

# Chapter 1

## Introduction to Monolithic Application, Service-Oriented Architecture and Microservice

---

บรรยายโดย ผศ.ดร.ธราวดิษฐ์ ริติจรูญโรจน์ และอาจารย์สัญชัย น้อยจันทร์

คณะเทคโนโลยีสารสนเทศ

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง



# Monolithic



Monolithic เป็นคำนาม (Noun) ที่เกิดจากการประสมคำระหว่างคำว่า “monos” และ “lithos” ที่มาจากการภาษา Greek โดยที่

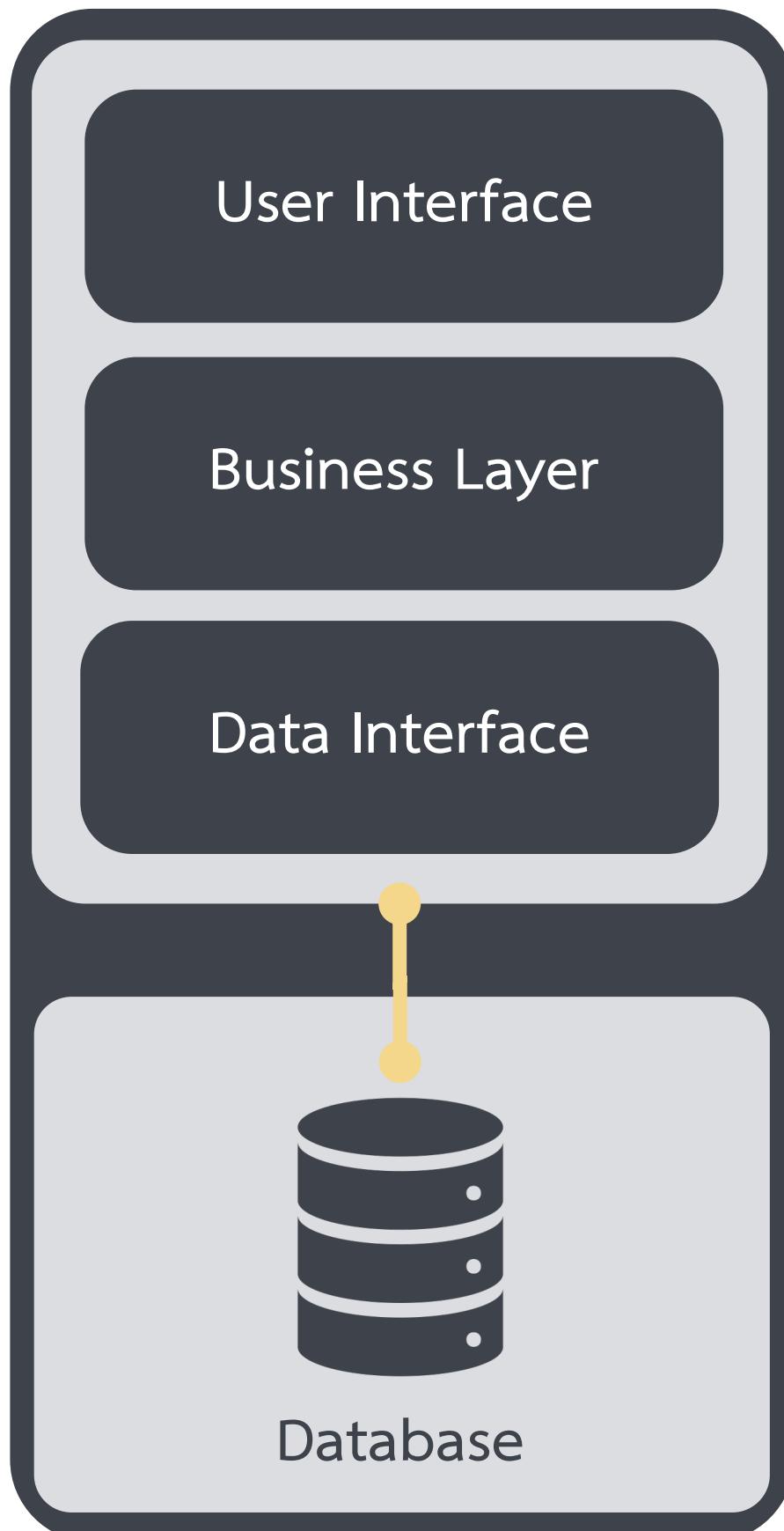
- Monos แปลว่า Single
- Lithos แปลว่า Stone

ซึ่งมีความหมายว่า “ก้อนหินตั้งตรงขนาดใหญ่” โดยเฉพาะที่มีรูปร่างเป็นหน้าที่เป็นเสาหรืออนุสาวรีย์



# Monolithic

## Monolithic Architecture



Monolithic Application คือ แนวคิดหนึ่งในการออกแบบและพัฒนาระบบงานต่าง ๆ ที่มีความเกี่ยวข้องกันในรูปแบบ “Single Unit or Single Application” โดยมีลักษณะคล้าย Stack ใน การพัฒนาระบบแบบ Monolithic จะมีความซับซ้อนค่อนข้างมาก ระบบมีขนาดใหญ่ และแต่ละงานจะมีความสัมพันธ์กันแบบ “Tightly Coupled” นอกจากนี้ ยังมีลักษณะเป็น one code base, one build system และ single executional program (war file)

การพัฒนาระบบแบบ Monolithic โดยส่วนใหญ่ได้แบ่งออกเป็น 3 ส่วน (หรือ เลเยอร์) ได้แก่ Data Interface (Model), User Interface (View) และ Business Layer (Controller)

ระบบที่ได้รับการพัฒนาแบบ Monolithic นั้นจะได้รับการออกแบบและใช้งานฐานข้อมูลเพียงตัวเดียว ระบบมีให้บริการ (หรือ Service) จากฝั่งผู้ให้บริการ (Server หรือ Backend) เพียงตัวเดียว สำหรับการจัดการต่าง ๆ อาทิ เช่น เพิ่ม แก้ไข และลบข้อมูลกับฐานข้อมูลสำหรับเวลาที่ผู้รับบริการ (Client) ร้องขอการเรียกใช้งาน Service ต่าง ๆ



# Monolithic

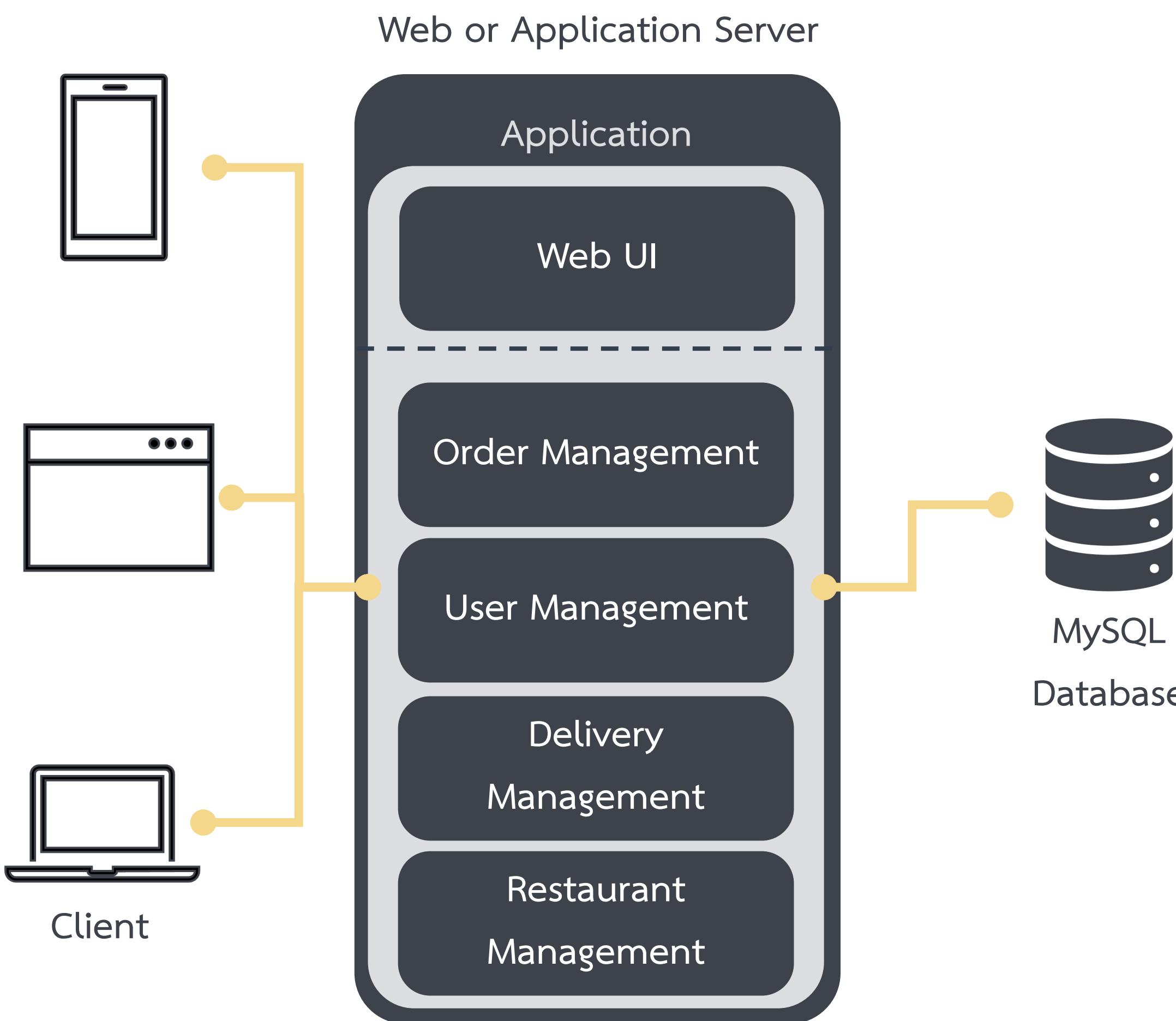


การพัฒนาระบบแบบ monolithic เหมาะสมกับระบบขนาดเล็ก ซึ่งจะมีข้อดีต่าง ๆ ดังนี้

- ระบบพัฒนาได้สะอาดๆ เนื่องจากทุกอย่างจะอยู่ในเพียงโปรเจคเดียว
- สามารถทดสอบ (Test) และนำไปใช้งาน (Deploy) ได้แบบตรงไปตรงมาไม่ซับซ้อน เนื่องจากทำทุกอย่างเพียงระบบเดียว
- ~~สะดวกต่อการปรับขนาดการรองรับการทำงานของระบบ (Scaling) โดยการสร้าง instance เพิ่มร่วมกับ Load Balancer~~



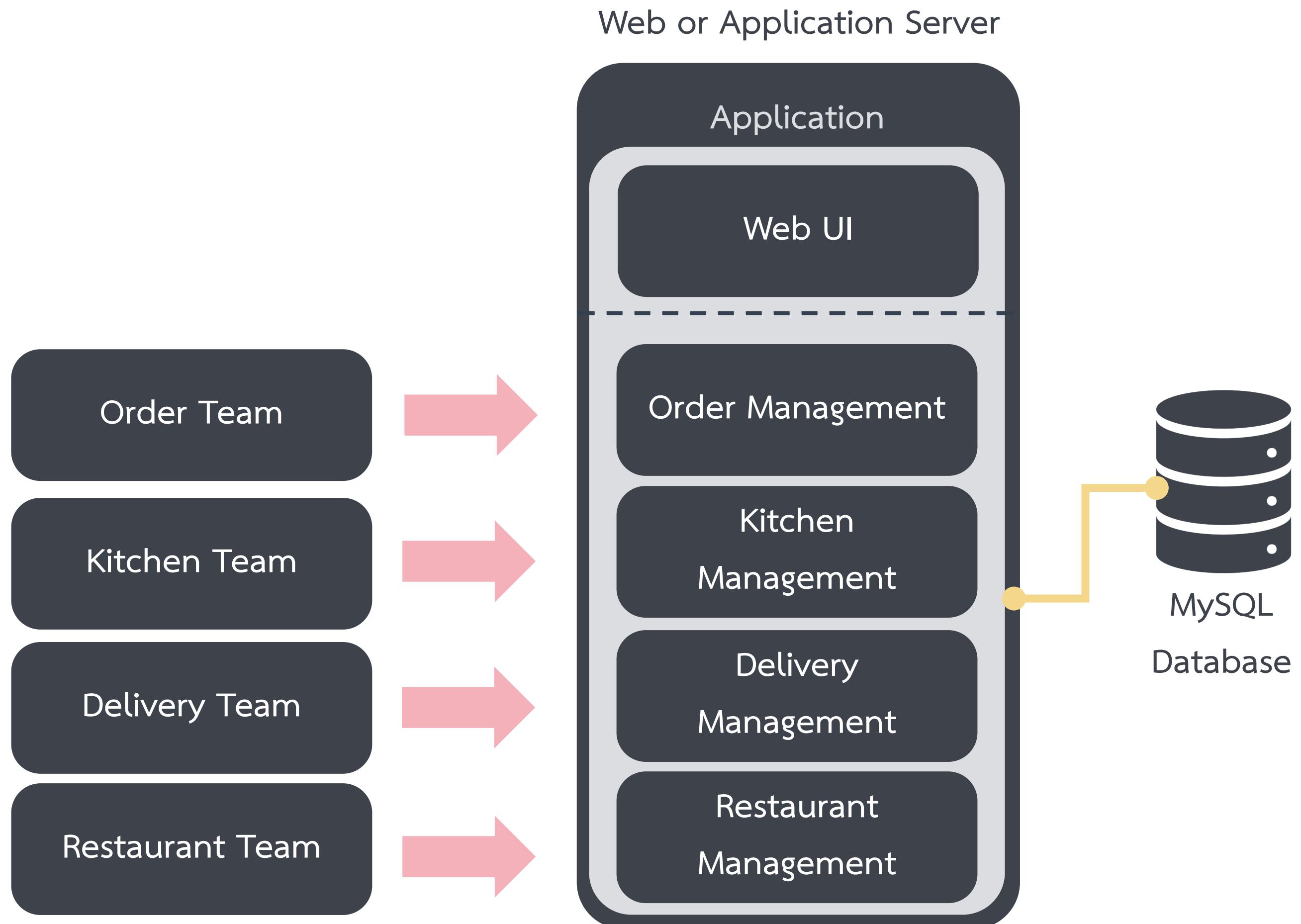
# Monolithic



ตัวอย่างระบบ Food Delivery Application ที่ได้รับการออกแบบ และพัฒนาระบบโดยอาศัยสถาปัตยกรรมแบบ Monolithic จากรูป จะเห็นว่าระบบประกอบไปด้วย 4 Service ได้แก่ Order Management, User Management, Delivery Management และ Restaurant Management ซึ่งทั้ง 4 Service ได้รับการ พัฒนาไว้ในที่เดียวกันแบบ “ Single Unit” และใช้ฐานข้อมูลเดียวกัน



# Monolithic



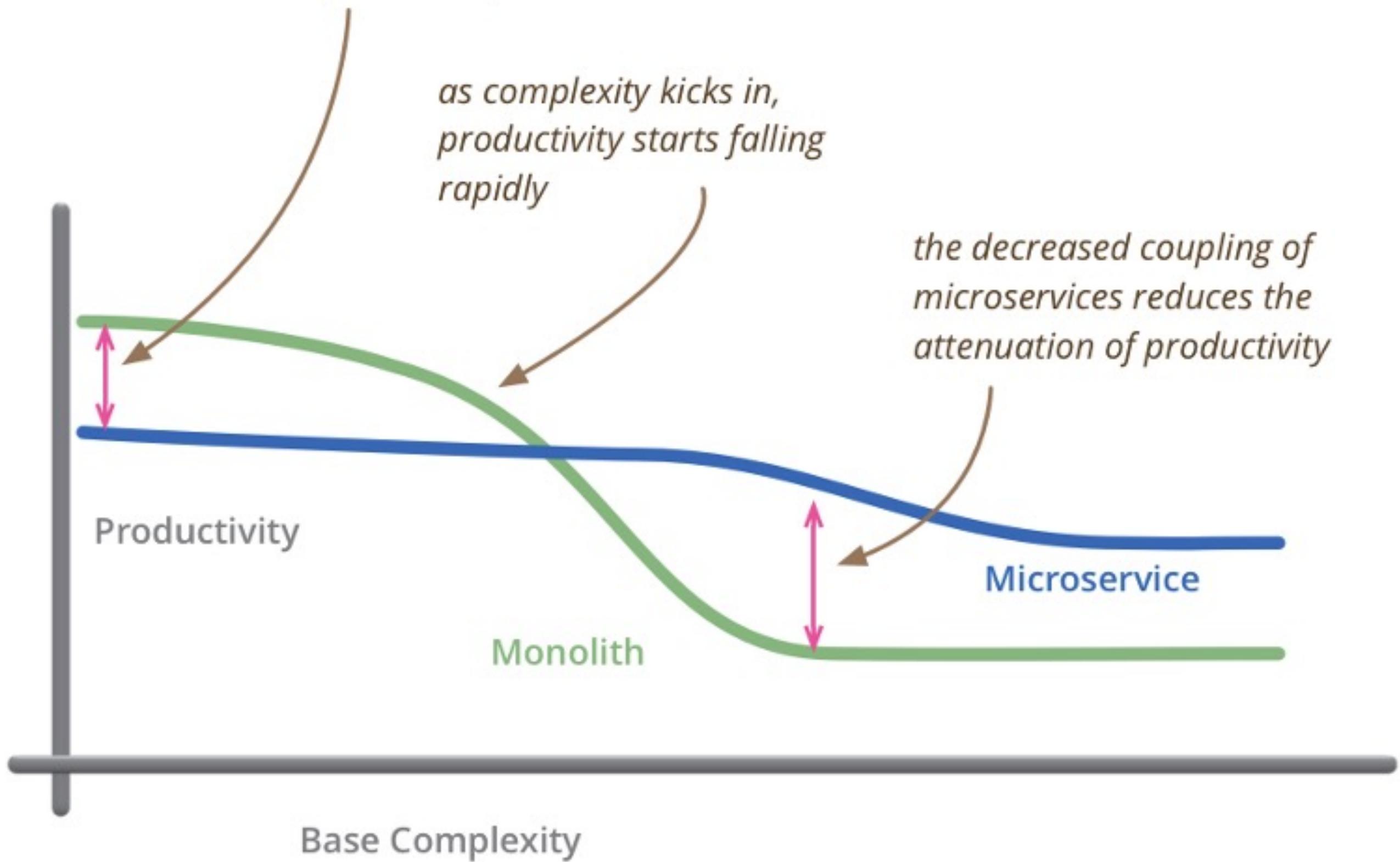
ในการทำงานจะมีการกำหนดแต่ละโมดูลให้แต่ละทีม เป็นผู้รับผิดชอบในการพัฒนาภายใต้เงื่อนไขเดียวกัน

- ภาษาและสถาปัตยกรรมที่ใช้ในการพัฒนา  
ในการพัฒนาระบบโดยส่วนใหญ่จะมีข้อตกลงกันเรื่องภาษาที่ใช้ในพัฒนา ซึ่ง 1 ระบบต่อ 1 ภาษา ทำให้ภาษาเป็นข้อจำกัดที่เกิดขึ้น เพราะแต่ละภาษามีข้อดี ข้อเสียแตกต่างกัน
- สภาพแวดล้อมและสเปคของเครื่องคอมพิวเตอร์  
งานแต่ละชนิดจะใช้เครื่องคอมพิวเตอร์สเปคแตกต่างกัน เช่น โมดูล Logging จะเน้นการเก็บข้อมูลแบบ write เป็นหลัก ขณะที่โมดูลการแนะนำตามความต้องการของลูกค้าแบบอัจฉริยะจะเน้นการประมวลผลเป็นหลัก
- ชนิดหรือประเภทของฐานข้อมูล  
งานแต่ละชนิดจะมีการเก็บข้อมูลที่แตกต่างขึ้นอยู่กับโครงสร้างของข้อมูล ซึ่ง ข้อมูลบางประเภทอาจจะมีโครงสร้างที่หลากหลายก็เป็นได้ทำให้ไม่เหมาะสม กับฐานข้อมูลประเภท Relational



