa các chức năng do chức năng của mỗi lớp là tách biệt với lớp khác, việc giảm phụ thuộc giúp tăng khả năng bảo trì, mở rộng. Ví dụ, Khi thay đổi logic nghiệp vụ để cập nhật quy trình xét tốt nghiệp, chỉ cần thay đổi trong Business Logic Layer mà không ảnh hưởng đến các lớp khác

- Chi phí rẻ, dễ dàng chuyển đổi qua phong cách kiến trúc khác hơn so với các kiến trúc phần mềm khác. Ví dụ, khi hệ thống phát triển lớn hơn và cần chuyển đổi sang kiến trúc microservices để tăng khả năng mở rộng, mỗi lớp trong kiến trúc layered có thể được tách thành các dịch vụ độc lập mà không gặp nhiều khó khăn.

- Dữ liệu nhất quán, việc quản lý và hiện thực các transaction dễ và đơn giản hơn do nó là kiến trúc nguyên khối. Ví dụ, khi sinh viên đăng ký học phần, hệ thống sẽ kiểm tra điều kiện tiên quyết và số tín chỉ tối đa trong Business Logic Layer, sau đó cập nhật trạng thái đăng ký trong cơ sở dữ liệu ở Data Access Layer. Do dữ liệu được quản lý tập trung, các thay đổi sẽ được cập nhật ngay lập tức và nhất quán.

- Dễ dàng theo dõi các log và truy vết các vấn đề, lỗi (monolithic). Ví dụ, khi có lỗi xảy ra trong quá trình sinh viên đăng ký học phần, hệ thống có thể ghi lại các log chi tiết từ Presentation Layer, Business Logic Layer đến Data Access Layer, giúp dễ dàng xác định nguyên nhân và khắc phục sự cố.

* + 1. Nhược điểm

- Khó có thể kiểm thử tích hợp. Ví dụ khi kiểm thử quá trình đăng ký học phần, cần kiểm thử tích hợp từ giao diện người dùng (Presentation Layer), logic kiểm tra điều kiện tiên quyết và số tín chỉ (Business Logic Layer) đến việc cập nhật cơ sở dữ liệu (Data Access Layer). Điều này yêu cầu kiểm thử toàn bộ các lớp cùng một lúc, làm tăng độ phức tạp và thời gian kiểm thử.

- Ngẽn cổ chai khi truy cập đông: Vào thời điểm mở đăng ký học phần, hàng ngàn sinh viên cùng lúc truy cập hệ thống để đăng ký. Nếu tất cả các yêu cầu đều được xử lý bởi một máy chủ duy nhất, máy chủ này có thể bị quá tải và dẫn đến tình trạng chậm phản hồi hoặc thậm chí sập hệ thống

- Xử lý song song là không khả thi bởi vì mỗi lớp trong kiến trúc layered được thiết kế để xử lý tuần tự.

- Khả năng đàn hồi (Elasticity) kém: Trong kỳ đăng ký học phần, nhu cầu tài nguyên tăng đột biến, nhưng vào các thời điểm khác, nhu cầu tài nguyên lại rất thấp. Việc duy trì một hệ thống với tài nguyên lớn suốt thời gian là không hiệu quả về mặt chi phí. Nếu hệ thống không có khả năng tự động điều chỉnh tài nguyên (auto scaling), điều này sẽ dẫn đến lãng phí tài nguyên hoặc gặp vấn đề khi có lượng truy cập lớn.

- Khó khăn trong việc bảo trì, việc một layer gặp lỗi có thể ảnh hưởng lên toàn bộ hệ thống. Ví dụ, nếu có một lỗi xảy ra ở tầng Data Access (ví dụ: lỗi kết nối cơ sở dữ liệu), toàn bộ ứng dụng có thể bị ảnh hưởng. Việc tìm kiếm và sửa chữa lỗi có thể mất nhiều thời gian và ảnh hưởng đến hoạt động của hệ thống.

- Hệ thống nguyên khối thường có khả năng chịu lỗi kém, một lỗi nhỏ có thể làm giảm tính sẵn sàng của toàn bộ hệ thống. Nếu có sự cố về server trong kỳ đăng ký học phần, toàn bộ hệ thống sẽ bị ảnh hưởng, khiến sinh viên không thể truy cập các tiện ích khác như xem thông tin học tập, thời khóa biểu, hoặc đăng ký xét tốt nghiệp.

* 1. **Pipeline Architecture Style**
     1. **Ưu điểm**

- Tính Cohesion cao và Coupling lỏng lẻo: Trong pipeline đăng ký học phần, ta có thể có một filter riêng biệt chịu trách nhiệm kiểm tra điều kiện tiên quyết của môn học (ví dụ: sinh viên đã hoàn thành các môn học yêu cầu trước đó chưa). Filter này chỉ tập trung vào một nhiệm vụ cụ thể, đảm bảo tính cohension cao. Đồng thời, filter này không cần biết chi tiết về cách thức hoạt động của các filter khác trong pipeline (ví dụ: filter kiểm tra số tín chỉ tối đa), chỉ cần nhận đầu vào và trả về kết quả, thể hiện tính coupling lỏng lẻo.

- Chi phí triển khai thấp: Thay vì phải xây dựng một hệ thống đăng ký học phần phức tạp từ đầu, ta có thể tận dụng các thư viện hoặc framework hỗ trợ kiến trúc pipeline (ví dụ: Apache Camel, Spring Integration). Các công cụ này cung cấp sẵn các thành phần cơ bản để xây dựng pipeline, giúp giảm thiểu thời gian và công sức phát triển.

- Có thể kiểm thử độc lập và tái sử dụng: Filter kiểm tra số tín chỉ tối đa có thể được kiểm thử độc lập bằng cách cung cấp các đầu vào khác nhau (ví dụ: sinh viên đã đăng ký 25 tín chỉ, sinh viên đã đăng ký 30 tín chỉ) và kiểm tra kết quả trả về. Filter này cũng có thể được tái sử dụng trong các pipeline khác (ví dụ: pipeline đăng ký khóa học hè) mà không cần phải sửa đổi.

- Tăng hiệu quả xử lý và tốc độ phản hồi: Kiến trúc pipeline cho phép chia nhỏ quá trình xử lý thành các bước độc lập và có thể thực hiện song song. Ví dụ, việc xác thực thông tin đăng nhập của sinh viên, kiểm tra điều kiện tiên quyết, xác nhận đăng ký, và gửi email thông báo đều có thể được xử lý ở các giai đoạn khác nhau mà không cần chờ đợi toàn bộ quy trình hoàn tất.

- Tính linh hoạt và dễ dàng mở rộng: Với kiến trúc pipeline, các bước xử lý được thiết kế thành các module độc lập. Khi cần thay đổi hoặc thêm mới một chức năng, chỉ cần thêm hoặc chỉnh sửa các module liên quan mà không ảnh hưởng đến toàn bộ hệ thống. Ví dụ, việc thêm một bước xác nhận đăng ký qua SMS bên cạnh email thông báo chỉ cần thêm một module mới vào pipeline.

- Đảm bảo tính toàn vẹn và chính xác của dữ liệu: Mỗi giai đoạn trong pipeline có thể được thiết kế để kiểm tra và xác nhận tính hợp lệ của dữ liệu trước khi chuyển sang giai đoạn tiếp theo. Điều này giúp giảm thiểu lỗi và đảm bảo rằng thông tin được xử lý chính xác và đầy đủ. Ví dụ, trước khi cho phép đăng ký môn học, hệ thống có thể kiểm tra tiên quyết và số tín chỉ tối đa của sinh viên.

- Dễ dàng theo dõi và quản lý quy trình: Mỗi giai đoạn trong pipeline có thể được giám sát và ghi nhận lại, giúp dễ dàng theo dõi và quản lý quá trình đăng ký học phần. Điều này hỗ trợ việc phân tích và cải thiện hiệu suất của hệ thống. Ví dụ, nếu nhiều sinh viên bị hủy đăng ký do không đủ điều kiện tiên quyết, nhà trường có thể dễ dàng nhận diện và điều chỉnh quy trình đào tạo.

* + 1. **Nhược điểm**

- Quản lý luồng dữ liệu: Việc chia nhỏ quy trình thành nhiều giai đoạn đòi hỏi phải quản lý luồng dữ liệu giữa các giai đoạn một cách cẩn thận. Điều này có thể làm tăng độ phức tạp của hệ thống, đặc biệt khi có nhiều nhánh xử lý khác nhau (ví dụ: sinh viên đăng ký thành công, sinh viên vào danh sách dự bị, sinh viên không đủ điều kiện).

- Xử lý lỗi: Khi xảy ra lỗi ở một giai đoạn, việc xác định nguyên nhân và phục hồi trạng thái hệ thống có thể trở nên khó khăn hơn so với kiến trúc truyền thống. Cần có cơ chế xử lý lỗi và ghi log chi tiết để hỗ trợ việc gỡ lỗi.

- Truyền dữ liệu giữa các giai đoạn: Việc truyền dữ liệu giữa các giai đoạn trong pipeline có thể gây ra độ trễ nhất định, đặc biệt khi dữ liệu có kích thước lớn hoặc khi có nhiều giai đoạn xử lý.

- Phụ thuộc giữa các giai đoạn: Nếu một giai đoạn phụ thuộc vào kết quả của giai đoạn trước đó, thì việc xử lý song song có thể bị hạn chế, dẫn đến độ trễ tăng lên.

- Khó khăn trong việc xử lý các tác vụ không đồng bộ: Việc gửi email xác nhận đăng ký là một tác vụ không đồng bộ, có thể mất một khoảng thời gian để hoàn thành. Nếu pipeline phải chờ tác vụ này hoàn thành mới chuyển sang giai đoạn tiếp theo, thì hiệu suất của hệ thống sẽ bị ảnh hưởng.

* 1. **Microkernel Architecture Style**
     1. **Ưu điểm**

- Kiến trúc tương đối đơn giản: Hệ thống sẽ được chia thành một kernel (lõi) đơn giản xử lý các tác vụ cơ bản như xác thực, quản lý người dùng và giao tiếp với các module plugin. Phần lớn các chức năng đặc thù như quản lý môn học, đăng ký, và theo dõi tiến độ học tập sẽ được triển khai dưới dạng các plugin độc lập. Điều này giúp đơn giản hóa việc phát triển và bảo trì hệ thống.

- Hiệu quả về mặt chi phí: Với ngân sách hạn chế, trường đại học có thể phát triển các plugin cơ bản trước để đáp ứng nhu cầu cấp bách như đăng ký học phần, quản lý tín chỉ, và kiểm tra điều kiện tiên quyết. Sau đó, các plugin bổ sung như quản lý thời khóa biểu, xem thông tin học tập, và theo dõi cựu sinh viên có thể được thêm vào khi có thêm ngân sách hoặc yêu cầu mới.

- Phù hợp cho các dự án ràng buộc về mặt thời gian. Các nhóm phát triển có thể làm việc song song trên các plugin khác nhau, rút ngắn thời gian hoàn thành. Việc thay đổi hoặc nâng cấp một plugin không ảnh hưởng đến hoạt động của các plugin khác.

- Tăng khả năng mở rộng, linh hoạt và kiểm thử: Mỗi chức năng như quản lý môn học, kiểm tra điều kiện tiên quyết, đăng ký lớp học, và gửi email thông báo có thể được triển khai dưới dạng các plugin riêng biệt. Điều này không chỉ giúp hệ thống dễ dàng mở rộng khi có nhu cầu mới mà còn giúp việc kiểm thử từng chức năng trở nên đơn giản hơn. Ví dụ, việc kiểm tra và xác nhận đăng ký có thể được kiểm tra độc lập mà không ảnh hưởng đến các phần khác của hệ thống.

- Việc kiểm tra được đơn giản hóa: Mỗi plugin, chẳng hạn như plugin quản lý đăng ký học phần, có thể được kiểm tra riêng lẻ. Điều này giúp phát hiện lỗi sớm hơn và dễ dàng hơn trong quá trình phát triển, giảm thiểu rủi ro gây ra lỗi toàn hệ thống.

- Tương đối tốt về mặt hiệu năng: Với kiến trúc microkernel, chỉ các plugin cần thiết mới được tải vào hệ thống tại một thời điểm nhất định. Ví dụ, plugin liên quan đến đăng ký học phần sẽ chỉ hoạt động mạnh trong giai đoạn mở đăng ký, trong khi các plugin khác có thể được giảm tải hoặc tắt để tiết kiệm tài nguyên. Điều này giúp tối ưu hóa hiệu năng của hệ thống và đảm bảo rằng nó có thể xử lý tải lớn trong giai đoạn đăng ký.

* + 1. **Nhược điểm**

- Giao tiếp giữa các plugin: Do kiến trúc microkernel, các plugin cần giao tiếp với nhau thông qua microkernel để trao đổi dữ liệu. Điều này có thể gây ra độ trễ nhất định, đặc biệt khi có nhiều plugin và lượng dữ liệu lớn. Ví dụ, khi sinh viên đăng ký môn học, plugin đăng ký phải liên hệ với plugin quản lý môn học để kiểm tra tiên quyết, rồi liên hệ với plugin thông tin sinh viên để kiểm tra tín chỉ. Quá trình này có thể chậm hơn so với kiến trúc monolithic, nơi tất cả logic nằm trong một khối duy nhất.

- Quản lý nhiều tiến trình: Mỗi plugin chạy như một tiến trình riêng biệt, đòi hỏi hệ điều hành quản lý nhiều tiến trình hơn. Điều này có thể gây áp lực lên tài nguyên hệ thống, đặc biệt khi có nhiều sinh viên sử dụng hệ thống đồng thời.

- Phát triển và bảo trì: Việc chia nhỏ hệ thống thành nhiều plugin đòi hỏi sự phối hợp chặt chẽ giữa các nhóm phát triển. Cần có quy ước rõ ràng về giao diện, định dạng dữ liệu trao đổi giữa các plugin. Việc bảo trì và nâng cấp cũng phức tạp hơn, vì phải đảm bảo tính tương thích giữa các phiên bản khác nhau của các plugin.

- Gỡ lỗi và xử lý lỗi: Khi xảy ra lỗi, việc xác định nguyên nhân và vị trí lỗi có thể khó khăn hơn. Lỗi có thể nằm ở microkernel, ở một plugin cụ thể, hoặc do lỗi giao tiếp giữa các plugin.