# Monolithic Hell

*for less-complex systems, the extra baggage required to manage microservices reduces productivity*



Ref: <http://martinfowler.com/>

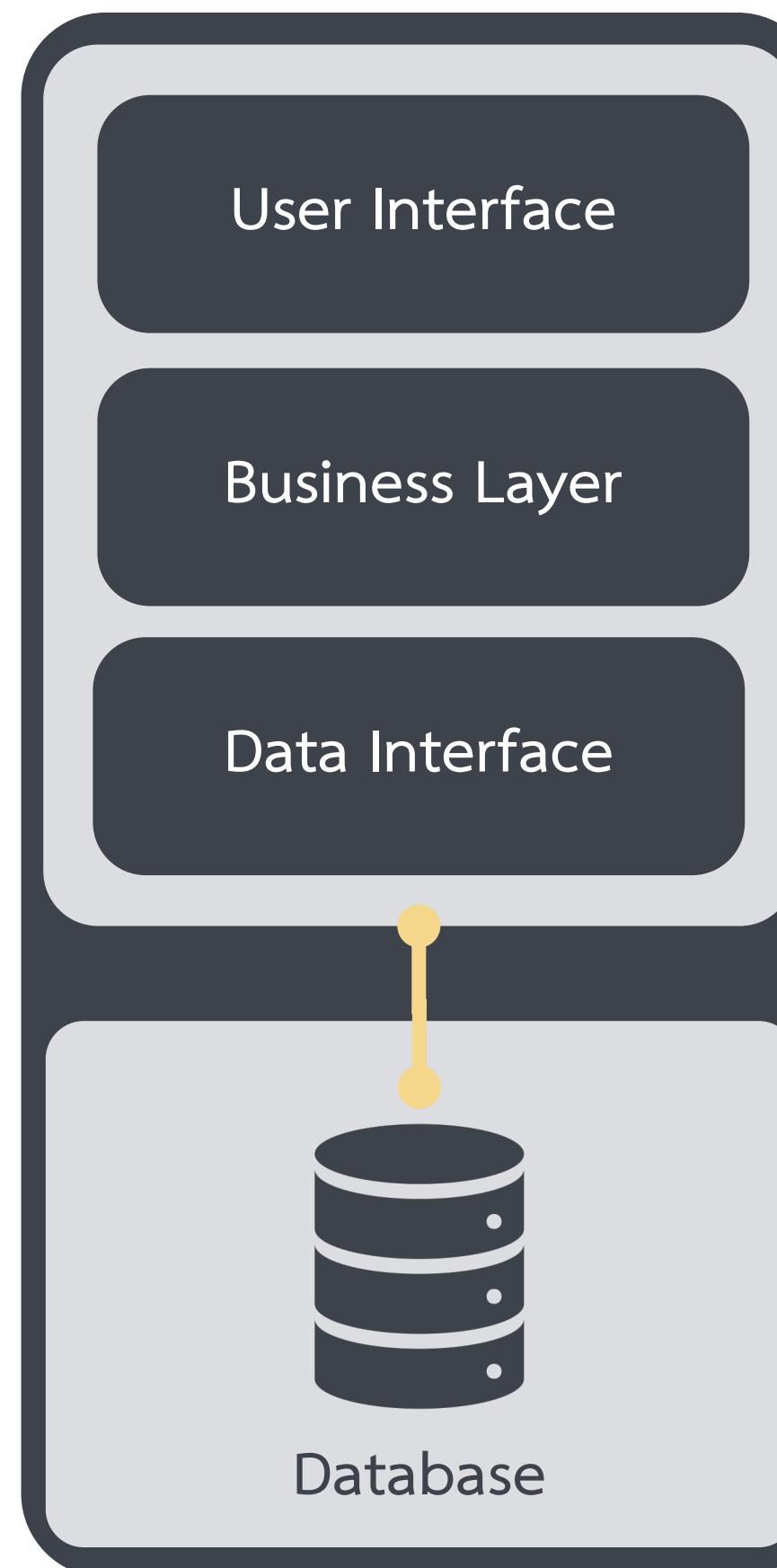
*but remember the skill of the team will outweigh any monolith/microservice choice*

การพัฒนาระบบบนหลักการ Monolithic จะมีประสิทธิภาพและรวดเร็วในช่วงแรก (Initial Stage) แต่เมื่อระบบได้รับการพัฒนาให้มีขนาดใหญ่มากขึ้นจะก่อให้เกิดเหตุการณ์ที่เรียกว่า “Monolithic Hell” กล่าวคือ ถ้าระบบมีขนาดใหญ่ขึ้น เนื่องจากในทางปฏิบัติภาครุ่งกิจจะมีการปรับเปลี่ยนตลอดเวลาเพื่อให้เกิดความเหมาะสม ทำให้ความต้องการ (Requirement) ของระบบมีการเพิ่มเติมหรือปรับเปลี่ยนตามไปด้วย ทำให้ระบบมีขนาดที่เติบโตมากขึ้นอยู่เสมอ (ปริมาณโค้ดมากขึ้น “Huge Code Bases”)



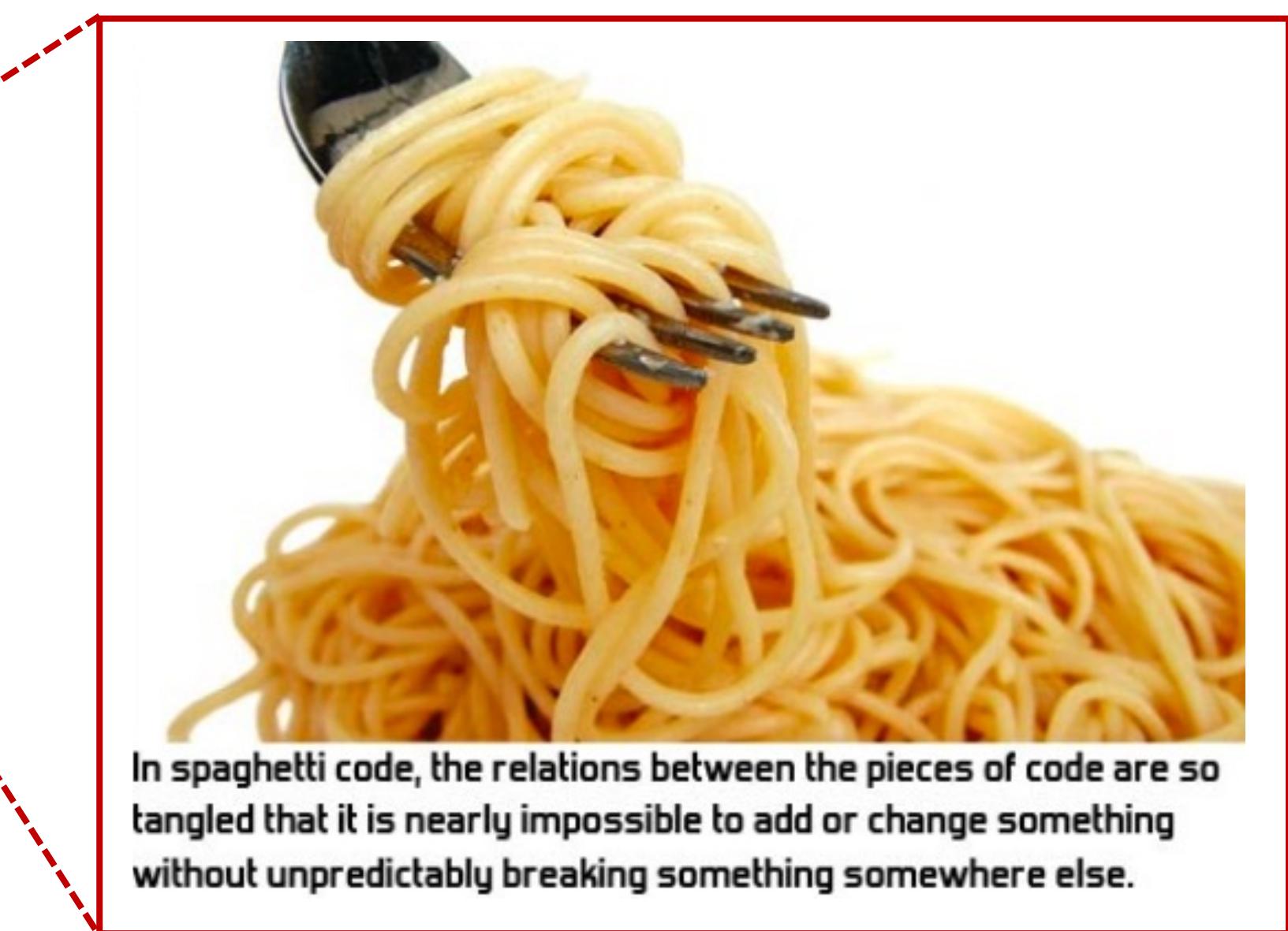
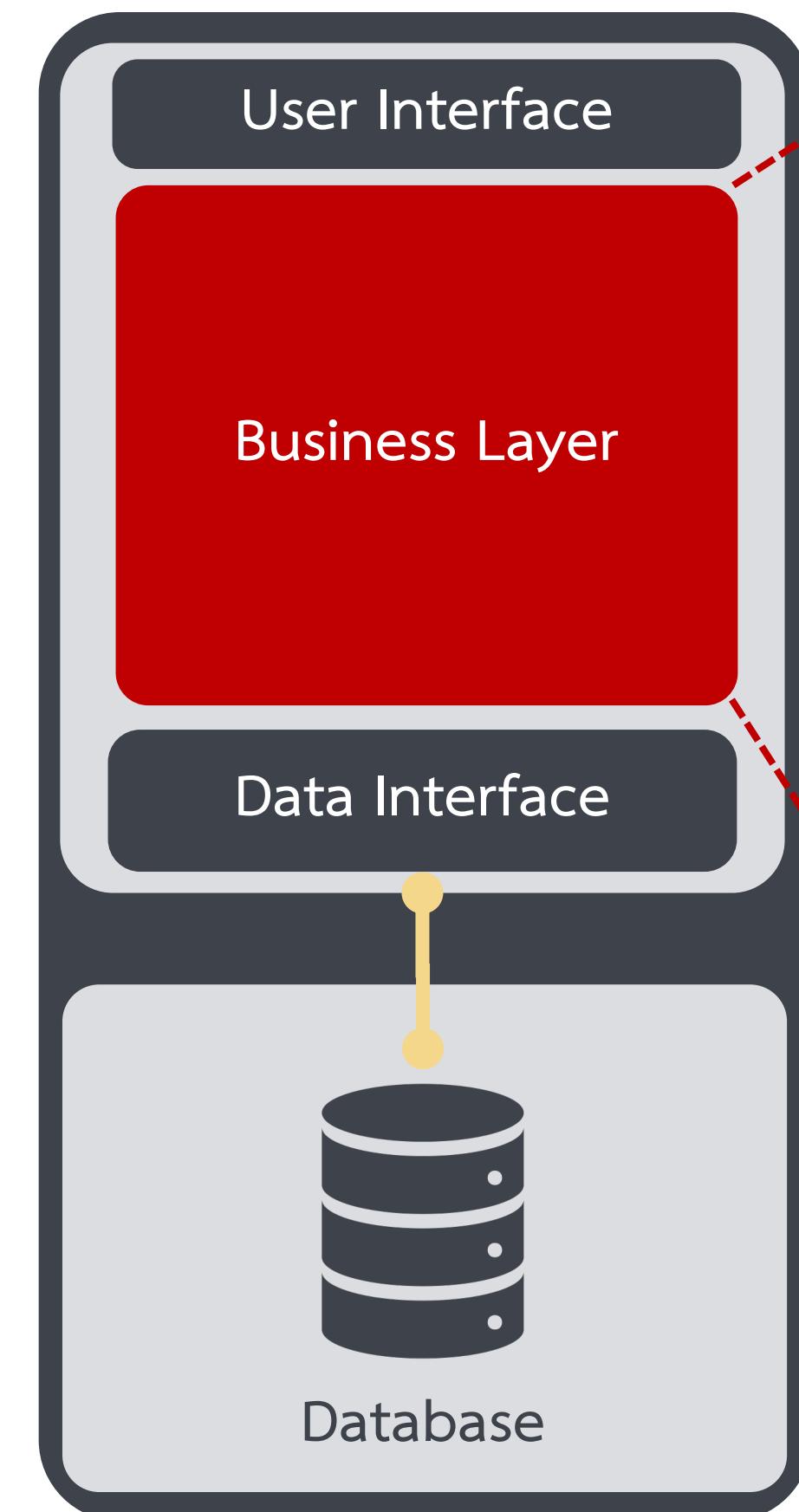
# Monolithic Hell

Monolithic Architecture



System Growth  
→

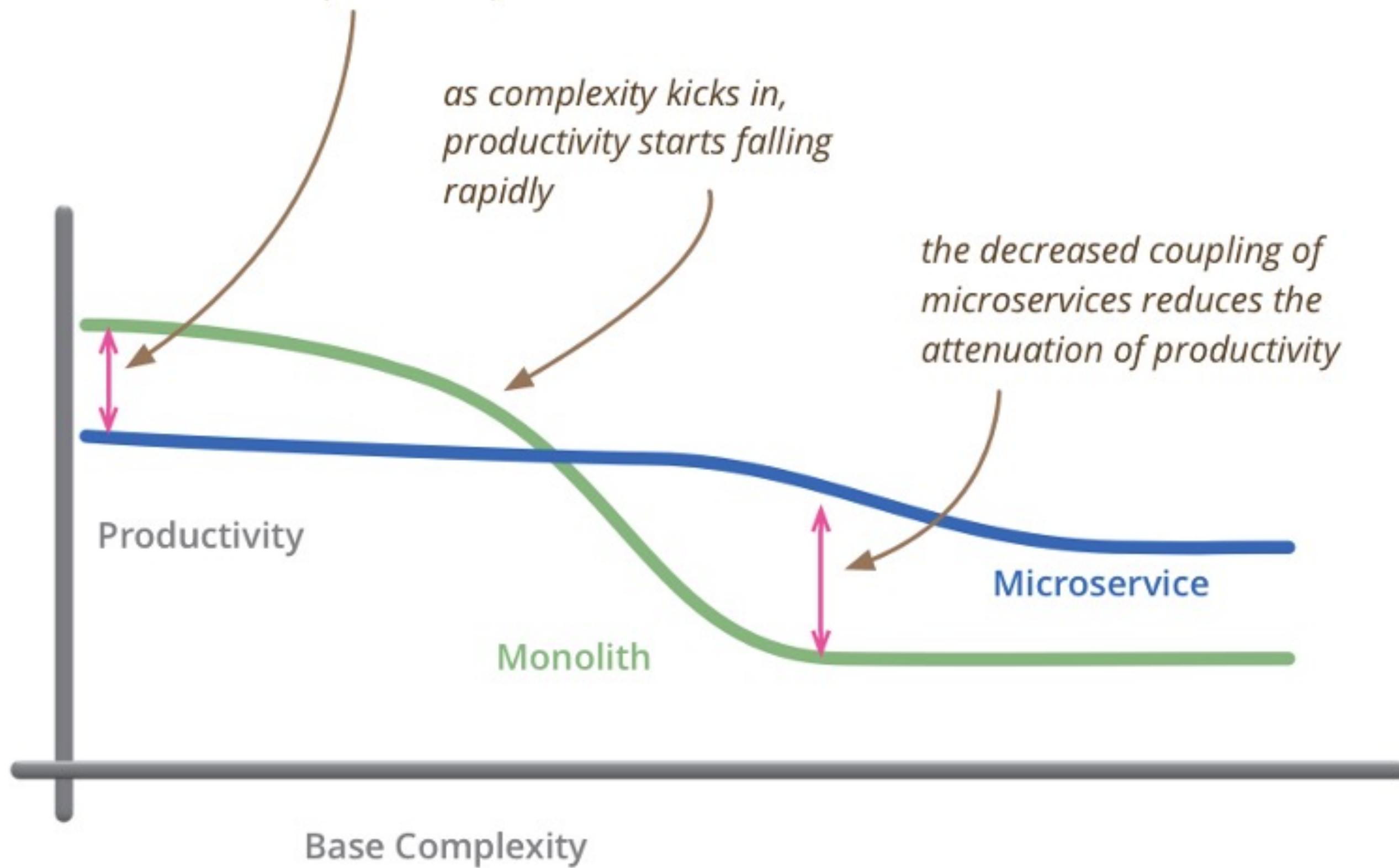
Monolithic Architecture





# Monolithic Hell

*for less-complex systems, the extra baggage required to manage microservices reduces productivity*



สิ่งเหล่านี้ส่งผลให้เกิด **ความลำบาก** และ **ล่าช้า** ต่อ การ develop, test, deploy และ scale เพราะ โค้ดมีปริมาณที่มากขึ้นเรื่อย ๆ และแต่ละโค้ดมี ความสัมพันธ์ที่ผูกมัดกันค่อนข้างมาก ("Tight Coupling") แก้ไขแล้วไม่รู้จะไปกระทบส่วน ใดบ้าง จากที่ระบบใช้งานได้อาจจะล่มก็เป็นได้

*but remember the skill of the team will outweigh any monolith/microservice choice*



# Monolithic

## ข้อเสียของการพัฒนาระบบแบบ Monolithic ประกอบด้วย 5 ประการ

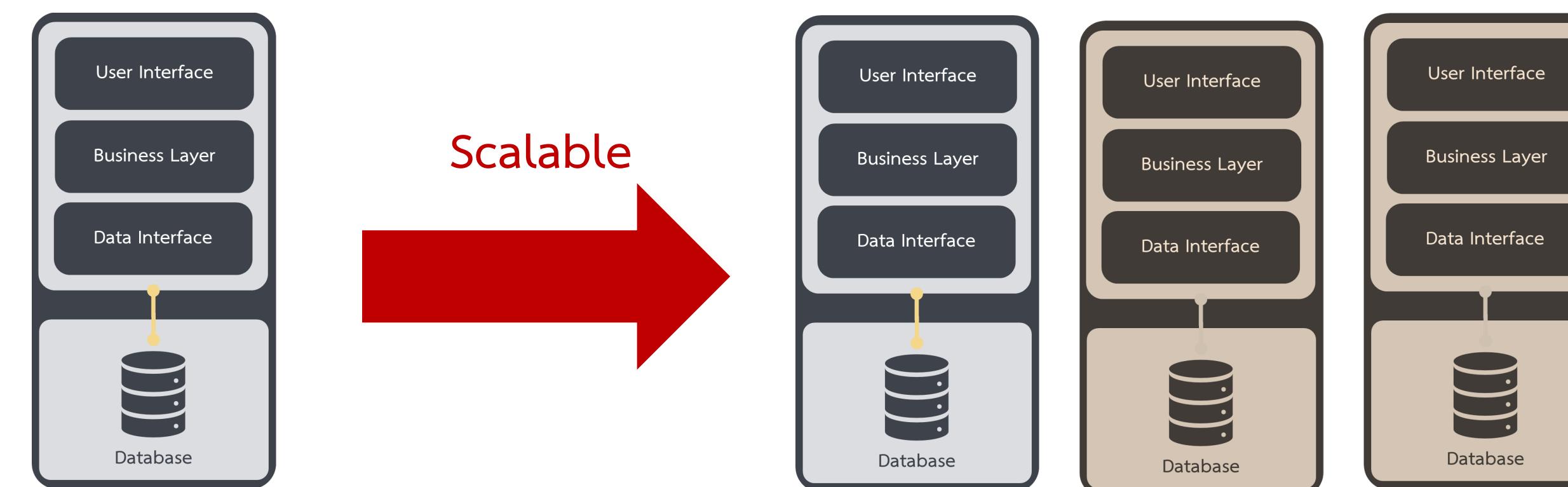
- ระบบใช้เวลาในการพัฒนาที่ค่อนข้างมาก (Slow Development)
  - ✓ เสียเวลาสำหรับการ Compile หรือ Run ไปค่อนข้างมาก เนื่องจากระบบส่วนใหญ่จะมีขนาดค่อนข้างใหญ่ (หรือมีชุดคำสั่งในปริมาณที่ค่อนข้างมาก) ส่งผลให้โปรแกรมที่ใช้ในการพัฒนา (IDE) ทำงานได้ล่าช้าและใช้เวลาค่อนข้างมาก
  - ✓ เสียเวลาทำความเข้าใจชุดคำสั่งเดิมของระบบ ส่งผลให้เกิดความล่าช้า เมื่อต้องการเพิ่มชุดคำสั่งใหม่หรือปรับปรุงแก้ไขชุดคำสั่งต่าง ๆ ในกรณีที่ผู้พัฒนาเป็นคนละคนกัน
- ระบบไม่ค่อยมีความยืดหยุ่น (Inflexible)
  - ✓ การนำเทคโนโลยีใหม่เข้ามาใช้งานหรือเลือกใช้เทคโนโลยีที่แตกต่างกันในแต่ละบริการเป็นเรื่องค่อนข้างยากลำบาก
  - ✓ การเปลี่ยนแปลงเทคโนโลยีในแต่ละบริการก็เป็นเรื่องค่อนข้างยากลำบากเช่นกัน
- ระบบขาดความน่าเชื่อถือ (Unreliable)
  - ✓ ถ้าชุดคำสั่งหรือ Service ได้เกิดทำงานผิดพลาดหรือขัดข้องอาจจะส่งผลให้ระบบขัดข้องหรือไม่สามารถทำงานต่อไปได้ ดังนั้น เมื่อเกิดความล้มเหลวขึ้นมาที่ส่วนใดส่วนหนึ่ง ก็จะส่งผลกระทบกันไปทั้งหมด หรือเรียกว่า “Single Point of Failure”