- Phụ thuộc vào microkernel: Microkernel đóng vai trò trung tâm trong việc điều phối và giao tiếp giữa các plugin. Nếu microkernel gặp sự cố hoặc không được thiết kế để mở rộng, toàn bộ hệ thống có thể bị ảnh hưởng.

- Giới hạn về kiến trúc: Mặc dù microkernel cho phép thêm bớt plugin dễ dàng, nhưng nếu không thiết kế cẩn thận, kiến trúc có thể trở nên phức tạp và khó quản lý khi số lượng plugin tăng lên.

- Đồng bộ dữ liệu: Trong hệ thống quản lý đăng ký học phần, việc đồng bộ dữ liệu giữa các plugin là rất quan trọng. Nếu không được thực hiện tốt, có thể dẫn đến sự không nhất quán dữ liệu, ví dụ như số lượng sinh viên đã đăng ký vượt quá giới hạn do lỗi đồng bộ giữa plugin đăng ký và plugin quản lý lớp học.

- Độ tin cậy của hệ thống: Nếu một plugin quan trọng bị lỗi, nó có thể ảnh hưởng đến toàn bộ hệ thống. Ví dụ, nếu plugin kiểm tra điều kiện tiên quyết bị lỗi, sinh viên có thể đăng ký vào các môn học mà họ không đủ điều kiện, gây ra sự cố trong hệ thống.

- Chi phí ban đầu: Mặc dù microkernel có thể hiệu quả về chi phí dài hạn, chi phí phát triển ban đầu để tạo ra một kiến trúc module hoá với nhiều plugin có thể cao hơn so với phát triển một hệ thống monolithic.

- Đào tạo và chuyển giao công nghệ: Đội ngũ phát triển và vận hành cần được đào tạo để làm quen với kiến trúc microkernel và cách phát triển, triển khai và bảo trì các plugin, điều này cũng có thể tăng chi phí và thời gian ban đầu.

* 1. **Service-Based Architecture Style**
     1. **Ưu điểm**

- Thường sử dụng CSDL chia sẽ tập trung. Cho phép các service tận dụng các câu lệnh truy vấn và join tương tự (Một custom shared library được sử dụng bởi tất cả domain services chẳng hạn như: JAR file or DDL, Shared library có thể chứa cả SQL Code) như kiến trúc monolithic, do số lượng dịch vụ từ (4-12, tốt nhất là 7) nên kết nối Database không phải là vấn đề trong Service-based

- Do Domain services là chia thô (Coarse-grained) nên các transaction trong database ACID liên quan đến các commit và rollback giúp đảm bảo tính toàn vẹn dữ liệu trong 1 domain service đơn. Chia thô (Coarse-grained) cho phép tích hợp và nhất quán về dữ liệu tốt hơn

- Những thay đổi ở domain chỉ ảnh hưởng đến service cụ thể, giao diện và cơ sở dữ liệu tương ứng.

- Việc chia ứng dụng thành các domain serivce được triển khai một cách riêng biệt cho phép thay đổi nhanh hơn, phạm vi kiểm thử tốt hơn do giới hạn về phạm vi của domain, khả năng deploy thường xuyên hơn và ít rủi ro hơn monolithic (Deployability). 3 tính chất trên (Agility, testability, deployability) giúp thời gian đưa sản phẩm ra thị trường nhanh hơn, cho phép tổ chức bàn giao chức năng mới và sửa lỗi với tốc độ tương đối cao

* + 1. **Nhược điểm**

- Nếu sử dụng CSDL chia sẻ tập trung, những thay đổi về Database có thể là một vấn đề (thay đổi trong lược đồ các bảng của CSDL). Nếu không thay đổi một cách chính xác thì có thể gây ảnh hưởng đến mọi service. Thực hiện thay đổi database là 1 nhiệm vụ rất tốn kém về mặt nổ lực và phối hợp. Sử dụng Shared library versioning có thể giải quyết vấn đề thay đổi trong CSDL

- Việc domain services là chia thô (Coarse-grained), một thay đổi được thực hiện đối với chức năng trong service sẽ yêu cầu kiểm thử toàn bộ service (1 thay đổi nhỏ trong Order Service – thay đổi function Order placement có thể yêu cầu kiểm thử ở cả quy trình thanh toán), trong khi microservice cùng thay đổi đó sẽ chỉ ảnh hưởng đến Order placement không yêu cầu thay đổi và kiểm thử ở Payment Service

* 1. **Service-Oriented Architecture Style**
     1. **Ưu điểm**

- Khả năng tái sử dụng và phát triển các phần mềm đang hiện hữu mà không cần đổi mới. Ví dụ, khi có sự thay đổi trong quy định về môn học tiên quyết hoặc số tín chỉ tối đa, chỉ cần cập nhật Course Management Service hoặc Enrollment Service mà không cần thay đổi toàn bộ hệ thống.

- Các service có thể giao tiếp với nhau trên nhiều nền tảng và ngôn ngữ khác nhau. Dịch vụ quản lý đăng ký học phần (Course Registration Service) có thể được xây dựng bằng Java, trong khi dịch vụ quản lý thông tin sinh viên (Student Information Service) được xây dựng bằng Python. Hai dịch vụ này vẫn có thể giao tiếp với nhau thông qua các giao thức chuẩn như RESTful API hoặc SOAP.

- Tự hồi phục sau khi bị lỗi mà không cần sự can thiệp của con người. Nếu dịch vụ gửi email xác nhận đăng ký gặp sự cố, hệ thống có thể tự động khởi động lại dịch vụ này mà không cần can thiệp thủ công, đảm bảo email thông báo vẫn được gửi đi.

- Nếu một thể hiện service nào đó không hoạt động thì một thể hiện khác vẫn có thể hoàn tất giao dịch cho khách hàng mà không bị ảnh hưởng gì. Trong thời gian cao điểm, khi số lượng sinh viên đăng ký học phần tăng đột biến, hệ thống có thể tự động khởi chạy thêm nhiều phiên bản của dịch vụ quản lý đăng ký học phần (Course Registration Service) để xử lý các yêu cầu đăng ký, đảm bảo hệ thống không bị quá tải.

- Tính loose coupling giúp tăng tính linh hoạt và khả năng triển khai cài đặt. Các dịch vụ như quản lý khóa học, quản lý sinh viên và dịch vụ xác nhận đăng ký có thể được phát triển, triển khai và cập nhật độc lập với nhau, không ảnh hưởng đến toàn bộ hệ thống.

- Tăng khả năng mở rộng bằng cách thêm nhiều thể hiện (instance) của một service. Khi số lượng sinh viên tăng lên, trường đại học có thể dễ dàng mở rộng hệ thống bằng cách thêm nhiều máy chủ chạy dịch vụ quản lý đăng ký học phần (Course Registration Service) để đáp ứng nhu cầu đăng ký học phần ngày càng tăng.

- Có thể chuyển tiếp nội dung yêu cầu đến một thể hiện khác khi cần, nhờ đó tăng khả năng sẵn sàng. Khi một máy chủ chạy dịch vụ quản lý đăng ký học phần (Course Registration Service) bị quá tải, hệ thống có thể tự động chuyển hướng yêu cầu đăng ký đến một máy chủ khác có tải nhẹ hơn, đảm bảo hệ thống luôn hoạt động ổn định.