# Monolithic

- ระบบขาดความคล่องตัวสำหรับการปรับขนาดของระบบ (Unscalable)

- ✓ การปรับขนาดของระบบเป็นสิ่งจำเป็นเพื่อรับการปรับมาลดข้อมูลที่มีปริมาณมหาศาลและปริมาณผู้ใช้งานที่มีจำนวนมากขึ้น ซึ่งถือว่าเป็นความท้าทายอย่างมากสำหรับระบบแบบ Monolithic เนื่องจากระบบมีลักษณะ Stack และแต่ละส่วนงานมีความสัมพันธ์กันแบบ “Tightly Coupled” ทำให้ไม่มีความเป็นอิสระออกจากกัน
- ✓ ถ้ามี Service ได้ต้องการเพิ่มขนาด เพื่อรับบริการที่มากขึ้น จะไม่สามารถขยายเฉพาะ Service นั้นได้ แต่ต้องขยายทั้งระบบแทน



- ระบบขาดความคล่องตัวในพัฒนาอย่างต่อเนื่อง (Blocks Continuous Development)

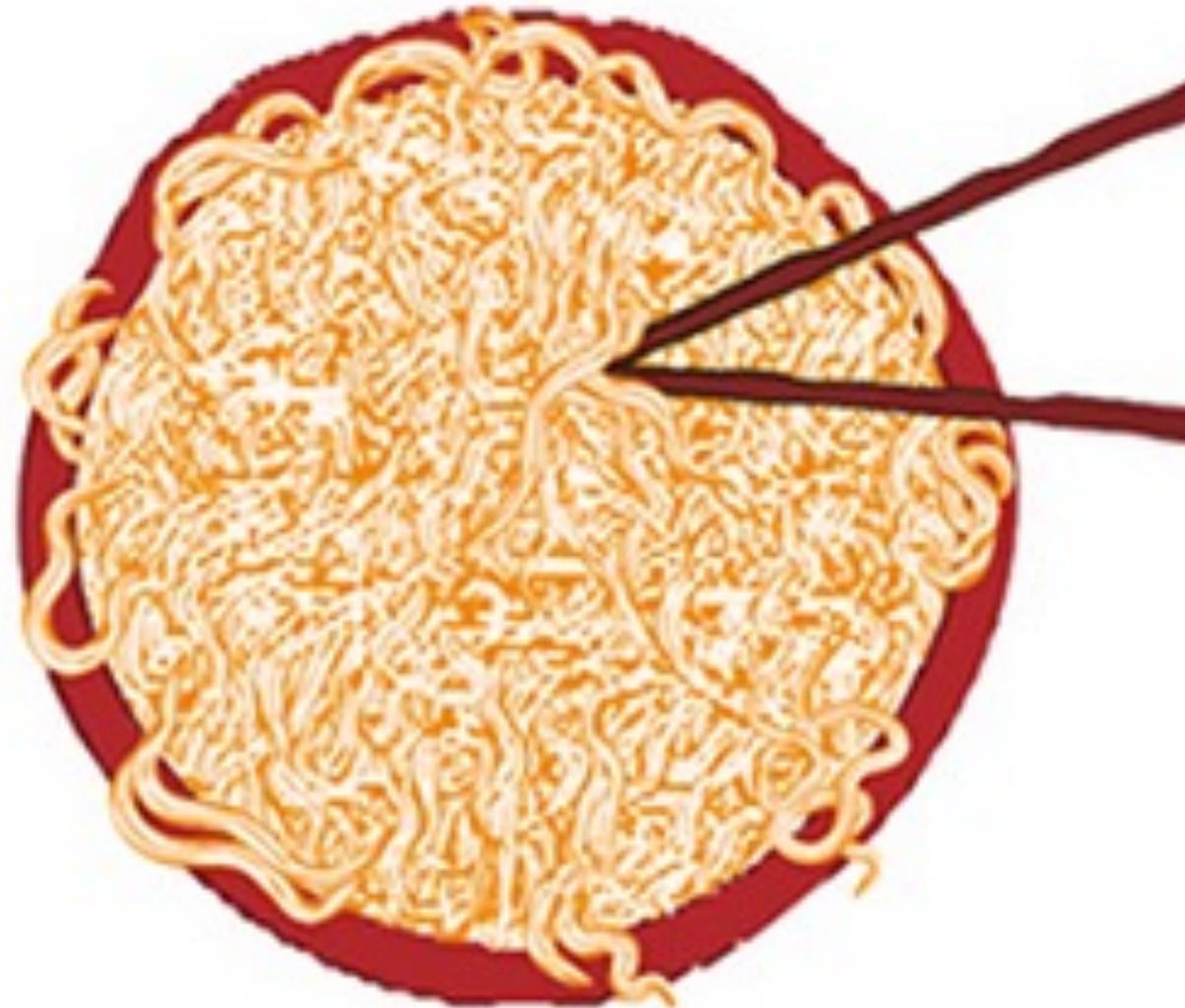
- ✓ การ Deploy ระบบมีความจำเป็นต้องทำพร้อมกันทุก Service ไม่สามารถแยกส่วนการ Deploy ได้ แล้วยังทำให้ยากต่อการทดสอบระบบ สิ่งเหล่านี้ส่งผลกระทบต่อระบบที่มีขนาดใหญ่มาก ทำให้ความเสี่ยงต่อระบบเพิ่มมากขึ้น



# Architecture Evolution

## 1990s and earlier

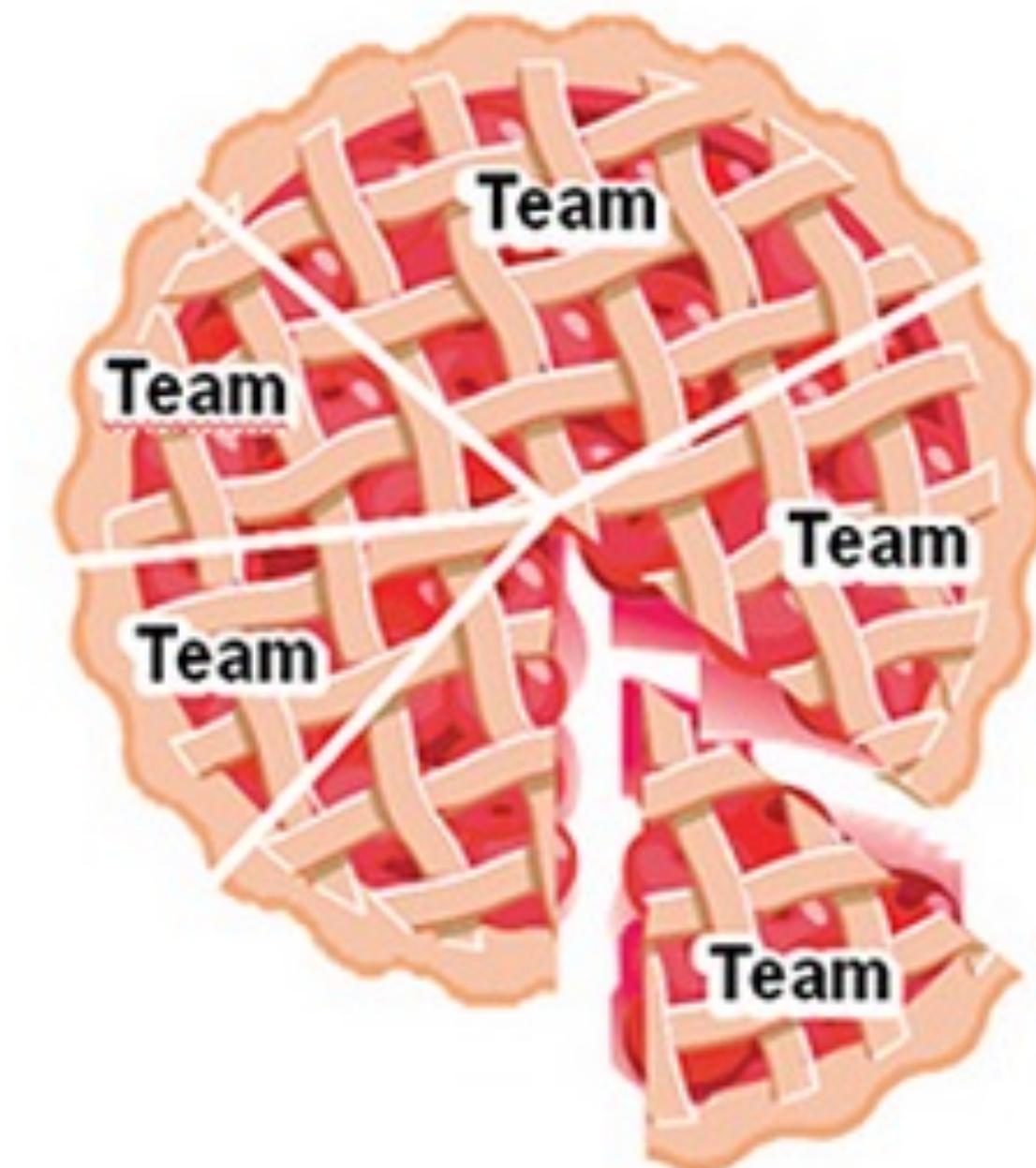
Pre-SOA (monolithic)  
Tight coupling



For a monolith to change, all must agree on each change. Each change has unanticipated effects requiring careful testing beforehand.

## 2000s

Traditional SOA  
Looser coupling



Elements in SOA are developed more autonomously but must be coordinated with others to fit into the overall design.

## 2010s

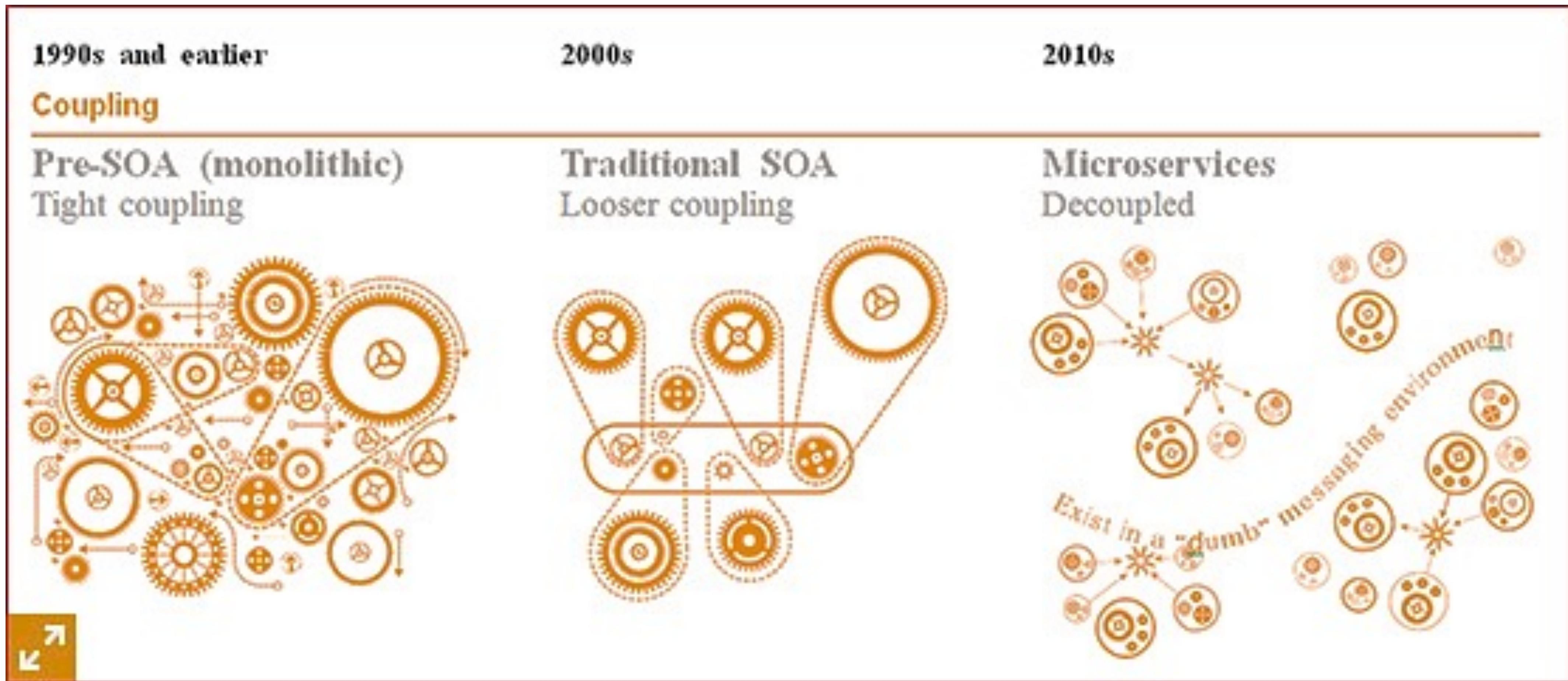
Microservices  
Decoupled



Developers can create and activate new microservices without prior coordination with others. Their adherence to MSA principles makes continuous delivery of new or modified services possible.

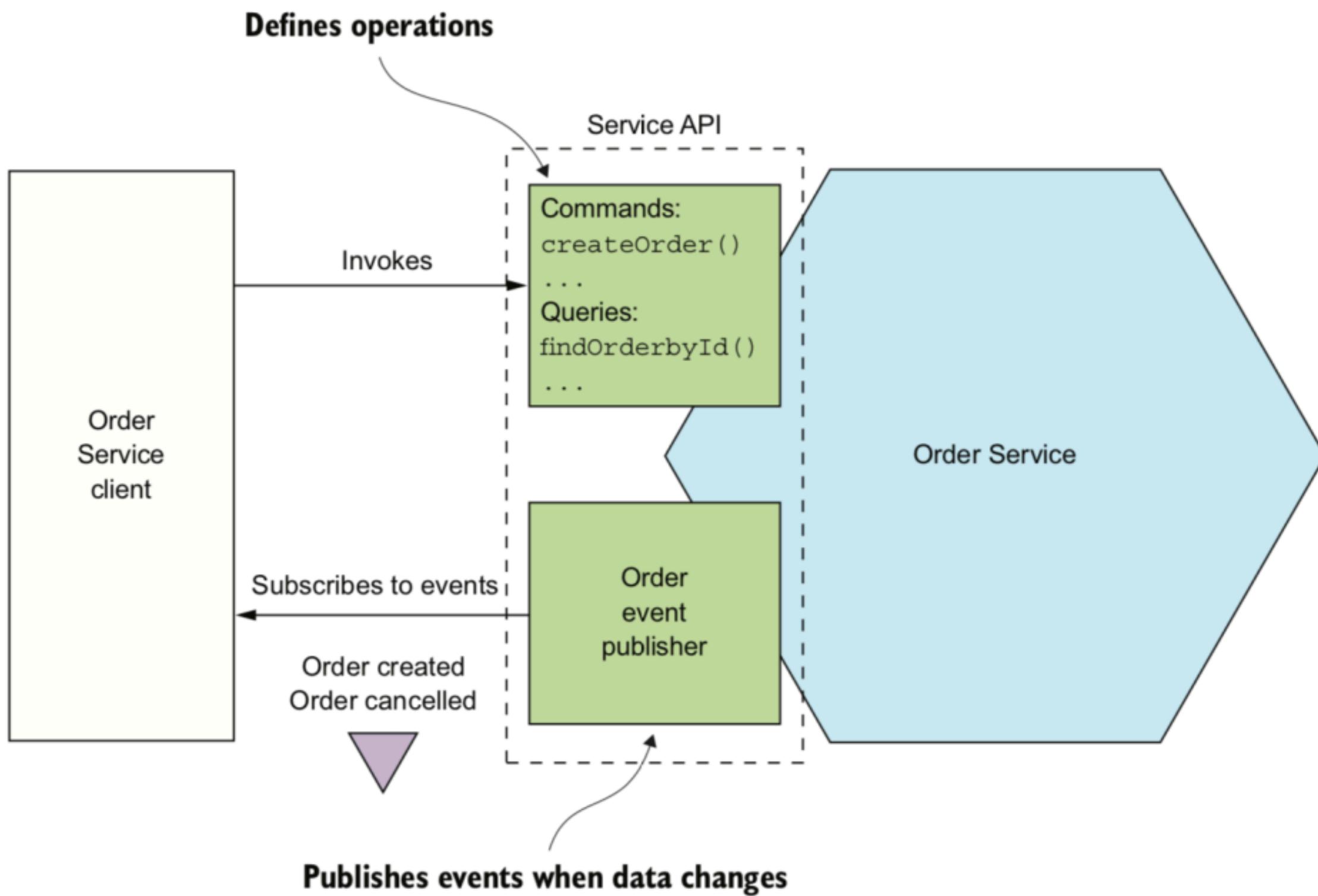


# Architecture Evolution





# Service



Service คือ ส่วนหนึ่งของโปรแกรมหรือระบบ (Software Component or Small Application) ที่มีหน้าที่รับผิดชอบจำกัดและซัดเจน และได้รับการออกแบบ standalone และ loosely couple ที่เป็นอิสระต่อกัน โดยเฉพาะอย่างยิ่ง ขณะทำการ Deploy

- **Standalone** หมายถึง ไม่พึ่ง Service อื่น ๆ สามารถทำงานได้ด้วยตัวมันเอง ถ้าเกิดเหตุการณ์ที่ Service อื่นไม่สามารถทำงานได้ตามปกติ ของเราก็ยังสามารถทำงานได้ตามปกติ
- **Loosely Couple** หมายถึง ไม่เกี่ยวข้องหรือผูกมัด Service ตัวใด กับ Service อื่น ๆ หรือผูกมัดให้น้อยที่สุดเท่าที่จำเป็น
- **Deploy** หมายถึง สามารถ Deploy แยกกันได้ ไม่ต้องรอหรือขึ้นอยู่กับ Service อื่น ๆ ซึ่งสามารถกล่าวได้ว่า Service มีความเป็นอิสระต่อกัน