- Vì mỗi dịch vụ phần mềm là một đơn vị độc lập, thật dễ dàng để cập nhật và bảo trì nó mà không làm ảnh hưởng đến các dịch vụ khác. Khi cần nâng cấp hệ thống quản lý sinh viên để bổ sung tính năng mới, các dịch vụ khác như quản lý khóa học hoặc hệ thống gửi email không bị ảnh hưởng.

- Các services dễ debug và test hơn là các đoạn code lớn như kiến trúc monolithic. Các dịch vụ như quản lý thời khóa biểu, đăng ký lớp học, và gửi thông báo có thể được kiểm tra riêng biệt, giúp dễ dàng phát hiện và sửa lỗi.

- Các dịch vụ độc lập có thể được phát triển song song và hoàn thành cùng một lúc. Nhóm phát triển dịch vụ quản lý đăng ký học phần (Course Registration Service) có thể làm việc độc lập với nhóm phát triển dịch vụ quản lý thông tin sinh viên (Student Information Service), giúp rút ngắn thời gian phát triển hệ thống.

- Tiết kiệm thời gian làm việc và tăng năng suất cho nhân sự. Sử dụng kiến trúc SOA cho phép nhân sự làm việc hiệu quả hơn do các dịch vụ được phát triển, kiểm tra và bảo trì độc lập, giảm thiểu thời gian quản lý hệ thống tổng thể.

- Hạn chế tối đa những yêu cầu về kỹ năng và đào tạo nhân viên mỗi khi phát triển phần mềm mới. Khi cần phát triển một dịch vụ mới, các nhà phát triển chỉ cần tập trung vào các công nghệ và kiến thức liên quan đến dịch vụ đó, mà không cần phải nắm rõ toàn bộ hệ thống.

- Quy trình phát triển, quản lý kinh doanh đơn giản, hiệu quả và nhanh chóng hơn. Nhờ tính linh hoạt và khả năng mở rộng của kiến trúc SOA, trường đại học có thể dễ dàng thêm mới hoặc cập nhật các tính năng của hệ thống đăng ký học phần một cách nhanh chóng và hiệu quả, đáp ứng tốt hơn nhu cầu của sinh viên và nhà trường.

* + 1. **Nhược điểm**

- Quản lý tin nhắn phức tạp: Trong ngày đầu mở đăng ký, hàng ngàn sinh viên đồng loạt truy cập hệ thống để đăng ký các môn học. Điều này tạo ra một lượng lớn yêu cầu giữa các dịch vụ (dịch vụ xác thực, dịch vụ kiểm tra điều kiện tiên quyết, dịch vụ đăng ký học phần, dịch vụ gửi email thông báo). Việc đảm bảo tất cả các tin nhắn được gửi và xử lý kịp thời, không bị mất mát hay trùng lặp, trở thành một thách thức lớn.

- Đầu tư lớn về nguồn lực: Để xây dựng hệ thống đăng ký học phần theo kiến trúc SOA, trường đại học cần đầu tư vào đội ngũ phát triển có kinh nghiệm với kiến thức về SOA, các công nghệ tích hợp, quản lý API, cũng như các công cụ và hạ tầng hỗ trợ SOA.

- Giảm hiệu suất do xác nhận đầu vào: Khi sinh viên đăng ký một môn học, dịch vụ đăng ký học phần cần xác nhận đầu vào từ nhiều dịch vụ khác như dịch vụ kiểm tra điều kiện tiên quyết, dịch vụ kiểm tra số tín chỉ đã đăng ký, dịch vụ kiểm tra số lượng sinh viên tối đa trong lớp. Việc này có thể làm tăng thời gian phản hồi và giảm hiệu suất tổng thể của hệ thống, đặc biệt là trong thời gian cao điểm.

- Lỗ hổng bảo mật: Các dịch vụ trong hệ thống đăng ký học phần giao tiếp với nhau thông qua các API. Nếu các API này không được bảo mật đúng cách, tin tặc có thể khai thác các lỗ hổng để truy cập trái phép vào dữ liệu nhạy cảm của sinh viên hoặc thực hiện các hành vi phá hoại hệ thống.

- Khó quản lý: Hệ thống đăng ký học phần có thể bao gồm hàng chục dịch vụ khác nhau, mỗi dịch vụ lại có nhiều phiên bản và cấu hình khác nhau. Việc quản lý, giám sát và đảm bảo tính ổn định của toàn bộ hệ thống trở nên phức tạp và tốn kém.

- Khả năng điều chỉnh quy mô của hệ thống bị ảnh hưởng đáng kể khi các dịch vụ chia sẻ nhiều tài nguyên và cần phối hợp để thực hiện chức năng của mình

- Có thể trở nên phức tạp hơn theo thời gian và gây ra sự phụ thuộc lẫn nhau giữa các dịch vụ. Dịch vụ đăng ký học phần phụ thuộc vào dịch vụ kiểm tra điều kiện tiên quyết, dịch vụ kiểm tra số tín chỉ đã đăng ký, dịch vụ kiểm tra số lượng sinh viên tối đa trong lớp. Nếu một trong các dịch vụ này gặp sự cố, dịch vụ đăng ký học phần cũng sẽ bị ảnh hưởng và không thể hoạt động bình thường.