# Service-Oriented Architecture

Service-Oriented Architecture (SOA) คือ การออกแบบและพัฒนาระบบที่มีความต้องการแก้ปัญหาและเปลี่ยนแปลงจาก Monolithic ให้อยู่ในรูปแบบชุดของ Services เพื่อให้เกิดความยืดหยุ่นต่อการใช้งาน ซึ่งแต่ละ Service สามารถติดต่อสื่อสารระหว่างกันได้ผ่านอินเตอร์เฟสต่าง ๆ จากแนวความคิดดังกล่าวทำให้สามารถ Deliver ระบบได้อย่าง

- ต่อเนื่อง (Consistence) และยั่งยืน (Sustainable)
- สามารถปรับเปลี่ยนได้ตามความต้องการของภาคธุรกิจ เพราะภาคธุรกิจมีการปรับตัวอย่างรวดเร็ว
- เป็นการใช้ต้นทุนอย่างคุ้มค่า (Cost Effectiveness)

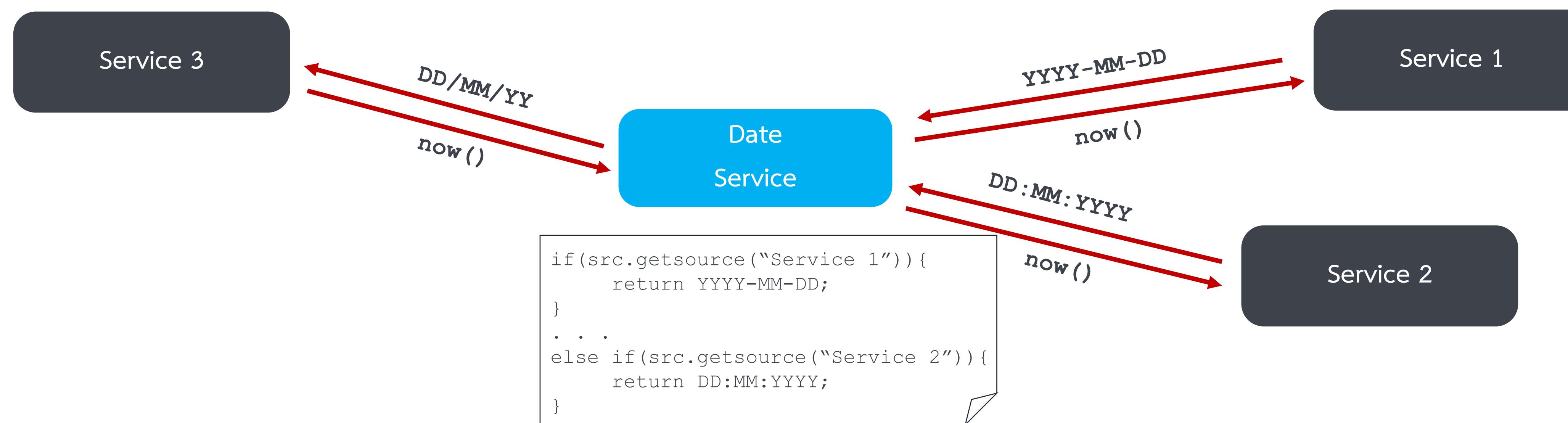
ซึ่งการออกแบบและพัฒนาระบบนพื้นฐานของ “Service-Oriented Architecture” มีแนวคิดที่ว่า

- ให้ความสำคัญกับ Business value มากกว่าลักษณะ Technical
- ให้ความสำคัญกับ Flexibility มากกว่า Optimization
- ให้ความสำคัญกับ Shared service มากกว่า Implementation ที่มีความเฉพาะ
- ให้ความสำคัญกับ เป้าหมายเชิงกลยุทธ์มากกว่า ผลประโยชน์เฉพาะโครงการ
- ให้ความสำคัญกับการปรับแต่งเชิงวิัฒนาการมากกว่าการแสวงหาความสมบูรณ์แบบเริ่มต้น

✓ กล่าวคือ การออกแบบและพัฒนาระบบ ควรค่อย ๆ ปรับ (วิัฒนาการ) ตาม Feedback ในแต่รอบของการพัฒนา



# Service-Oriented Architecture

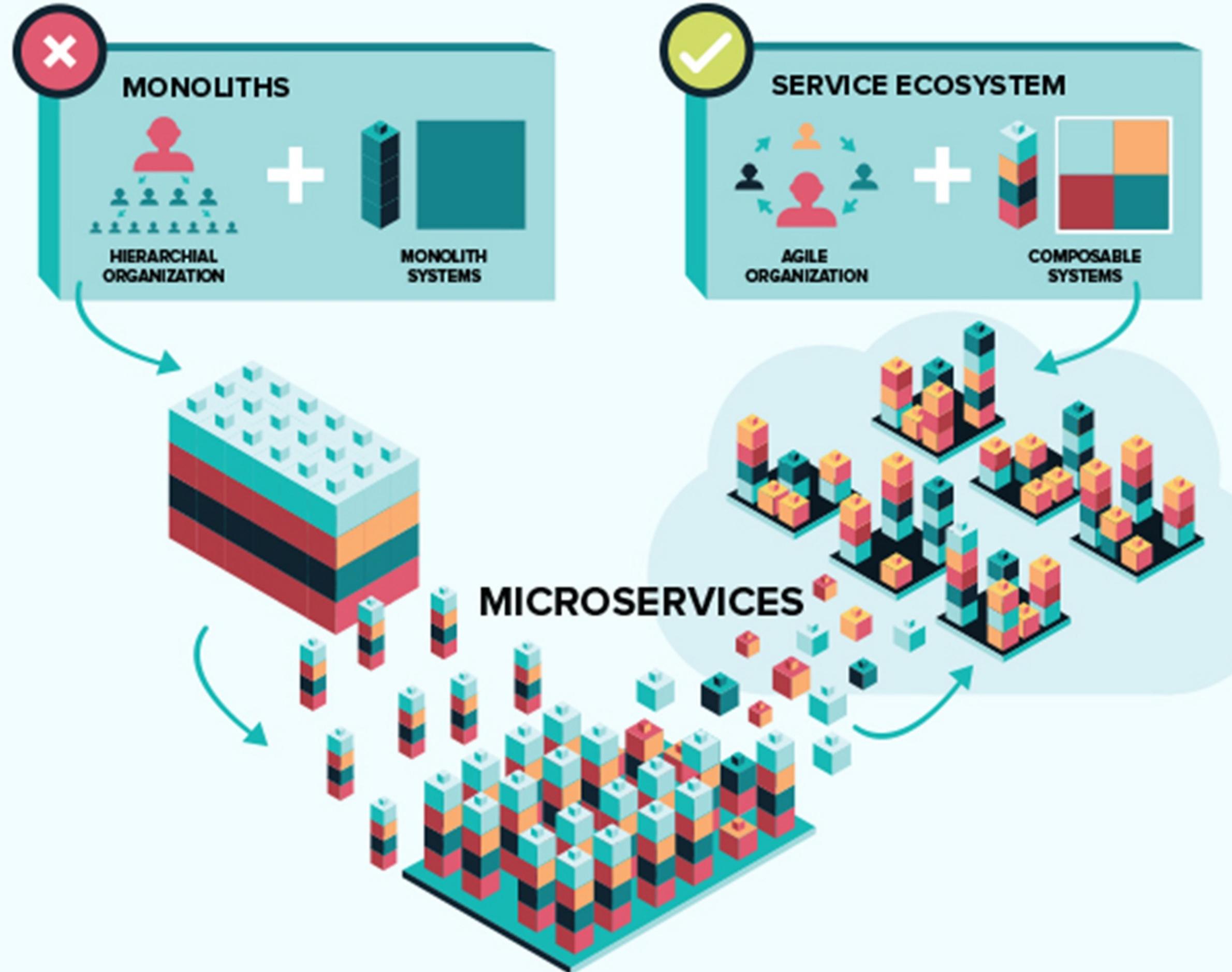


 “ให้ความสำคัญกับ Shared service มากกว่า Implementation ที่มีความเฉพาะ” อาจก่อให้เกิด **ผลเสีย** ตามมาดังต่อไปนี้

- ✓ คนพัฒนาโปรแกรมทำตัวเสมือนเป็นหมอดูและคาดเดาเอาว่าอนาคต Service อะไรจะใช้งานเยอะ ทั้งที่เมื่อ Deploy จริง ๆ อาจจะไม่มีใครเรียกใช้งานก็ได้
- ✓ การสร้าง Shared service มีจุดประสงค์เพื่อ Reuse แต่ต้องระวังเรื่องของ
  - อาจก่อให้เกิด “Single Point of Failure”
  - Reuse หมายความว่า ให้นำผลลัพธ์ที่ได้จาก Shared service ไปใช้งาน ไม่ควรแก้ไขผลลัพธ์เหมาะสมกับ Service นั้น ๆ ก่อน (ถ้าจะแก้ไขให้ปรับปรุงตาม Business Logic ห้ามแก้ไขตามความต้องการของ Service ที่เรียกใช้งาน)



# Microservice Architecture



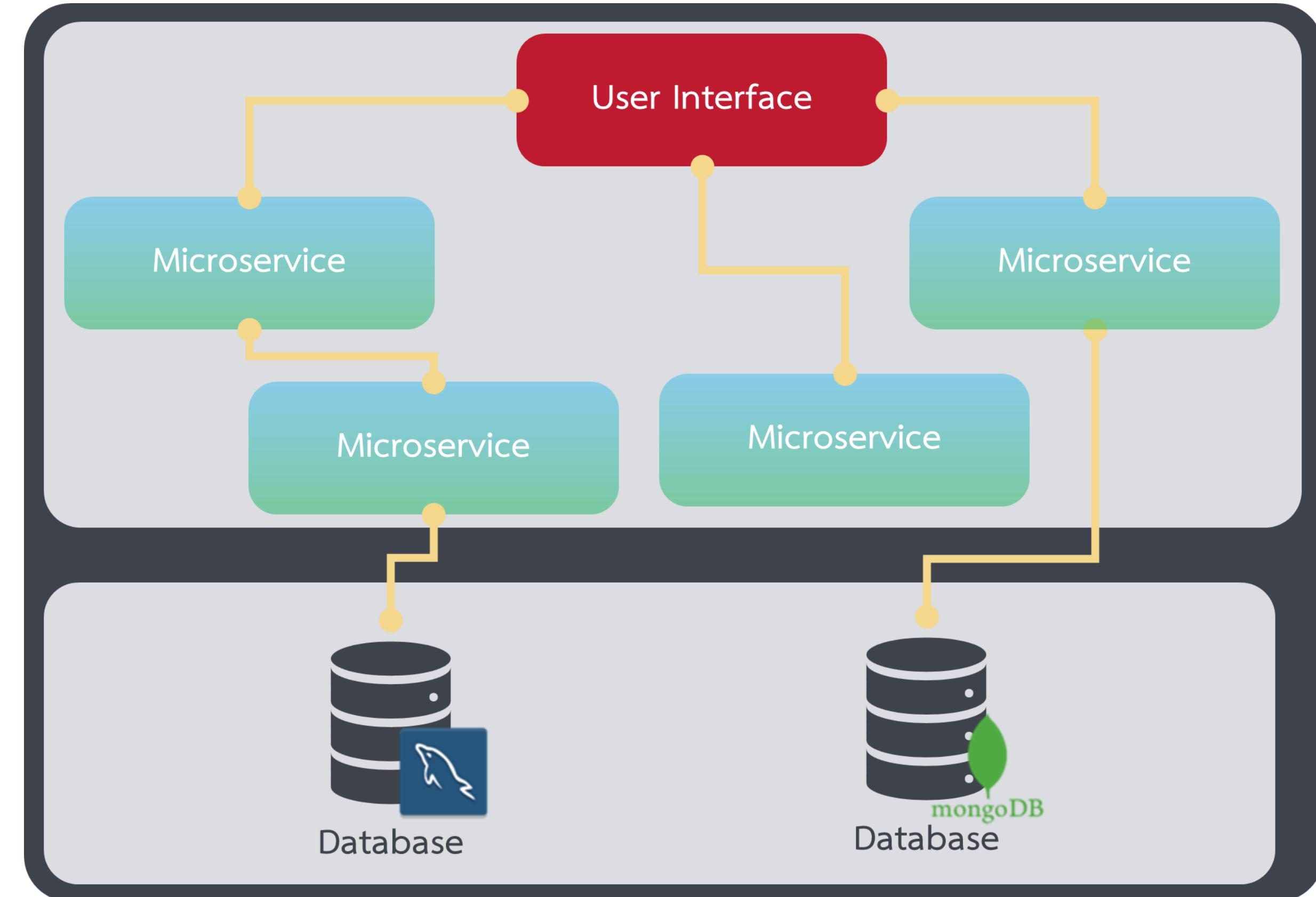
Microservice คือ แนวคิดหนึ่งในการออกแบบและพัฒนาระบบงานต่าง ๆ ที่แบ่งแยกงานต่าง ๆ ออกเป็นงานบริการ (Service) ย่อย ๆ ที่มีขนาดเล็ก โดยที่แต่ละ Service

- รวมมีการทำงานเพียงอย่างเดียว (Single Responsibility)
- ควรทำงานให้จบภายในตัวมันเอง (ด้วยตัวมันเอง)
- ซึ่งอาจจะมี Data source หรือ Storage ของตัวเอง
- ไม่ควรเกี่ยวข้องหรือผูกมัด Service ตัวใด หรือผูกมัดให้น้อยที่สุดเท่าที่จำเป็น (Loosely Couple)

ซึ่งระบบที่ได้รับการพัฒนาบนหลักการ Microservice จะเกิดจากการประกอบกันระหว่าง Service ต่าง ๆ ที่มีความเป็นอิสระต่อกัน



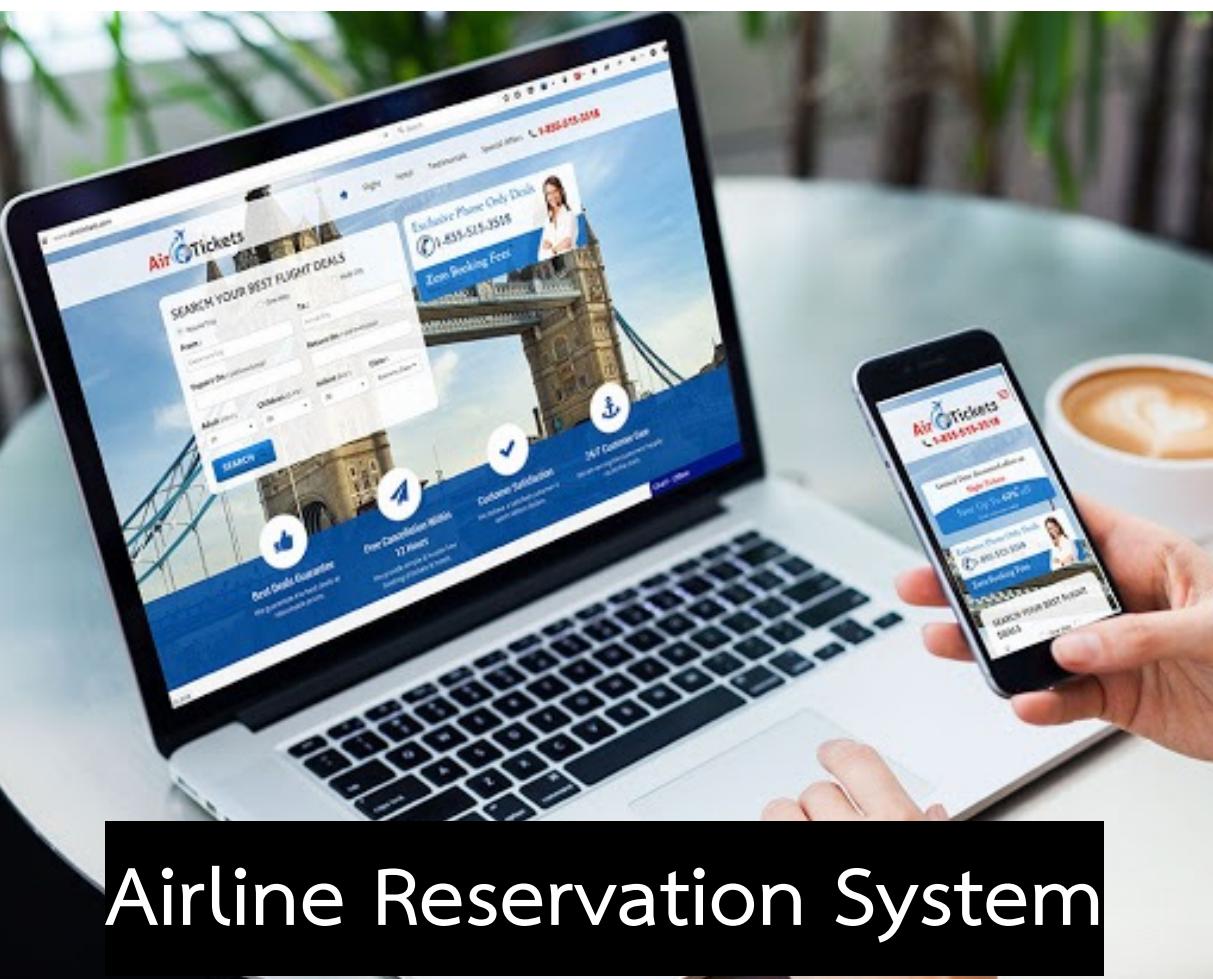
# Microservice Architecture



นอกจากนี้ งานที่ใหญ่และมีความซับซ้อนจะถูกแยกออกเป็นงานเล็กและกระจายไปให้แต่ละ Service รับผิดชอบ ส่งผลให้ Service มีขนาดเล็กและเกิดความซับซ้อนที่น้อยมาก



# ตัวอย่าง Airline Reservation System



Web or Application Server

Booking Flights

Allocate Seats

Display Prices

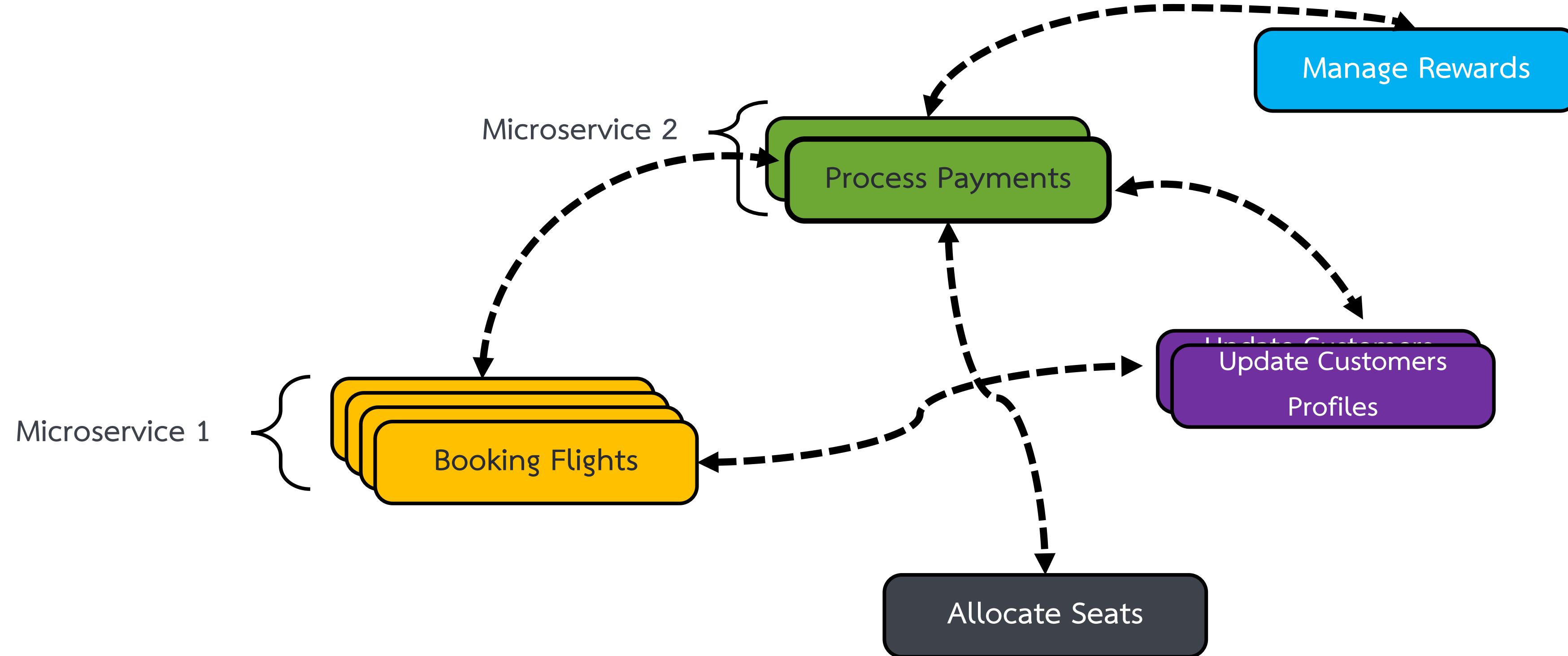
Process Payments

Update Customers  
Profiles

Manage Rewards

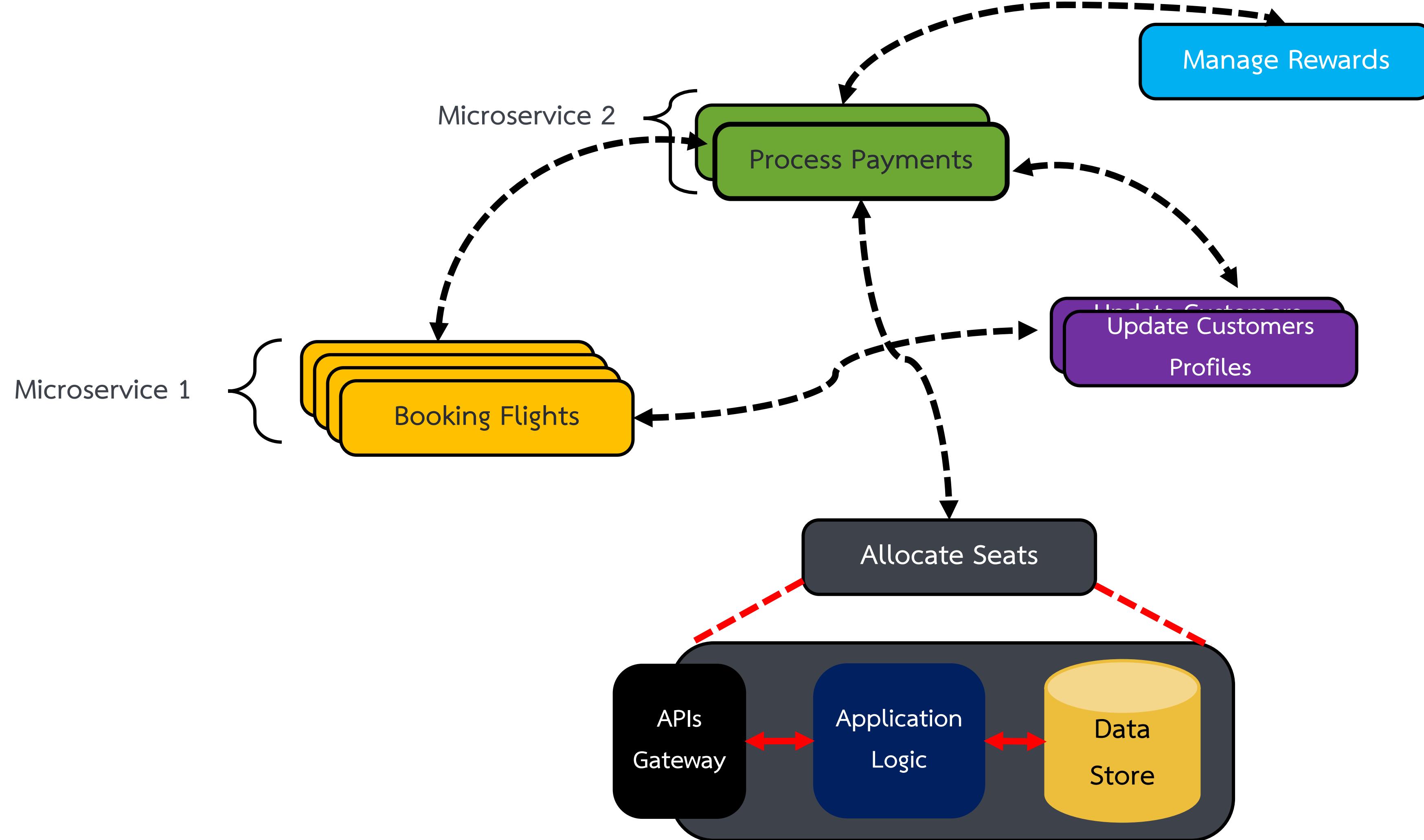


# ตัวอย่าง Airline Reservation System





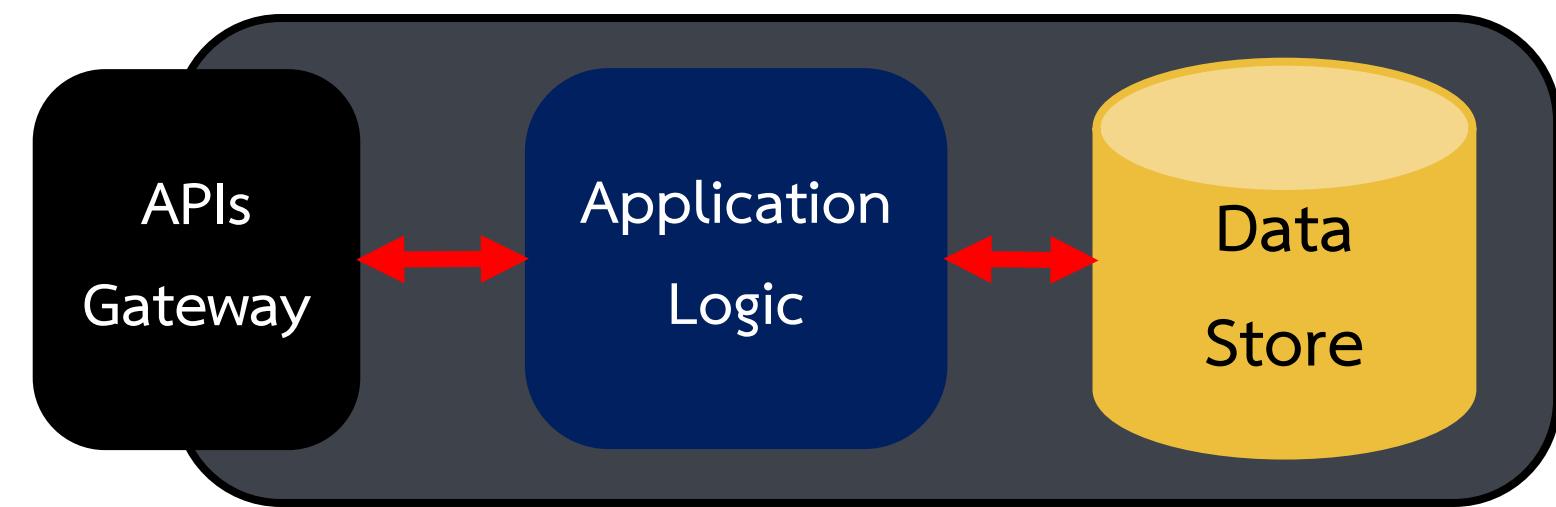
# ตัวอย่าง Airline Reservation System





# องค์ประกอบภายใน Microservice

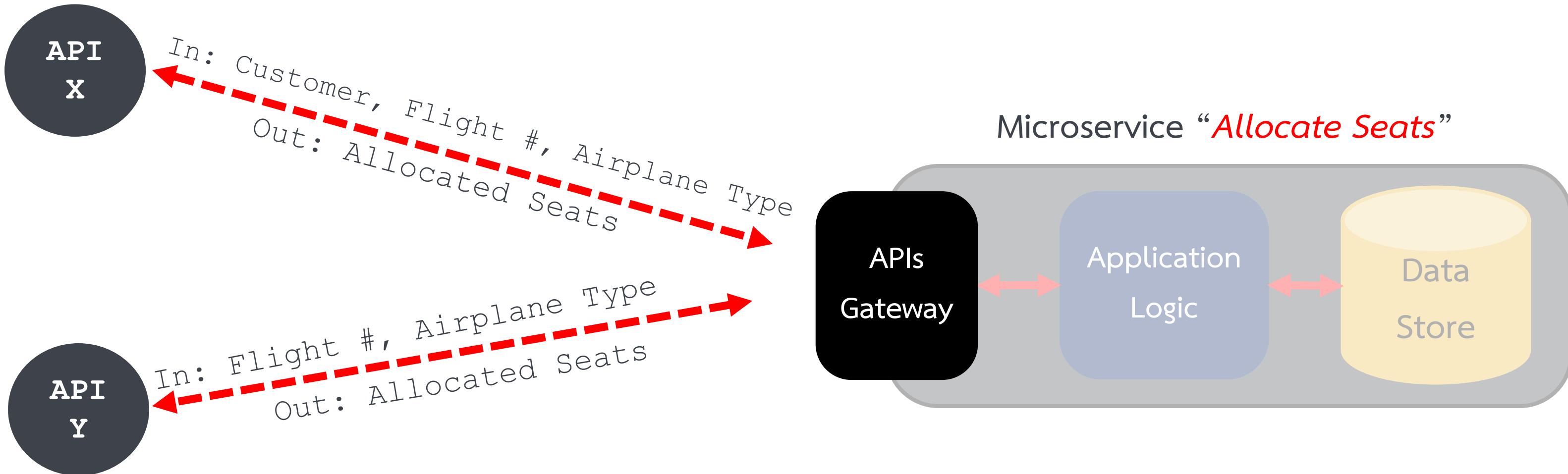
ภายใน Microservice 1 ตัว จะประกอบไปด้วย 3 องค์ประกอบหลัก ได้แก่ (1) APIs Gateway, (2) Application Logic (หรือ Business Logic) และ (3) Data Store โดยมีรายละเอียดดังต่อไปนี้





# องค์ประกอบภายใน Microservice

ภายใน Microservice 1 ตัว จะประกอบไปด้วย 3 องค์ประกอบหลัก ได้แก่ (1) APIs Gateway, (2) Application Logic (หรือ Business Logic) และ (3) Data Store โดยมีรายละเอียดดังต่อไปนี้

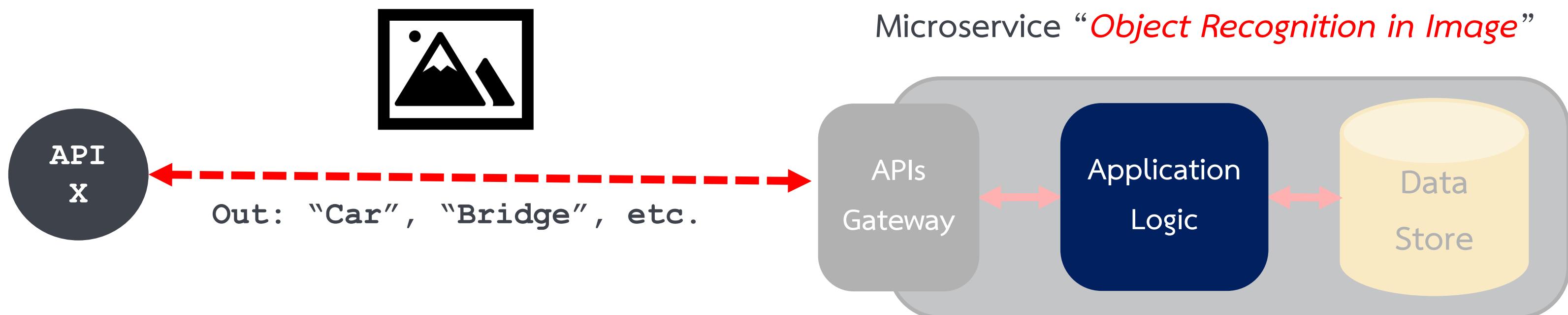


**API** คือ อินเตอร์เฟสสำหรับการเข้าถึง service นั้น ๆ ซึ่งเปรียบได้กับเป็นประตูบ้านของ service นั้น ๆ โดยมีการกำหนดกฎเกณฑ์หรือเงื่อนไขต่าง ๆ สำหรับการเรียกใช้งาน service นั้น ๆ ไว้ด้วย



# องค์ประกอบภายใน Microservice

ภายใน Microservice 1 ตัว จะประกอบไปด้วย 3 องค์ประกอบหลัก ได้แก่ (1) APIs Gateway, (2) Application Logic (หรือ Business Logic) และ (3) Data Store โดยมีรายละเอียดดังต่อไปนี้

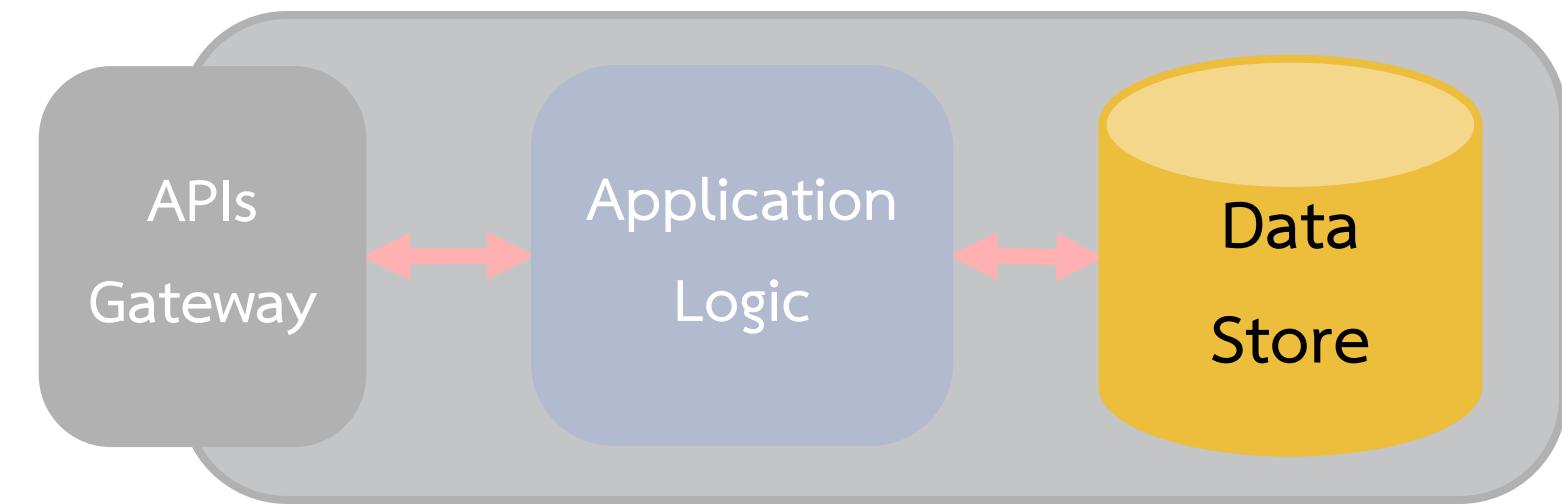


**Application Logic (หรือ Business Logic)** คือ ส่วนที่ทำหน้าที่เป็นสมองของ service ถ้าจะประมวลผลหรือทำงานบางอย่าง (“Do Something Useful”) และมีความเป็นอิสระในด้านต่าง ๆ อาทิเช่น การเลือกภาษาและเทคโนโลยีที่จะพัฒนา Service นั้น ๆ โดยเลือกจากภาษาหรือเทคโนโลยีที่ตอบโจทย์และมีประสิทธิภาพมากที่สุด



# องค์ประกอบภายใน Microservice

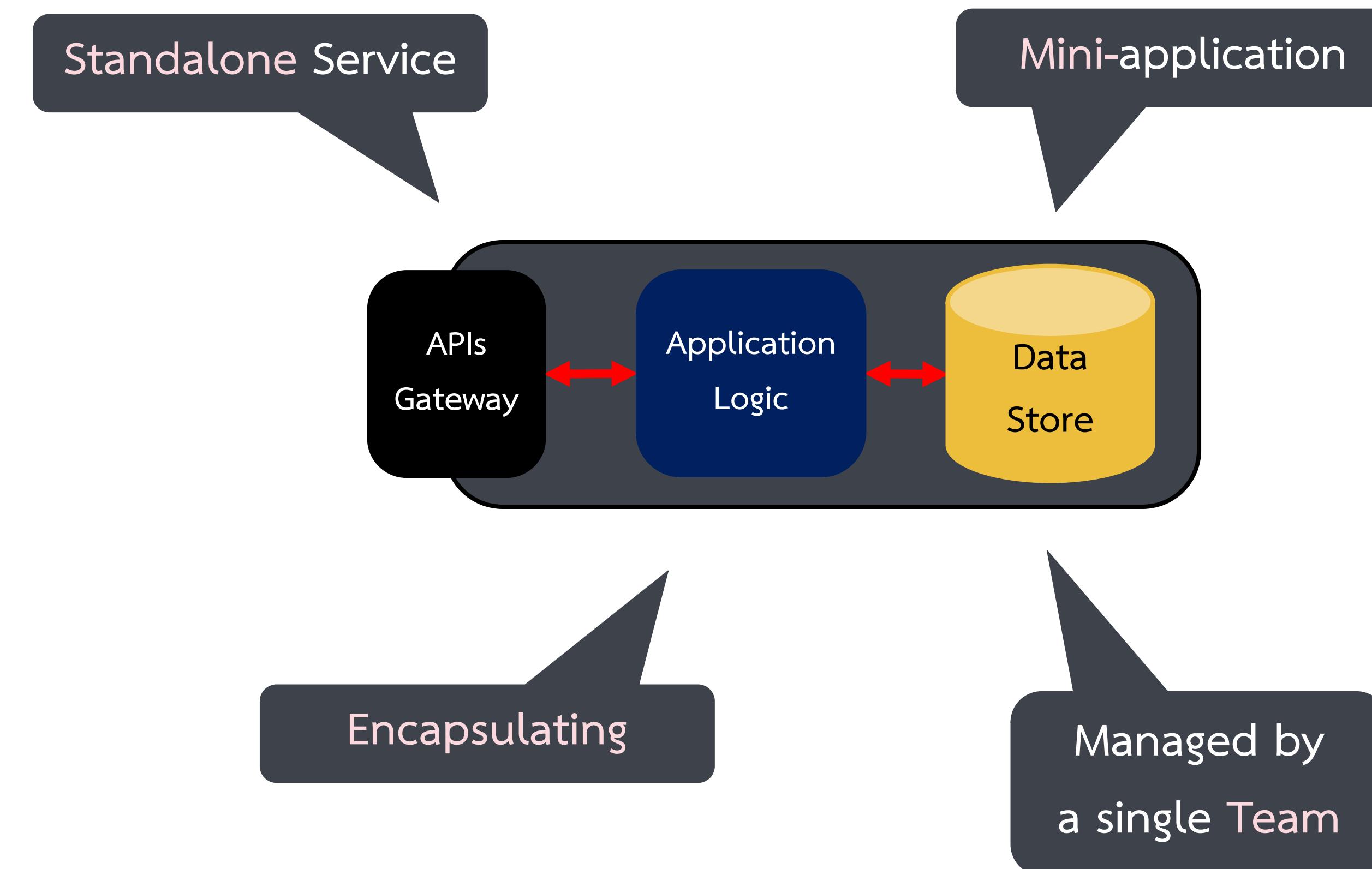
ภายใน Microservice 1 ตัว จะประกอบไปด้วย 3 องค์ประกอบหลัก ได้แก่ (1) APIs Gateway, (2) Application Logic (หรือ Business Logic) และ (3) Data Store โดยมีรายละเอียดดังต่อไปนี้