* 1. **Event-Driven Architecture Style**
     1. **Ưu điểm**
* Khả năng chịu lỗi tốt: Nếu hệ thống gửi email gặp sự cố, các sự kiện khác (như cập nhật danh sách chờ, thông báo đóng học phí) vẫn được xử lý bình thường.
* Giảm sự phụ thuộc lẫn nhau giữa producer và consumer giúp dễ dàng thêm hoặc bớt consumers và producers. Khi có yêu cầu thêm tính năng mới (ví dụ: tích hợp với hệ thống thư viện), ta chỉ cần thêm một Event Consumer mới mà không cần thay đổi các thành phần khác.
* Xử lý dữ liệu theo thời gian thực bằng cách phát hiện và phản ứng nhanh chóng với các sự kiện mới. Khi sinh viên đăng ký môn học, hệ thống sẽ nhanh chóng phát hiện và xử lý các sự kiện đăng ký, cập nhật danh sách các lớp còn chỗ trống và danh sách dự bị theo thời gian thực. Sinh viên sẽ nhận được thông báo ngay lập tức về trạng thái đăng ký của mình, giúp họ có thể điều chỉnh lựa chọn kịp thời nếu cần.
* Cung cấp một lịch sử hoàn chỉnh: Mỗi sự kiện đăng ký, xác nhận, và hủy bỏ đăng ký đều được ghi lại chi tiết. Điều này không chỉ giúp ích trong việc phục hồi trạng thái khi có sự cố mà còn hỗ trợ việc phân tích dữ liệu sau này, như phân tích xu hướng đăng ký của sinh viên hoặc xác định các vấn đề trong quy trình đăng ký.
* Mỗi dịch vụ có thể được mở rộng và nâng cấp một cách độc lập để phù hợp nhất với nhu cầu. Ví dụ, khi lượng sinh viên tăng lên, dịch vụ đăng ký học phần có thể được mở rộng để xử lý nhiều yêu cầu đồng thời hơn mà không ảnh hưởng đến các dịch vụ khác.
  + 1. **Nhược điểm**
* Xử lý luồng sự kiện phức tạp: Khi sinh viên đăng ký một môn học, hệ thống phải xử lý nhiều sự kiện liên quan như: kiểm tra điều kiện tiên quyết, kiểm tra số tín chỉ tối đa, kiểm tra số lượng sinh viên trong lớp, cập nhật thông tin đăng ký, gửi email xác nhận, và có thể thêm sinh viên vào danh sách chờ. Việc quản lý và xử lý đồng thời các sự kiện này đòi hỏi kỹ năng cao và có thể gây ra lỗi nếu không được thực hiện cẩn thận.
* Đảm bảo tính nhất quán và đúng đắn: Nếu hai sinh viên cùng đăng ký vào một lớp chỉ còn một vị trí trống trong cùng một thời điểm, hệ thống cần đảm bảo chỉ một sinh viên được đăng ký và sinh viên còn lại được đưa vào danh sách chờ. Việc xử lý đồng thời và đảm bảo tính nhất quán của dữ liệu trong trường hợp này là một thách thức lớn.
* Sử dụng message broker có thể tạo ra bottleneck: Vào ngày mở đăng ký, hàng ngàn sinh viên có thể đăng ký cùng lúc. Nếu hệ thống message broker không đủ mạnh để xử lý lượng lớn sự kiện này, nó có thể trở thành điểm nghẽn, làm chậm quá trình đăng ký và gây ra trải nghiệm người dùng kém.
* Đảm bảo tính nhất quán dữ liệu giữa các microservices: Hệ thống có thể có nhiều microservices riêng biệt để quản lý thông tin sinh viên, môn học, lớp học, đăng ký, thanh toán, v.v. Khi sinh viên đăng ký một môn học, thông tin này cần được cập nhật đồng bộ trên tất cả các microservices liên quan. Việc đảm bảo tính nhất quán dữ liệu trong môi trường phân tán này là một vấn đề phức tạp.
* Độ trễ trong xử lý sự kiện: Khi sinh viên đăng ký một môn học, hệ thống cần kiểm tra nhiều điều kiện trước khi xác nhận đăng ký. Nếu các sự kiện này phải được truyền qua message broker và xử lý bởi nhiều microservices khác nhau, có thể có độ trễ đáng kể so với việc xử lý trực tiếp trên một hệ thống tập trung.
* Độ phức tạp của logging và monitoring: Với hàng trăm sự kiện khác nhau xảy ra trong hệ thống (đăng ký, hủy đăng ký, thanh toán, xét tốt nghiệp, v.v.), việc ghi log và theo dõi tất cả các sự kiện này để phát hiện và khắc phục lỗi có thể trở nên cực kỳ phức tạp và tốn kém.
  1. **Space-Based Architecture Style**
     1. **Ưu điểm**

- Giải pháp tốt xử lý vấn đề high scalability, elasticity và high concurrency. Nó cũng là lựa chọn tốt cho ứng dụng có nhiều biến số và không thể dự đoán được số lượng người dùng đồng thời.

- Khả năng mở rộng cao, linh hoạt cao và hiệu suất cao bằng cách loại bỏ cơ sở dữ liệu trung tâm như là ràng buộc đồng bộ trong hệ thống và thay vào đó tận dụng replicated các lưới dữ liệu trong bộ nhớ

- Không có dữ liệu trung tâm tham gia vào xử lý các transactional tiêu chuẩn của ứng dụng, việc ngẽn cổ chai trong CSDL được loại bỏ, do đó mang lại khả năng mở rộng (Scalability) gần như là vô hạn trong ứng dụng. Giúp có thể xử lý được hàng triệu người dùng đồng thời

* + 1. **Nhược điểm**

- Rất phức tạp do sử dụng caching và tính nhất quán cuối của kho lưu trữ dữ liệu chính (primary data store). Phải cẩn thận để đảm bảo không có dữ liệu nào bị mất trong trường hợp xảy ra sự cố

- Kiểm thử khá phức tạp do liên quan đến việc mô phỏng scalability và elastricity. Việc kiểm thử với hàng trăm ngàn người dùng đồng thời ở mức độ cao rất là phức tạp và tốn kém

- Kiến trúc này khá tốn kém, chủ yếu đến từ các sản phẩm caching, mức sử dụng tài nguyên cao trong các hệ thống đám mây