**Data Store** คือ ส่วนที่ทำหน้าที่เก็บข้อมูลหรือเรียกข้อมูลมาใช้งาน ซึ่งแต่ละ service ควรมี Data store ของตัวเองที่ได้รับการห่อหุ้มอยู่ (Encapsulation) ภายใน service นั้น ๆ หมายความว่า service อื่นไม่ควรเข้าถึง Data store ของ service อื่นได้โดยไม่ผ่าน API Gateway และไม่ควรมีการแชร์ Data store ร่วมกับ service อื่น ๆ เพื่อรับ (1) การเปลี่ยนชนิดของ Data store โดยไม่กระทบต่อ service ที่เกี่ยวข้อง และ (2) การเลือกเทคโนโลยีและชันดูฐานข้อมูลที่เหมาะสมใน use case นั้น



# องค์ประกอบภายใน Microservice





# ขนาดของ microservice ควรเท่าไหร่

การระบุขนาดของ Microservice ที่ชัดเจนอาจจะขึ้นอยู่กับ Design Pattern ที่นักศึกษาใช้ (ซึ่งจะกล่าวในบทที่ 2) แต่ก็มีแนวทางปฏิบัติร่วมกันอยู่ว่า

- ทีมที่พัฒนาต่อ 1 Microservice ควรมีขนาดเล็ก ประมาณ 6 – 12 คน (Two Pizza Boxes, Amazon)



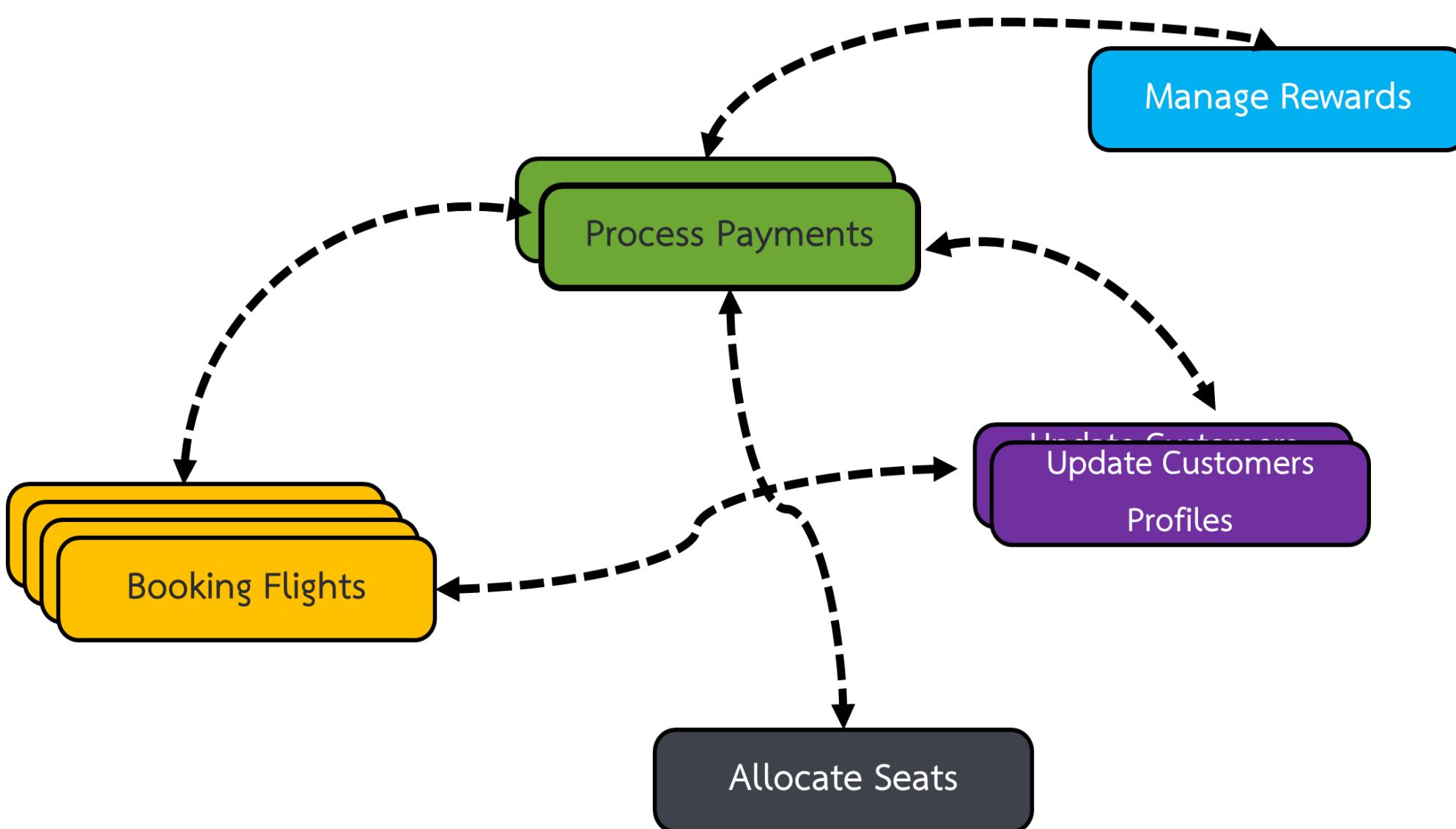
- ขอบเขตของงานไม่ซับซ้อน ชัดเจน แคบและให้ความสนใจเพียง Business Logic เดียว
- อาศัยแนวคิดที่ว่า

**“DO ONE THING AND DO IT WELL”**

ซึ่งขนาดของ Microservice ถือว่าเป็นปัจจัยหนึ่งที่ใช้แบ่งระบบขนาดใหญ่ (Monolithic Application) ออกเป็นกลุ่มของ Microservice ได้



# ความสัมพันธ์แบบหลวม (Loosely Coupled)



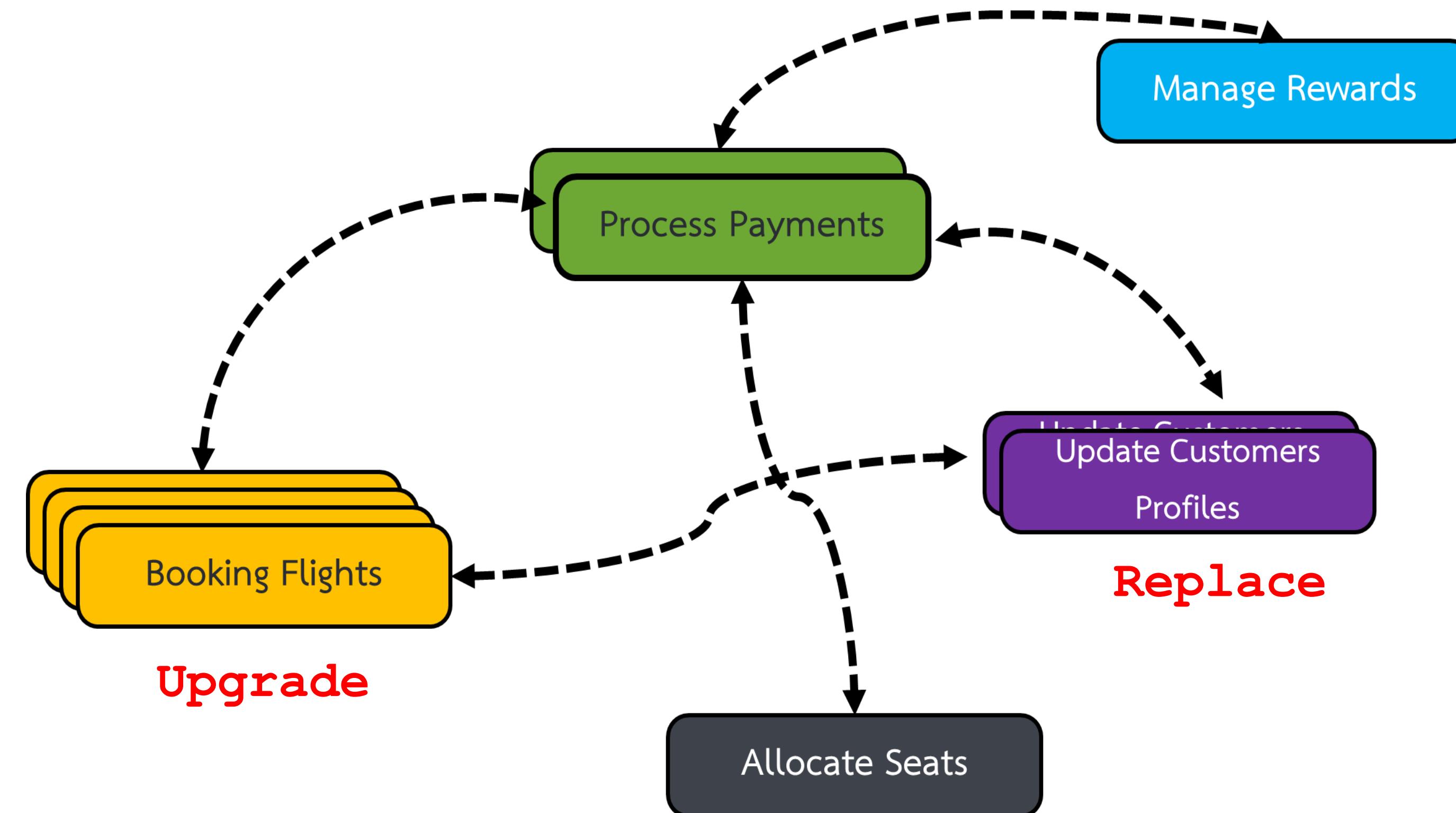
**“MICROSERVICE ARE LOOSELY COUPLED  
BETWEEN THEMSELVES”**

เพื่อ **หลีกเลี่ยง** ความสัมพันธ์ที่ผูกมัดกันค่อนข้างมาก (Tight Coupling) ดังนั้น

- ให้ออกแบบบนพื้นฐาน “Black Box” ที่ซ่อนความซับซ้อนและ Business Logic ภายใต้ตัวเองจาก Microservice อื่น ๆ ซึ่งมีขอบเขตการทำงานที่ชัดเจนระหว่าง Microservice อื่น
- การพัฒนาแต่ละ Microservice จะต้องพัฒนาขึ้นด้วยโค้ดภายในตัวเองและไม่มีการอ้างอิงถึงโค้ดจากภายนอกของขอบเขตของตนเอง
- การติดต่อระหว่าง Microservice สามารถทำได้ผ่าน API Gateways



# ลดดเปลี่ยนและอัพเกรดได้อย่างอิสระ<sup>ๆ</sup> (Independently Replaceable and Upgradable)



จากการออกแบบข้างต้นทำให้แต่ละ Microservice มีความสัมพันธ์แบบหลวม (Loosely Coupled) ส่งผลให้มีอิสระต่อการลดเปลี่ยนและอัพเกรด Microservice แต่ละตัวโดยไม่กระทบต่อ Microservice อื่น ๆ



# หนึ่ง Microservice เป็นหนึ่งสินค้า

การที่เรามองว่า หนึ่ง Microservice เป็นหนึ่งสินค้าหรือบริการจะทำให้เกิดมุ่งมองหรือแง่คิดที่ว่า

- เป็นการเชื่อม Microservice นั้น ๆ เข้ากับ Business Logic ได้จริง แทนที่จะมองว่าเป็นเพียงชุดฟังก์ชัน ซึ่งทำให้มีพัฒนาต่อเนื่องในการปรับตัวและปรับปรุง Microservice ตามแผนงานเฉพาะ (Software Roadmap)
- ซึ่ง Roadmap ที่ได้จะแบ่งแยกเส้นทางการพัฒนาของแต่ละ Microservice ออกจากกันแต่ยังคงมีการเชื่อมโยงกับข้อกำหนดทางธุรกิจอยู่ ทำให้แต่ละทีมจะสนใจเพียงการทำให้ Microservice นั้นทำงานได้ตรงข้อกำหนดอย่างถูกต้องและอัปเดตอย่างต่อเนื่องตามข้อกำหนดใหม่ ๆ ที่เข้ามา

- Microservice #1



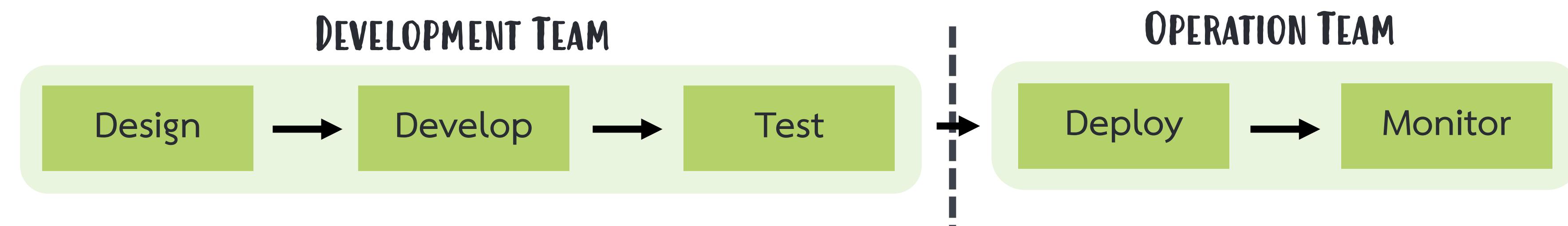
- Microservice #2



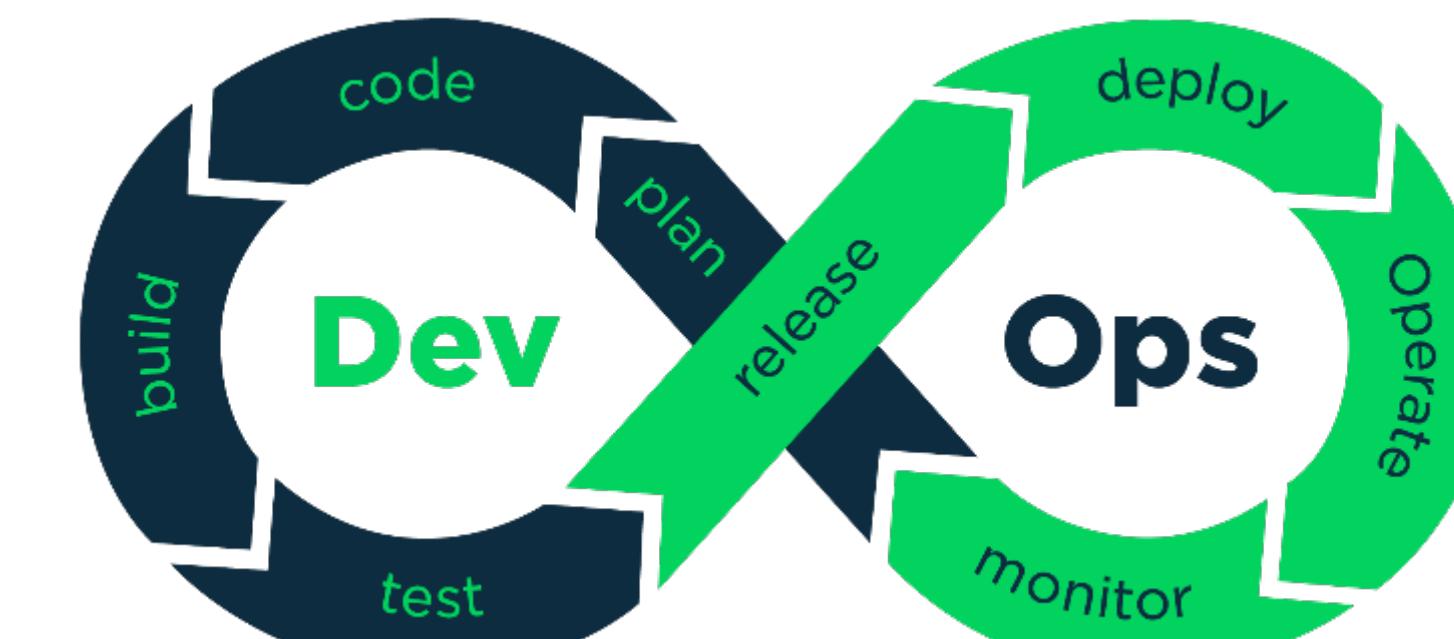


# Development & Operation (DevOpd)

ในการพัฒนาระบบแบบ Monolithic ดั้งเดิมจะประกอบไปด้วย 5 ขั้นตอน ได้แก่ Design, Develop, Test, Deploy และ Monitor ตามลำดับ โดยได้กำหนดให้ขั้นตอน Design, Develop และ Test เป็นความรับผิดชอบของทีมพัฒนา ขณะที่ Deploy และ Monitor เป็นความรับผิดชอบของทีมดำเนินการ

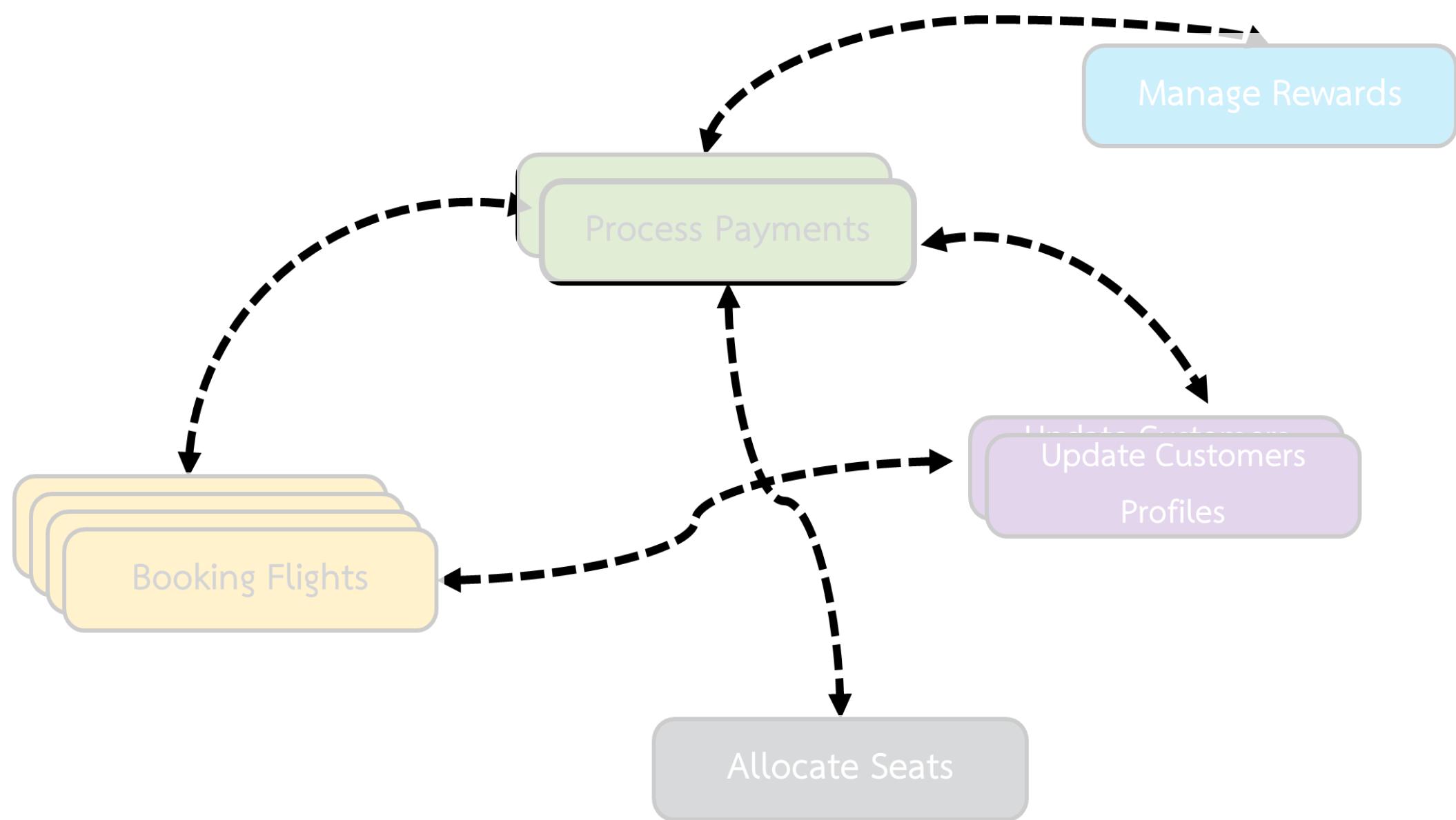


แต่การพัฒนาระบบแบบ Microservice พยายามหลีกเลี่ยงสิ่งนี้ โดยการมองว่าให้ทีมพัฒนาได้รับผิดชอบอย่างเต็มไม้เพียงแต่ขั้นตอนการพัฒนาแต่รวมถึงการดูแลที่ในสภาพแวดล้อมการใช้งานจริงภายใต้แนวความคิดที่ว่า “You build, you run it.” (Amazon) เลยเป็นที่มาของคำว่า “**DEV**ELOPMENT & **OP**ERATION” หรือเรียกว่า ๆ ว่า **DEVOPS** นั้นเอง ดังนั้น นักพัฒนารับผิดชอบและดูแล service ที่ตัวเองได้รับมอบหมายตลอดอายุการใช้งานอย่างต่อเนื่อง





# การสื่อสารระหว่าง Service



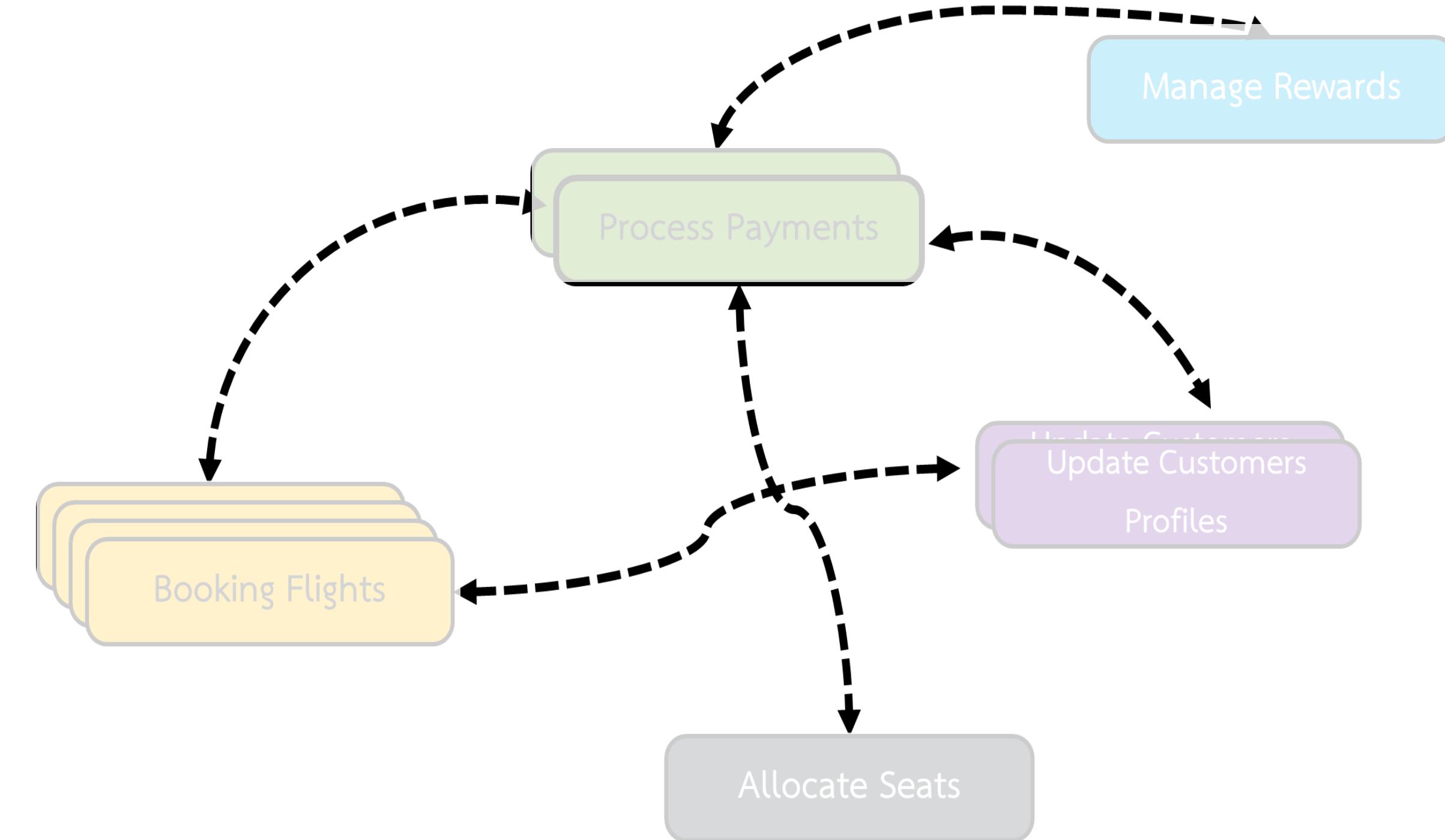
สำหรับการสื่อสารของระบบที่ได้รับการพัฒนาแบบ Monolithic แต่ละส่วนของระบบหรือโปรแกรมย่อยอาจจะทำงานอยู่ใน Process เดียวกัน หรือหลาย Process บนเครื่องแม่ข่ายเดียวกันก็ได้ ซึ่งแต่ละส่วนจะสื่อสารกัน **In-Memory** บนเครื่องแม่ข่ายเดียวกันในรูปแบบการเรียกใช้ Function ตัวอย่างเช่น Function A เรียกใช้งาน Function B

ขณะที่ การสื่อสารของระบบที่ได้รับการพัฒนาแบบ Microservice จะถูกแบ่งออกอย่างชัดเจนหรือกระจายออกไปอยู่บนหลากหลายเครื่องแม่ข่าย การสื่อสารระหว่าง Service เพื่อแลกเปลี่ยนข้อมูลหรือร้องขอการประมวลผลบางอย่างสามารถทำได้ โดยอาศัย **Message Exchange Pattern** (จะกล่าวในบทที่ 3) ในเบื้องต้นจะมี 2 ประเภทหลัก ได้แก่

- Request – Response
- Observer with a Message Broker



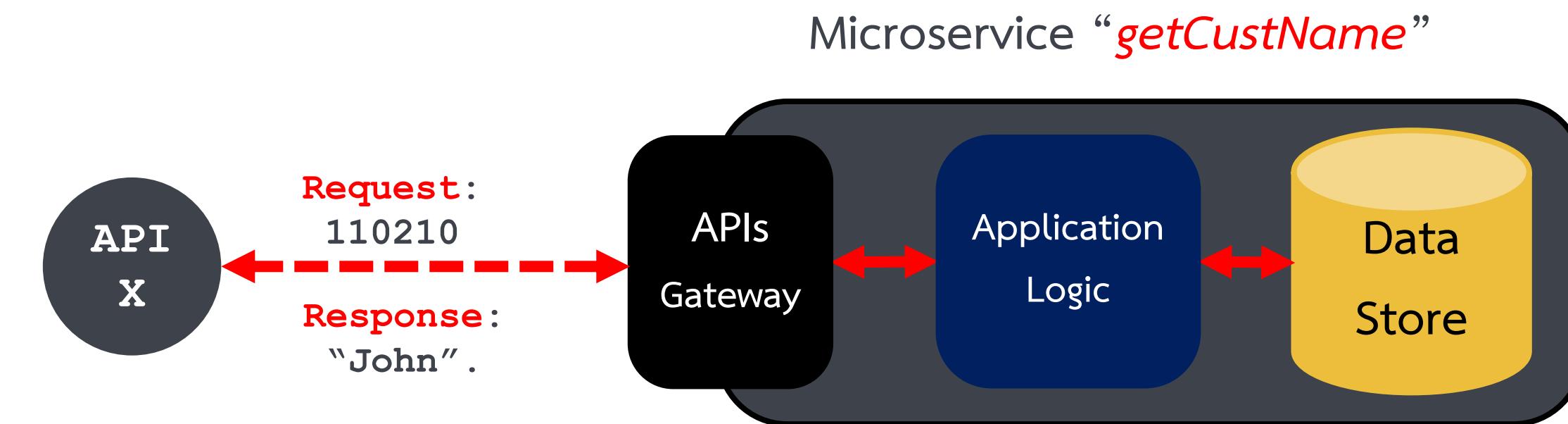
# การสื่อสารแบบ Request – Response



คือ การสื่อสารรูปแบบหนึ่งที่มีการทำงานคล้ายกับ Client – Server สำหรับการท่องเว็บไซต์ผ่าน Web Browser ซึ่งแต่ละ service สามารถร้องขอข้อมูลหรือการประมวลผลจาก service อื่น ๆ ได้ในรูปแบบการสื่อสารแบบกระจาย (Decentralized Communication) โดยอาศัย ท่อการสื่อสารแทนการถ่ายโอน input และ output ระหว่าง service



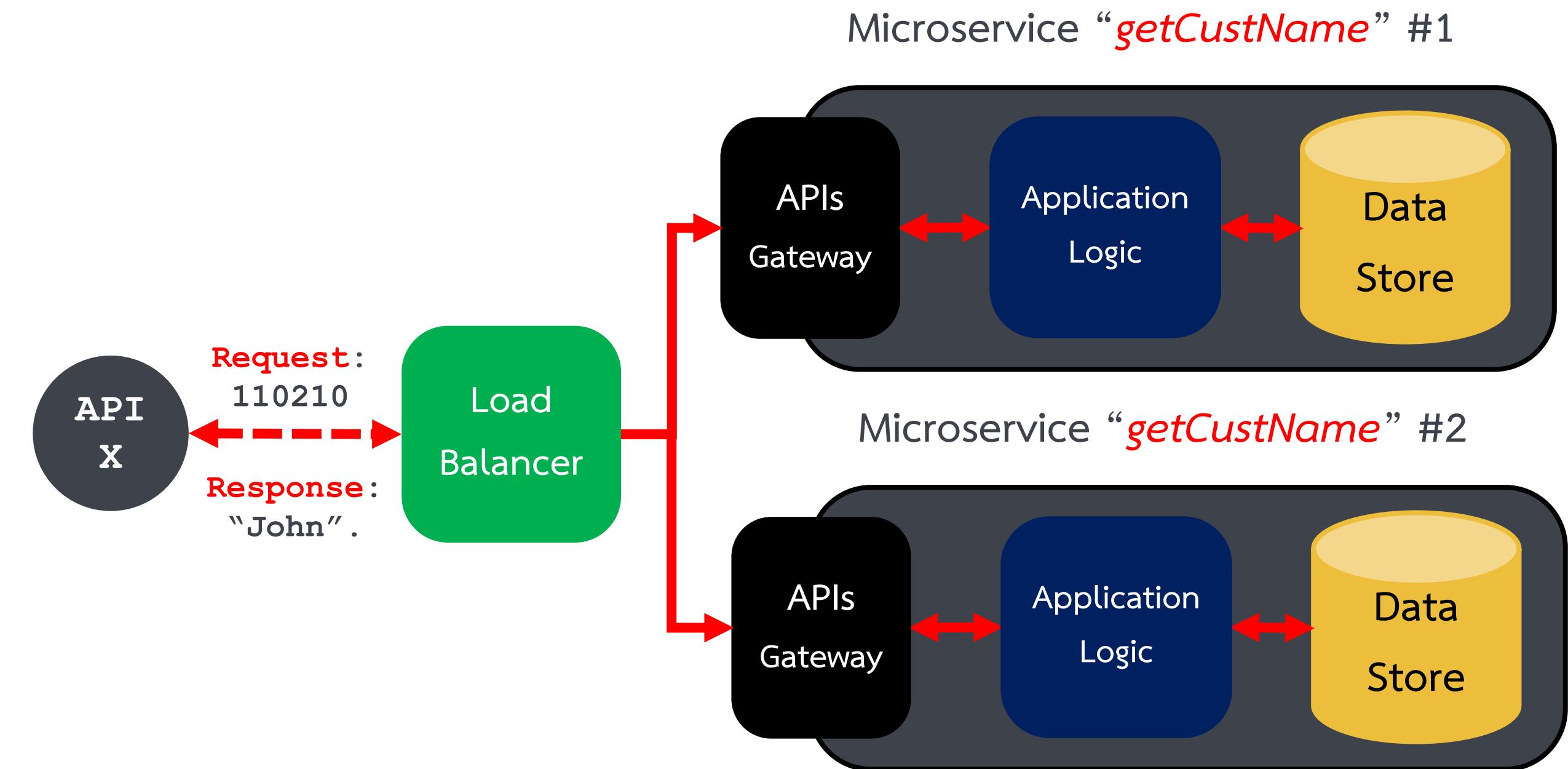
# การสื่อสารแบบ Request – Response



การสื่อสารแบบ Request – Response จะแบ่งออกเป็น 2 ฝั่ง ได้แก่ Sender และ Receiver ตามลำดับ โดยที่ Sender จะเป็นคนส่ง Request ไป จากนั้น จะทำการรอจนกว่าปลายทางจะ Response ตอบมาถึงไปทำงานอย่างอื่นต่อได้



# การสื่อสารแบบ Request – Response

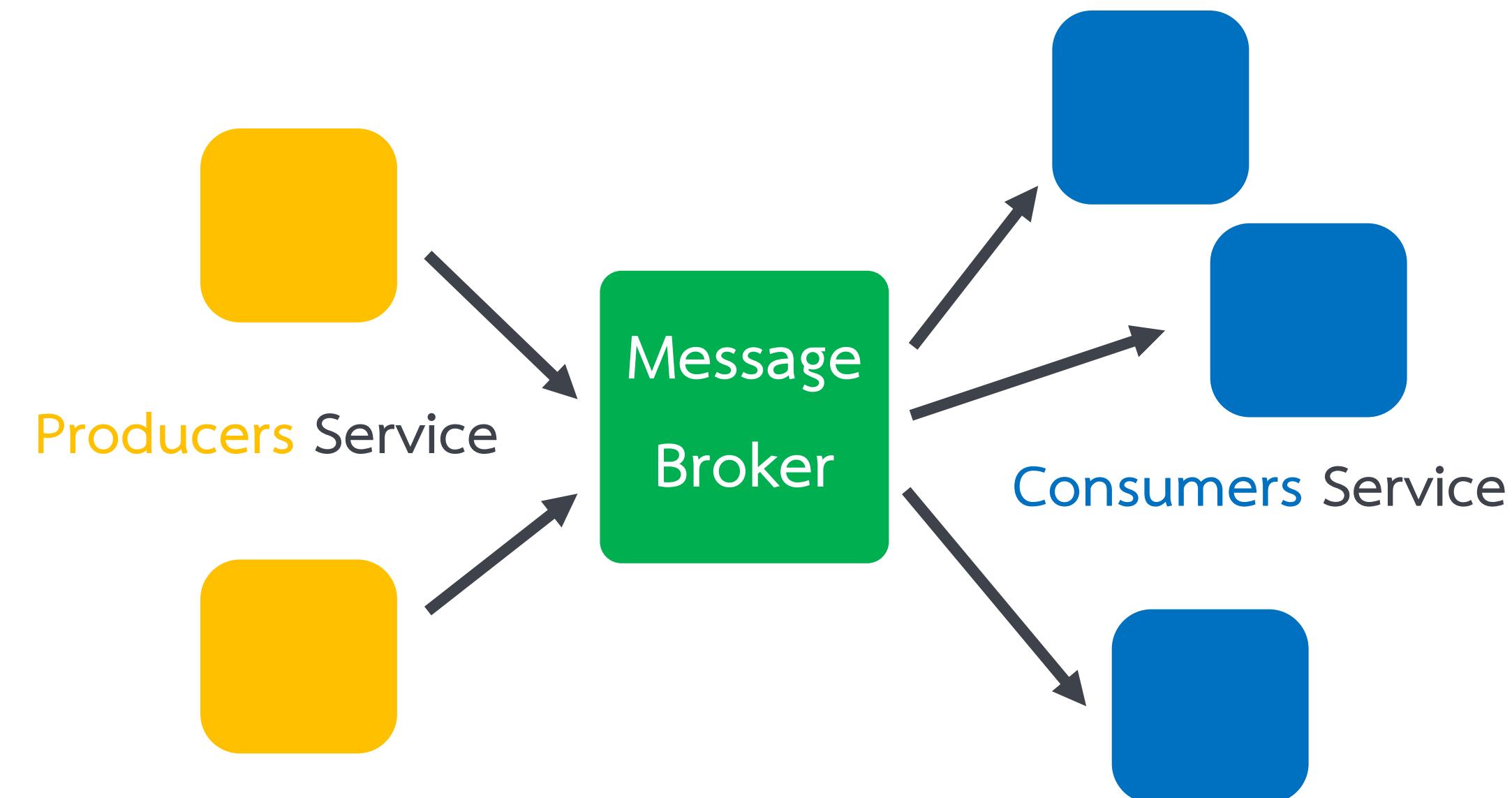


แต่ในบางกรณี นักศึกษาอาจใช้งาน Load Balancer เพื่อนำมาช่วยจัดการกับเหตุการณ์ High Traffic Load ได้ ซึ่งสามารถกระจายงานให้กับ instances ของ Microservice นั้น ๆ ได้



# การสื่อสารแบบ Observer with a Message Broker

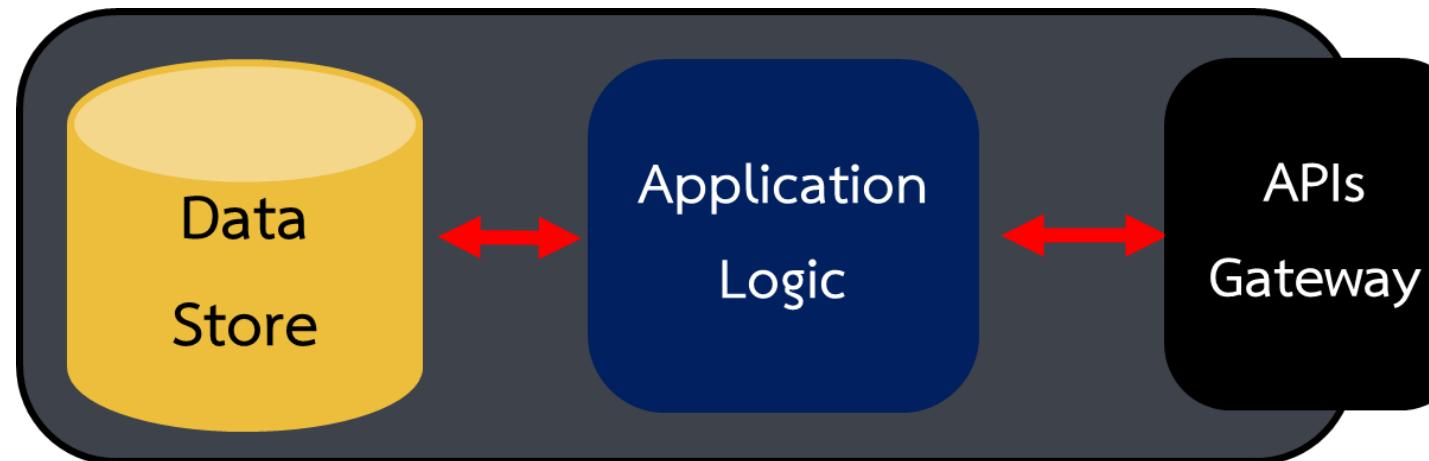
Observer Pattern นิยมนำมาใช้จัดการกับเหตุการณ์ในรูปแบบการกระจาย (distributed event handling system) โดยอาศัยสิ่งที่เรียกว่า Message Broker ให้ทำหน้าที่สร้างห่อเชื่อมต่อ data pipeline ระหว่าง (1) producers (หรือ data sources) ที่เป็นผู้สร้าง Message เข้ากับ (2) consumers ที่จะค่อยการมาถึงของ Message ใหม่ จาก data pipeline ตัวอย่าง Message Broker ได้แก่ Apache Kafka (อยู่บนหลักการ Event Driven) หรือ RabbitMQ