* 1. **Orchestration-Driven Service-Oriented Architecture**
     1. **Ưu điểm**
* Quản lý quy trình phức tạp: ODS cho phép định nghĩa và quản lý một cách linh hoạt các quy trình đăng ký học phần phức tạp, bao gồm các bước kiểm tra điều kiện tiên quyết, giới hạn tín chỉ, xác nhận đăng ký, xử lý danh sách chờ, gửi thông báo, v.v.
* Dễ dàng mở rộng và thay đổi: Khi có thay đổi trong quy trình đăng ký (ví dụ: thay đổi quy định về tín chỉ, thêm tính năng mới), có thể dễ dàng cập nhật quy trình orchestration mà không cần thay đổi nhiều trong các dịch vụ riêng lẻ.
* Tối ưu hóa tài nguyên: Các dịch vụ như "Kiểm tra điều kiện tiên quyết", "Tính toán tín chỉ", "Gửi email thông báo" có thể được sử dụng lại trong nhiều quy trình khác nhau, giúp tiết kiệm thời gian và công sức phát triển.
* Đơn giản hóa việc bảo trì: Việc cập nhật hoặc sửa lỗi trong một dịch vụ sẽ tự động áp dụng cho tất cả các quy trình sử dụng dịch vụ đó. Ví dụ, khi có sự thay đổi trong quy định môn học tiên quyết (ví dụ như thêm hoặc bỏ môn tiên quyết), việc cập nhật dịch vụ kiểm tra này sẽ tự động áp dụng cho tất cả các quy trình đăng ký học phần. Điều này giúp đảm bảo rằng tất cả các sinh viên đều phải tuân thủ quy định mới mà không cần thay đổi mã nguồn trong nhiều nơi khác nhau của hệ thống.
* Theo dõi trạng thái: ODS cho phép theo dõi trạng thái của từng yêu cầu đăng ký, giúp dễ dàng xác định các vấn đề và xử lý các trường hợp ngoại lệ.
* Báo cáo và phân tích: Dữ liệu về số lượng đăng ký, tỷ lệ thành công, thời gian xử lý, v.v. có thể được thu thập để phân tích và cải thiện hiệu suất của hệ thống.
* Xử lý lỗi: ODS có thể được thiết kế để tự động xử lý các lỗi xảy ra trong quá trình đăng ký, ví dụ: gửi lại yêu cầu, thông báo cho người dùng, hoặc chuyển yêu cầu sang quy trình xử lý lỗi riêng biệt.
* Khả năng mở rộng: Vào ngày mở đăng ký, hệ thống sẽ phải xử lý lượng truy cập lớn từ các sinh viên đăng nhập để đăng ký môn học. Kiến trúc ODS cho phép dễ dàng mở rộng hệ thống bằng cách triển khai nhiều phiên bản của dịch vụ đăng ký trên nhiều máy chủ khác nhau. Khi nhu cầu tăng cao, trường đại học có thể thêm các máy chủ mới để đảm bảo hệ thống vẫn hoạt động mượt mà và không bị quá tải.
* Phù hợp với kiến trúc microservices: Hệ thống quản lý đăng ký học phần có thể được chia thành nhiều microservices nhỏ, độc lập như dịch vụ đăng ký môn học, dịch vụ kiểm tra điều kiện tiên quyết, dịch vụ gửi email thông báo, dịch vụ quản lý thông tin học tập, và dịch vụ xét tốt nghiệp. Mỗi microservice sẽ có nhiệm vụ riêng và có thể được phát triển, triển khai và bảo trì một cách độc lập. Điều này không chỉ giúp quản lý hệ thống dễ dàng hơn mà còn đảm bảo rằng các phần của hệ thống có thể được cập nhật mà không ảnh hưởng đến toàn bộ hệ thống.
* Tích hợp với các hệ thống khác: Hệ thống ODS dễ dàng tích hợp với các hệ thống bên ngoài như hệ thống quản lý email, cổng thanh toán học phí, hệ thống quản lý nhân sự và cựu sinh viên. Điều này giúp tạo ra một hệ sinh thái liên kết chặt chẽ, cung cấp trải nghiệm liền mạch cho người dùng.
* Quản lý quy trình nghiệp vụ phức tạp: Với ODS, nhà trường có thể dễ dàng thiết lập các quy trình nghiệp vụ phức tạp như quản lý danh sách chờ, xét duyệt mở thêm lớp học hay quy trình xét tốt nghiệp. Các quy trình này có thể được định nghĩa và quản lý thông qua các công cụ orchestration, giúp đảm bảo tính nhất quán và minh bạch.
* Cá nhân hoá trải nghiệm người dùng: ODS cho phép tích hợp các dịch vụ phân tích và cá nhân hóa, giúp cung cấp cho sinh viên trải nghiệm tốt hơn. Ví dụ, sinh viên có thể nhận được gợi ý các môn học phù hợp dựa trên lịch sử học tập và ngành học của mình.
* Bảo mật và quản lý dữ liệu: Kiến trúc ODS cho phép quản lý bảo mật và quyền truy cập ở mức dịch vụ, đảm bảo rằng chỉ những người dùng được ủy quyền mới có thể truy cập vào các dịch vụ và dữ liệu nhạy cảm. Điều này giúp bảo vệ thông tin cá nhân và học tập của sinh viên.
* Quản lý và giảm sát hiệu quả: Với các công cụ orchestration, nhà trường có thể giám sát và quản lý hiệu quả toàn bộ hệ thống, từ việc theo dõi tình trạng đăng ký môn học đến việc giám sát tình trạng gửi email xác nhận và quản lý danh sách chờ. Điều này giúp phát hiện và giải quyết các vấn đề kịp thời, đảm bảo hệ thống hoạt động trơn tru.
  + 1. Nhược điểm
* Độ phức tạp cao: ODS đòi hỏi sự đầu tư lớn về công nghệ và nhân lực để thiết kế, triển khai và duy trì. Việc xây dựng các dịch vụ nhỏ gọn, độc lập và phối hợp chúng một cách hiệu quả yêu cầu kỹ năng và kinh nghiệm cao. Đây có thể là thách thức lớn đối với các tổ chức không có đội ngũ kỹ thuật mạnh.
* Hiệu suất và độ trễ: Mỗi lần một dịch vụ được gọi thông qua tầng orchestration, sẽ có độ trễ nhất định do quá trình giao tiếp giữa các dịch vụ. Trong trường hợp hệ thống đăng ký học phần, nơi mà hàng ngàn sinh viên có thể đăng nhập cùng lúc, điều này có thể dẫn đến tình trạng chậm trễ hoặc tắc nghẽn hệ thống.
* Khó khăn trong việc đảm bảo tính nhất quán: Khi dữ liệu được phân tán qua nhiều dịch vụ, việc đảm bảo tính nhất quán của dữ liệu trở nên phức tạp. Ví dụ, nếu có sự cố xảy ra trong quá trình đăng ký học phần, việc khôi phục trạng thái nhất quán của hệ thống có thể gặp nhiều khó khăn.
* Quản lý lỗi và xử lý ngoại lệ: Việc quản lý lỗi và xử lý ngoại lệ trong một hệ thống ODS đòi hỏi phải có cơ chế theo dõi và khôi phục mạnh mẽ. Nếu một dịch vụ gặp sự cố, tầng orchestration cần phải biết cách xử lý và khôi phục từ lỗi đó, điều này có thể phức tạp và đòi hỏi nhiều tài nguyên.
* Chi phí phát triển và vận hành cao: Việc phát triển và duy trì một hệ thống ODS đòi hỏi đầu tư lớn về tài chính, cả trong việc phát triển ban đầu và vận hành lâu dài. Nhà trường cần có đội ngũ chuyên gia có kinh nghiệm để phát triển và bảo trì hệ thống này.
* Khả năng mở rộng hạn chế bởi tầng orchestration: Tầng orchestration có thể trở thành nút thắt cổ chai khi hệ thống cần mở rộng quy mô. Khi số lượng dịch vụ và lượng người dùng tăng lên, việc điều phối và quản lý các dịch vụ có thể trở nên khó khăn và kém hiệu quả.
* Phụ thuộc vào kết nối mạng: ODS hoạt động dựa trên việc các dịch vụ liên tục giao tiếp với nhau qua mạng. Nếu kết nối mạng bị gián đoạn hoặc có vấn đề, hệ thống có thể bị ảnh hưởng nghiêm trọng. Điều này đặc biệt quan trọng trong hệ thống đăng ký học phần, nơi mà thời gian phản hồi nhanh chóng là cần thiết.
* Khó khăn trong việc bảo mật và quản lý quyền truy cập: Mỗi dịch vụ trong hệ thống ODS cần được bảo mật và quản lý quyền truy cập một cách cẩn thận. Việc quản lý các chính sách bảo mật và quyền truy cập cho nhiều dịch vụ phân tán có thể trở nên phức tạp và dễ gây ra lỗi bảo mật.
* Khó khăn trong việc theo dõi và giám sát toàn diện: Mặc dù có các công cụ hỗ trợ giám sát và theo dõi, việc theo dõi toàn diện và chi tiết toàn bộ hệ thống ODS có thể phức tạp. Để phát hiện và xử lý kịp thời các vấn đề, nhà trường cần đầu tư vào các giải pháp giám sát và phân tích mạnh mẽ.
* Phụ thuộc vào orchestrator: Orchestrator đóng vai trò trung tâm trong việc điều phối các dịch vụ, nếu orchestrator gặp sự cố, toàn bộ hệ thống có thể bị ảnh hưởng. Việc thay đổi hoặc nâng cấp orchestrator có thể yêu cầu phải dừng hoạt động của hệ thống.
* Khó khăn trong việc kiểm thử: Việc kiểm thử toàn bộ hệ thống ODS có thể phức tạp do sự tương tác giữa nhiều dịch vụ và orchestrator.
  1. **Microservices Architecture**
     1. **Ưu điểm**
* Sự linh hoạt trong lựa chọn công nghệ: Trường đại học có thể sử dụng Python và Django cho microservice quản lý thông tin sinh viên, Node.js và Express cho microservice xử lý đăng ký môn học, và Java và Spring Boot cho microservice tính toán tín chỉ. Sự kết hợp này cho phép tận dụng điểm mạnh của từng ngôn ngữ và framework, đồng thời tạo điều kiện cho việc kiểm thử và áp dụng các công nghệ mới một cách dễ dàng.
* Tính độc lập cao: Microservice quản lý thông tin sinh viên có thể được phát triển, triển khai và mở rộng độc lập với microservice xử lý đăng ký môn học. Điều này giúp giảm thiểu sự phụ thuộc lẫn nhau, tăng tính linh hoạt và khả năng đáp ứng với các thay đổi. Ví dụ, khi số lượng sinh viên tăng đột biến, trường đại học có thể dễ dàng mở rộng microservice xử lý đăng ký mà không ảnh hưởng đến các microservice khác.
* Dễ dàng bảo trì và cập nhật: Việc cập nhật hoặc nâng cấp một phần của hệ thống, chẳng hạn như cập nhật quy trình đăng ký hay thêm tính năng mới vào quản lý tín chỉ, có thể được thực hiện mà không cần ảnh hưởng đến các dịch vụ khác.
* Tăng tốc quá trình phát triển: Bằng cách chia nhỏ hệ thống thành các dịch vụ nhỏ, quá trình phát triển có thể diễn ra song song, giúp tăng hiệu suất và giảm thời gian hoàn thành các tính năng mới.
* Khả năng chịu lỗi cao: Mỗi dịch vụ là một phần độc lập, do đó, khi có sự cố xảy ra trong một dịch vụ, chỉ ảnh hưởng đến phần đó mà không làm ngừng hoạt động toàn bộ hệ thống. Ví dụ, nếu hệ thống đăng ký học phần gặp sự cố, các tính năng khác như quản lý tín chỉ vẫn hoạt động bình thường.
* Scale dễ dàng: Các dịch vụ có thể được scale riêng biệt theo nhu cầu. Ví dụ, dịch vụ quản lý đăng ký có thể được scale lên nhiều bản sao trong thời gian đăng ký nhiều sinh viên, trong khi dịch vụ quản lý thông tin học tập có thể được scale theo nhu cầu truy cập thông tin của sinh viên.
* Triển khai độc lập: Trường đại học có thể triển khai một phiên bản mới của microservice quản lý thông tin sinh viên mà không cần phải triển khai lại toàn bộ hệ thống. Điều này giúp giảm thiểu rủi ro và đảm bảo tính ổn định của hệ thống.
  + 1. **Nhược điểm**
* Quản lý nhiều database và transactions: Hệ thống quản lý đăng ký học phần cần đảm bảo rằng các giao dịch như đăng ký môn học, cập nhật tín chỉ, và xác nhận đăng ký được thực hiện một cách nhất quán và an toàn. Ví dụ khi một sinh viên đăng ký môn học, cần kiểm tra xem sinh viên đã thỏa mãn các môn tiên quyết, còn đủ tín chỉ để đăng ký hay không, và cập nhật danh sách sinh viên đăng ký vào lớp. Những thao tác này thường yêu cầu tương tác với nhiều dịch vụ và cơ sở dữ liệu khác nhau (như quản lý sinh viên, quản lý môn học). Đảm bảo tính nhất quán và an toàn dữ liệu trong các giao dịch phân tán này là một thách thức lớn
* Độ trễ mạng: Giao tiếp giữa các dịch vụ qua mạng có thể làm tăng độ trễ, ảnh hưởng đến trải nghiệm của người dùng. Ví dụ, khi sinh viên đăng ký học phần, hệ thống phải liên lạc với nhiều dịch vụ để hiển thị danh sách môn học, các môn học đã đăng ký, và trạng thái lớp học. Nếu có độ trễ mạng đáng kể, sinh viên có thể gặp phải trải nghiệm không mượt mà, mất thời gian chờ đợi lâu hoặc thậm chí bị timeout.
* Khó khăn trong việc đảm bảo tính toàn vẹn dữ liệu: Nếu microservice xử lý đăng ký môn học gặp sự cố trong quá trình cập nhật dữ liệu, có thể dẫn đến tình trạng dữ liệu không nhất quán, ví dụ như một sinh viên được đăng ký vào một lớp học đã đầy. Việc đảm bảo tính toàn vẹn dữ liệu trong một hệ thống phân tán như microservice là một bài toán phức tạp, đòi hỏi các kỹ thuật và công cụ đặc biệt.
* Testing phức tạp: Để kiểm thử quy trình đăng ký môn học, cần phải mô phỏng nhiều dịch vụ khác nhau (quản lý sinh viên, quản lý môn học, hệ thống email). Điều này đòi hỏi một môi trường kiểm thử phức tạp và gần như tương tự môi trường sản xuất, gây tốn kém thời gian và chi phí.
* Phối hợp giữa các team: Trong một hệ thống phức tạp với nhiều microservices, việc phối hợp giữa các team phát triển tương ứng với mỗi microservice là một thách thức, đòi hỏi sự giao tiếp và quản lý dự án hiệu quả. Ví dụ, nếu team phát triển microservice quản lý thông tin sinh viên thay đổi cấu trúc dữ liệu của sinh viên mà không thông báo cho team phát triển microservice xử lý đăng ký môn học, có thể dẫn đến lỗi trong quá trình đăng ký

1. **Select appropriate software architecture style**

**3.0 Lựa chọn**

- Chọn kiến trúc: Event Microservice

* 1. **Vì sao chọn?**
     1. **Phù hợp với tính phức tạp của hệ thống**

- Hệ thống đăng ký học phần có nhiều nghiệp vụ phức tạp như quản lý sinh viên, môn học, lớp học, đăng ký, thanh toán, tốt nghiệp, cựu sinh viên... Microservice cho phép chia nhỏ hệ thống thành các dịch vụ độc lập, mỗi dịch vụ tập trung vào một nghiệp vụ cụ thể, giúp dễ dàng phát triển, bảo trì và mở rộng.

- Các microservice giao tiếp với nhau thông qua messaging, giảm sự phụ thuộc giữa các thành phần. Điều này giúp việc thay đổi hoặc cập nhật một dịch vụ không ảnh hưởng đến các dịch vụ khác.

- Ngoài ra kiến trúc evendriven microservice có thể khắc phục tình trạng mất mát dữ liệu khi các service giao tiếp với nhau, khi giữ 2 service giao tiêp với nhau nếu 1 trong 2 bị mất kết nối vì trường hợp nào đó thì với kiên trúc thuần microservice sẽ làm mát đi dữ liệu nếu không có biện pháp khắc phục vần đề này, nhưng với kiên trúc even driven nêu tự tin rằng service bị ngắt kết nối sẽ được khôi phục lại nhanh chóng thì kiên trúc này phù hợp hơn

* + 1. **Khả năng mở rộng linh hoạt**

- Trong thời gian đăng ký, hệ thống cần xử lý lượng lớn yêu cầu đăng ký đồng thời. Microservice cho phép mở rộng từng dịch vụ độc lập, đáp ứng nhu cầu tăng đột biến mà không cần mở rộng toàn bộ hệ thống. Điều này rất quan trọng đối với một hệ thống như quản lý đăng ký học phần, nơi có thể có các đợt cao điểm (như khi mở đăng ký học phần mỗi học kỳ). Nếu cần, các dịch vụ như đăng ký môn học có thể được mở rộng nhanh chóng để đáp ứng lượng lớn yêu cầu từ sinh viên.

- Mỗi microservice có thể được phát triển và triển khai bằng công nghệ phù hợp nhất với yêu cầu của dịch vụ đó, giúp tối ưu hóa việc sử dụng tài nguyên. Ví dụ, dịch vụ gửi email có thể sử dụng một công nghệ khác so với dịch vụ quản lý cơ sở dữ liệu học tập. Điều này cho phép tận dụng các công nghệ hiện đại và tối ưu hóa hiệu năng của từng phần của hệ thống.

* + 1. **Tăng tốc độ phát triển**

- Các team phát triển có thể làm việc song song trên các microservice khác nhau, giúp tăng tốc độ phát triển và đưa sản phẩm ra thị trường nhanh hơn.

- Mỗi microservice có thể được triển khai độc lập, giúp giảm thiểu rủi ro và thời gian downtime khi cập nhật hệ thống.

- Có thể sử dụng được nhiểu ngôn ngữ lập tình khác nhau cho các tác vụ khác nhau

* + 1. **Tính sẵn sàng cao**

- Nếu một microservice gặp sự cố, các microservice khác vẫn hoạt động bình thường, đảm bảo tính sẵn sàng cao cho hệ thống.

- Microservice có thể được khởi động lại hoặc thay thế nhanh chóng khi gặp sự cố, giúp giảm thiểu thời gian downtime.

* + 1. **Tăng cường khả năng kiểm thử và triển khai liên tục**
* Với kiến trúc microservice, việc kiểm thử và triển khai liên tục (CI/CD) sẽ trở nên dễ dàng hơn. Các dịch vụ nhỏ có thể được kiểm thử độc lập và triển khai nhanh chóng mà không ảnh hưởng đến toàn bộ hệ thống. Điều này giúp cải thiện chất lượng phần mềm và giảm thời gian phát hành các tính năng mới.
  + 1. **Phù hợp với yêu cầu phát triển trong tương lai**

- Microservice cho phép thêm các tính năng mới một cách dễ dàng bằng cách thêm các microservice mới hoặc mở rộng các microservice hiện có.

- Các microservice có thể sử dụng các công nghệ khác nhau, cho phép lựa chọn công nghệ phù hợp nhất cho từng nghiệp vụ.

* 1. **Được gì?**

- Nâng cao trải nghiệm người dùng: Hệ thống phản hồi nhanh hơn và ổn định hơn, mang lại trải nghiệm người dùng tốt hơn.

- Dễ dàng tích hợp với các hệ thống khác: Microservices có thể dễ dàng tích hợp với các hệ thống khác thông qua API.

- Thích nghi tốt hơn với thay đổi: Hệ thống có thể dễ dàng thích nghi với các thay đổi trong yêu cầu nghiệp vụ.

- Tăng cường hiệu suất và khả năng chịu tải: Các dịch vụ có thể được triển khai và mở rộng một cách độc lập. Nếu một dịch vụ gặp sự cố hoặc cần mở rộng, chỉ cần điều chỉnh hoặc mở rộng dịch vụ đó mà không ảnh hưởng đến các dịch vụ khác.

- Độ tin cậy cao: Với cơ chế phân tán, nếu một dịch vụ gặp sự cố, các dịch vụ khác vẫn có thể hoạt động bình thường, giảm thiểu khả năng gián đoạn toàn hệ thống.

- Dễ dàng tích hợp và phát triển thêm: Với kiến trúc microservice, việc tích hợp thêm các dịch vụ mới hoặc kết nối với các hệ thống bên ngoài dễ dàng hơn. Điều này đặc biệt quan trọng khi hệ thống cần kết nối với các dịch vụ bên ngoài như hệ thống email, cổng thanh toán, hoặc các dịch vụ học tập trực tuyến.

- Quản lý và bảo trì dễ dàng hơn: Mỗi dịch vụ có thể được phát triển, quản lý, và bảo trì bởi các đội ngũ riêng biệt, giúp tối ưu hóa nguồn lực và tăng cường hiệu quả làm việc.

* 1. **Đánh đổi gì?**
* Microservices làm tăng thêm độ phức tạp về kỹ thuật do các service cần phải giao tiếp, đồng bộ, và quản lý độc lập. Nhưng nó cũng cho phép chặt chẽ kiểm soát giai đoạn đăng ký học phần, kiểm tra tiên quyết, và thông tin sinh viên theo cách linh hoạt hơn.
* Microservices cho phép các nhóm phát triển và bảo trì từng dịch vụ một cách độc lập, điều này có thể là lợi ích so với các kiến trúc nguyên khối như Layered, nhưng lại khiến việc quản lý tổng thể trở nên phức tạp.
* Trong microservices, mỗi dịch vụ có thể được triển khai một cách độc lập trên cloud và mở rộng tùy theo nhu cầu thực tế (ví dụ, tăng tài nguyên cho dịch vụ đăng ký học phần trong thời gian cao điểm). Điều này khác biệt so với kiến trúc nguyên khối hoặc các kiến trúc khác có đặc tính đàn hồi kém hơn.
* Microservices tăng cường tính chịu lỗi do mỗi dịch vụ hoạt động độc lập; một dịch vụ gặp sự cố không gây ảnh hưởng trực tiếp đến các dịch vụ khác. Trong khi đó, các kiến trúc như Layered hoặc Monolithic dễ chịu ảnh hưởng bởi lỗi lan truyền.
* Sử dụng cloud capabilities như auto scaling, cơ chế resilience có thể được cải thiện so với việc sử dụng một server monolithic, giúp tránh được việc sập server trong giai đoạn cao điểm đăng ký học phần.
* Các microservices có thể được triển khai riêng biệt, điều này giúp việc cập nhật tính năng mới trở nên nhanh chóng và hiệu quả, nhưng cũng đòi hỏi phải có quy trình deployment và continuous delivery được thiết lập tỉ mỉ.
* Mỗi microservice có thể có cơ sở dữ liệu riêng, khiến cho việc quản lý giao dịch có thể phức tạp nếu cần đồng bộ hóa dữ liệu giữa các dịch vụ.

1. **Implement a chosen architecture style**

Ảnh có chứa văn bản, ảnh chụp màn hình, biểu đồ, thiết kế

Mô tả được tạo tự động

1. **Modelling and presenting a software architecture**
   1. **Use case**

A diagram of people with text and words

Description automatically generated

**5.2 UML: package diagram, deployment diagram, sequence diagram, component diagram**

**5.2.1 Package diagram**

5.2.1 Frontend – Package diagram

A blue and white diagram

Description automatically generated

5.2.2 Backend – Package diagram

5.2.2.1 Subject Service – Student Service - Lecturer Service

A diagram of a computer network

Description automatically generated with medium confidence

5.2.2.2 Auth Service

A diagram of a folder

Description automatically generated

**5.3 Deployment diagram**

A diagram of a computer

Description automatically generated

5.3 Sequence diagram

5.4. Component diagram

* 1. Class diagram

5.5.1 Subject Service – Class diagram

A diagram of a computer flowchart

Description automatically generated

5.2 Student Service – Class diagram

A screenshot of a computer

Description automatically generated

5.3. Lecturer Service – Class diagram

A diagram of a computer

Description automatically generated with medium confidence