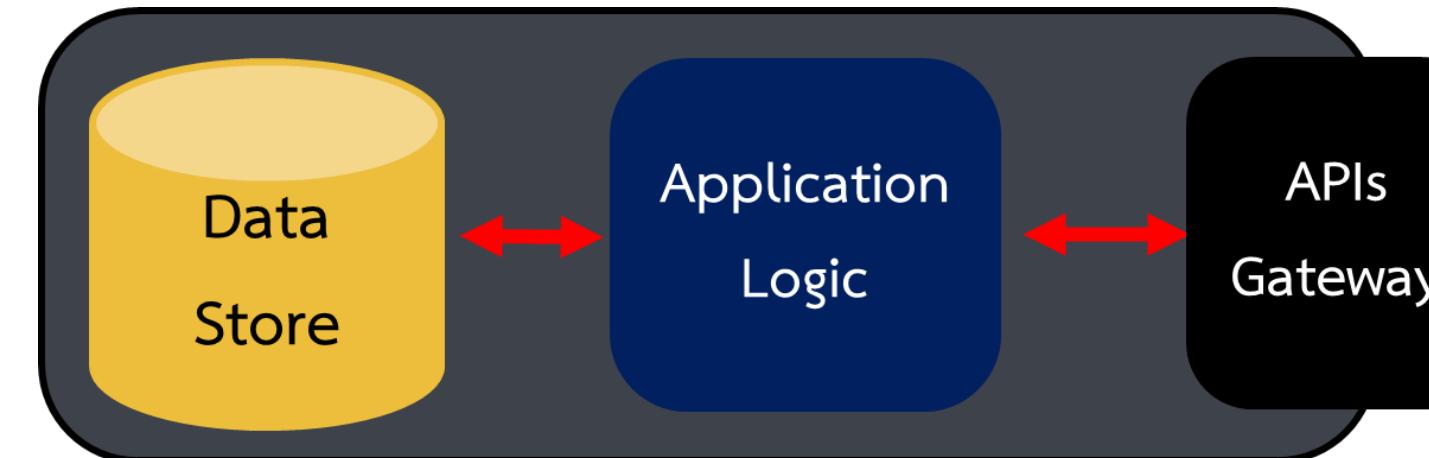


# การสื่อสารแบบ Observer with a Message Broker

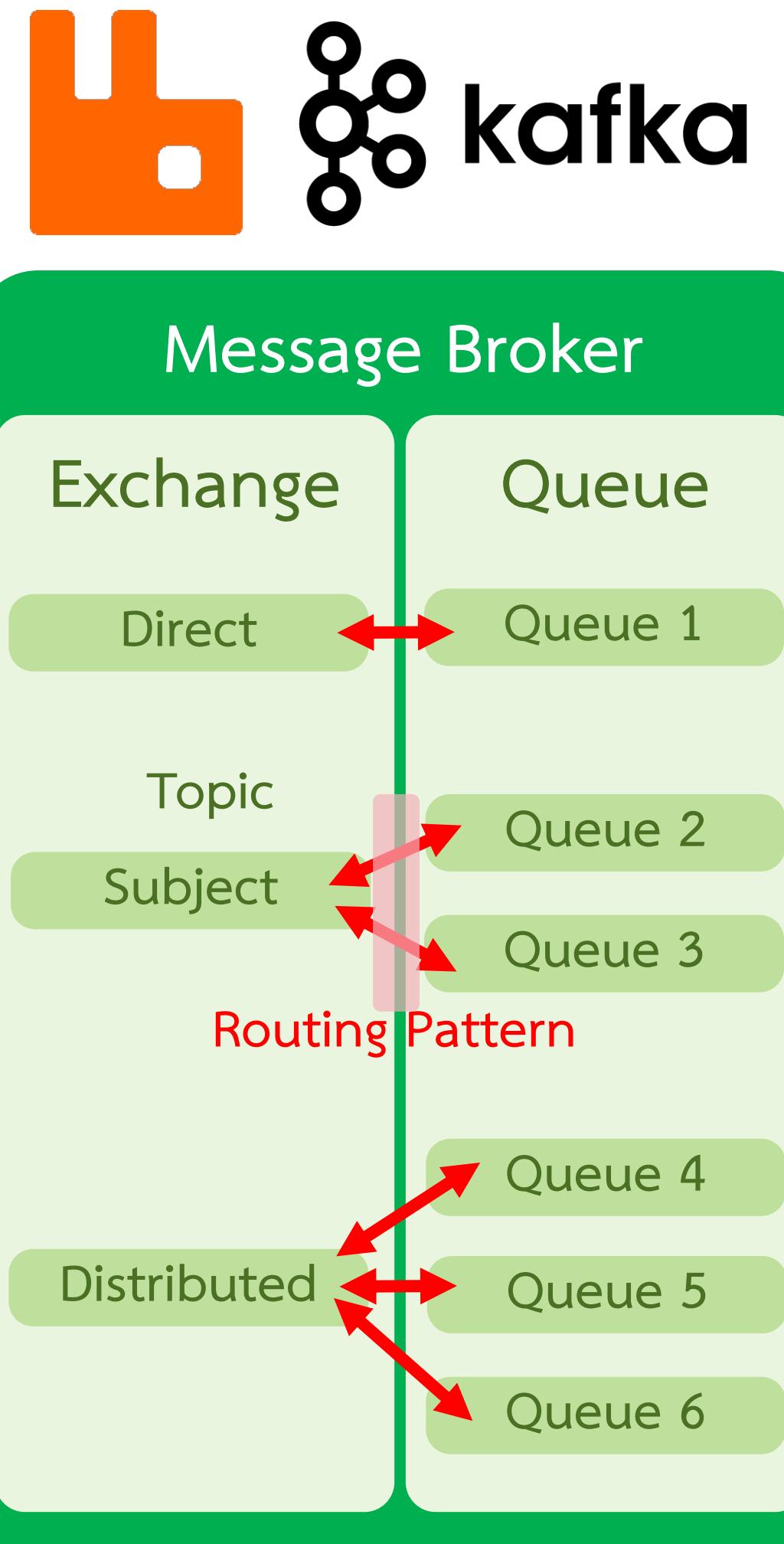
Producers Microservice #1



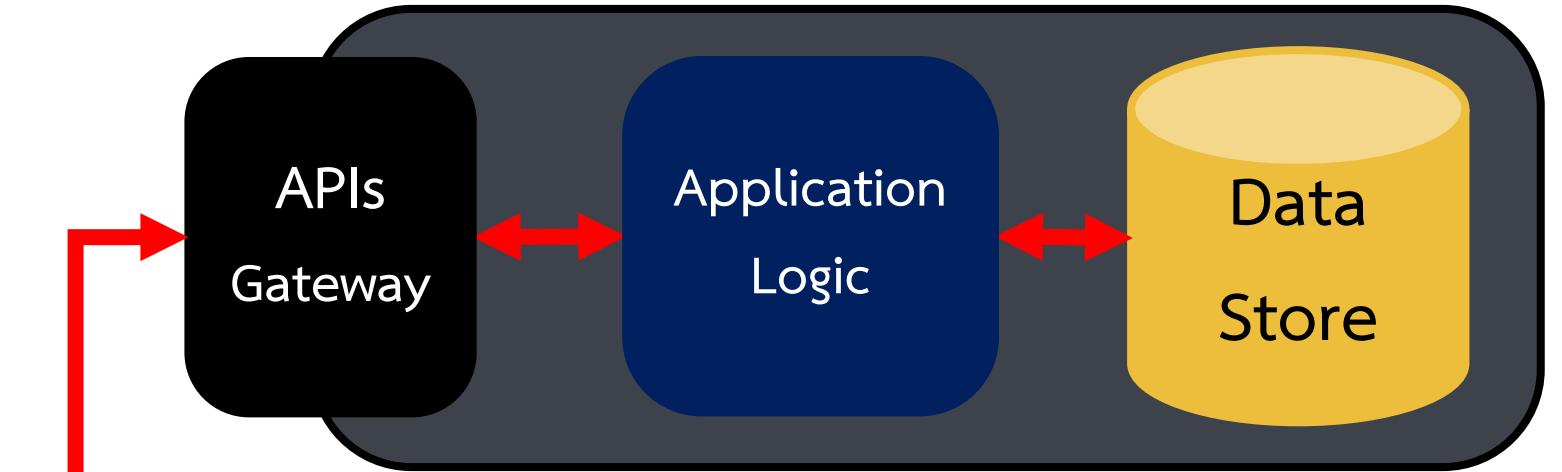
Producers Microservice #2



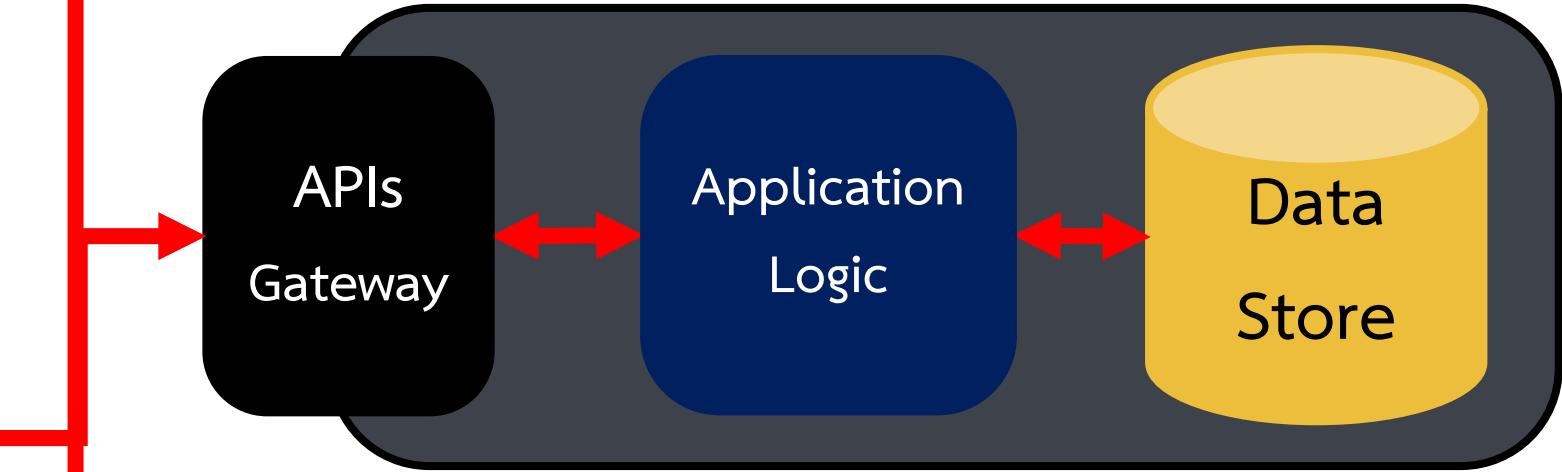
- หนึ่ง Message ส่งไปหนึ่ง Consumers  
(หลาย Instance มีการทำ Load Balancer)
- หนึ่ง Message ส่งไปมากกว่าหนึ่ง Consumers



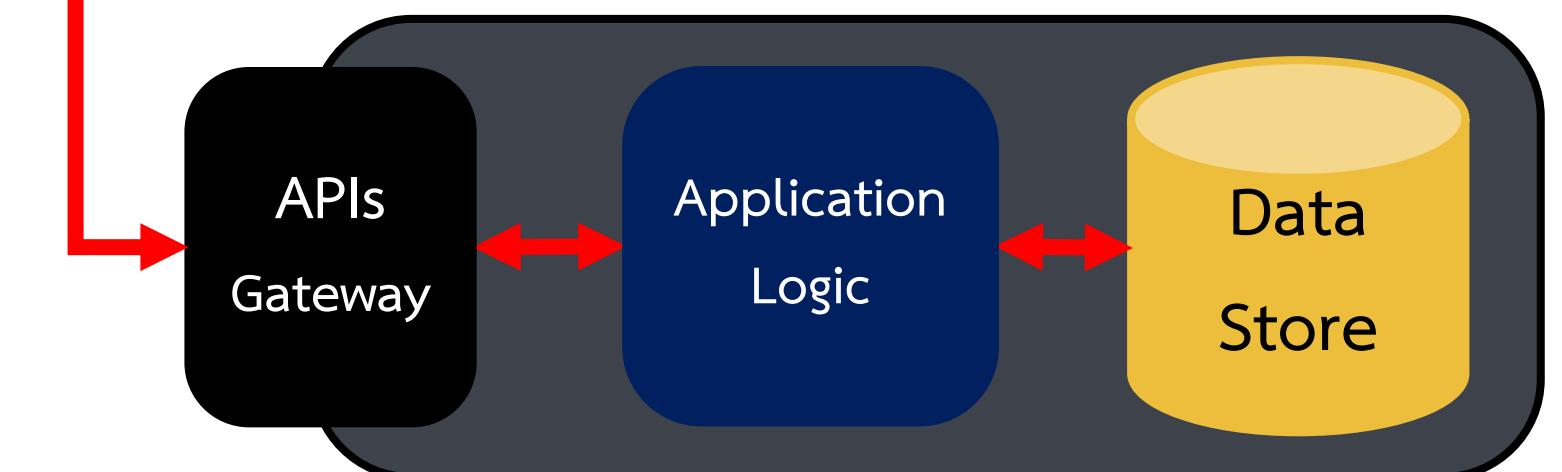
Consumers Microservice “getCustName” #1



Consumers Microservice “getCustName” #2



Consumers Microservice “getSeatID”





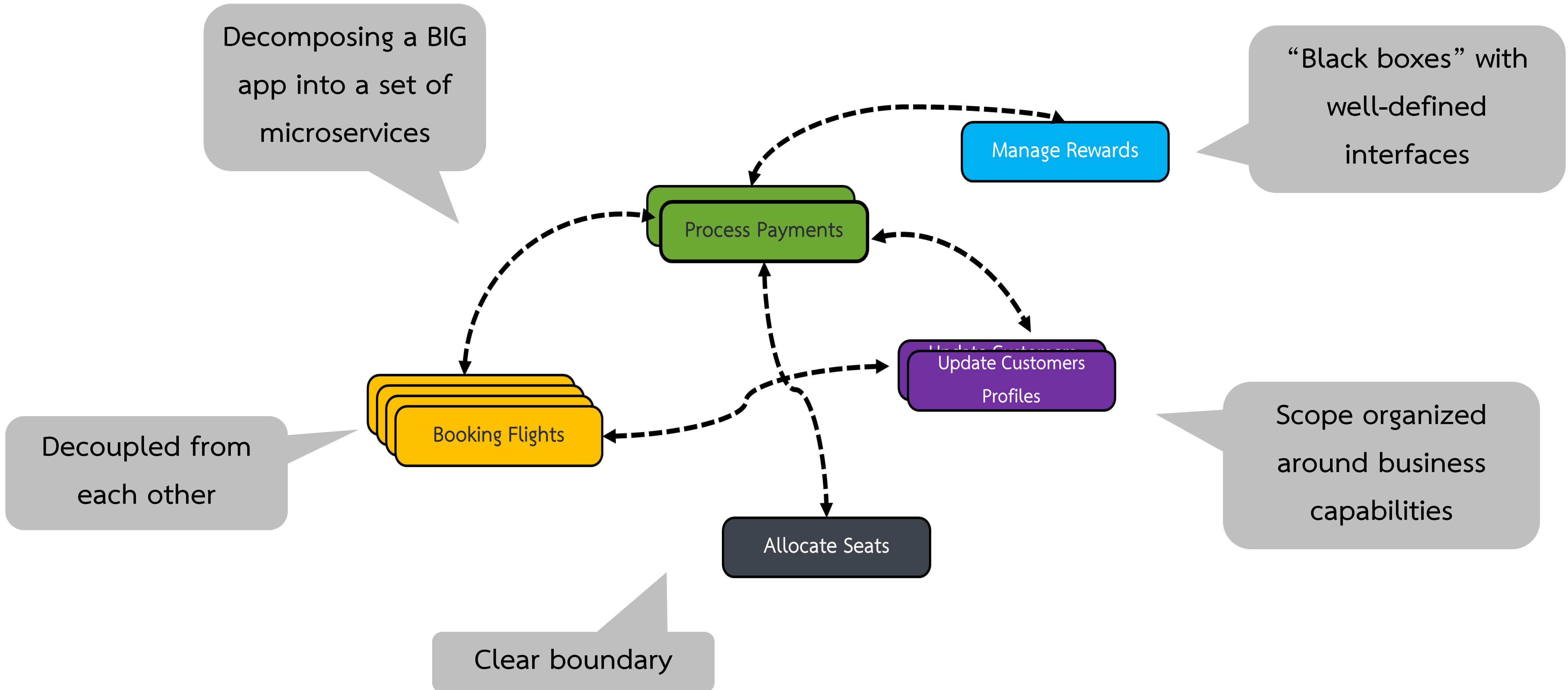
# การสื่อสารแบบ Observer with a Message Broker

## ข้อดีของการสื่อสารแบบ Observer with a Message Broker

- **producers** ไม่จำเป็นต้องรู้ว่าใครเป็นคนจัดการ Message (**consumers**) ปล่อยให้เป็นหน้าที่ของ **message broker** เป็นตัวจัดการ ขณะที่ การสื่อสารแบบ Request – Response จำเป็นต้องทราบว่าใครคือผู้ส่งและผู้รับ
- **message broker** จะเป็นตัวจัดการ messages ให้ กล่าวคือ ถ้ามี messages เข้ามาแต่ service ที่เป็น **consumers** ยังไม่ว่า **message broker** จะย้าย message ไปอยู่ใน queue ให้ก่อนจนกว่า **consumers** จะว่าง สิ่งนี้ทำให้ **producers** ไม่ต้องจัดการความผิดพลาดและการส่ง message ใหม่
- การเพิ่ม **consumers** ประเภทใหม่จะไม่กระทบต่อผู้ส่ง **producers**
- **consumers** ไม่ต้องค่อยถาม **message broker** อญูตตลอดเวลาว่า “มี messages ใหม่ที่เกี่ยวข้องมาหรือไม่” ถ้ามี **message broker** จะแจ้งเตือนมาที่ **consumers** โดยอัตโนมัติ



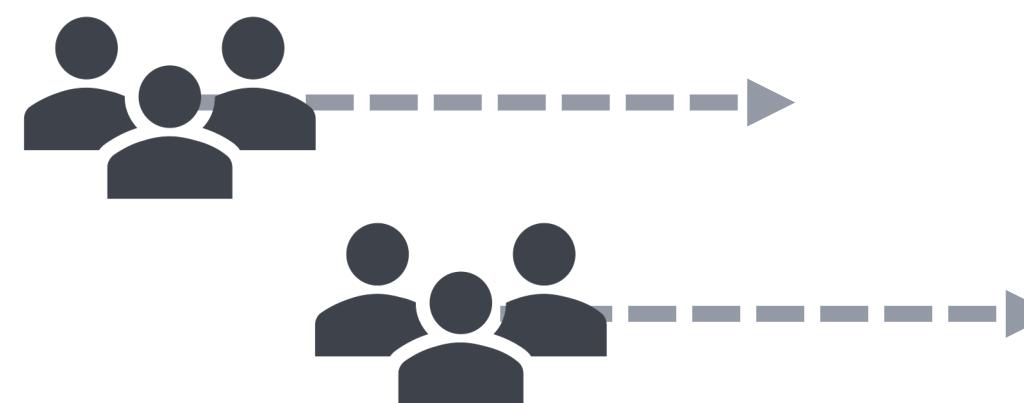
# Microservice Architecture



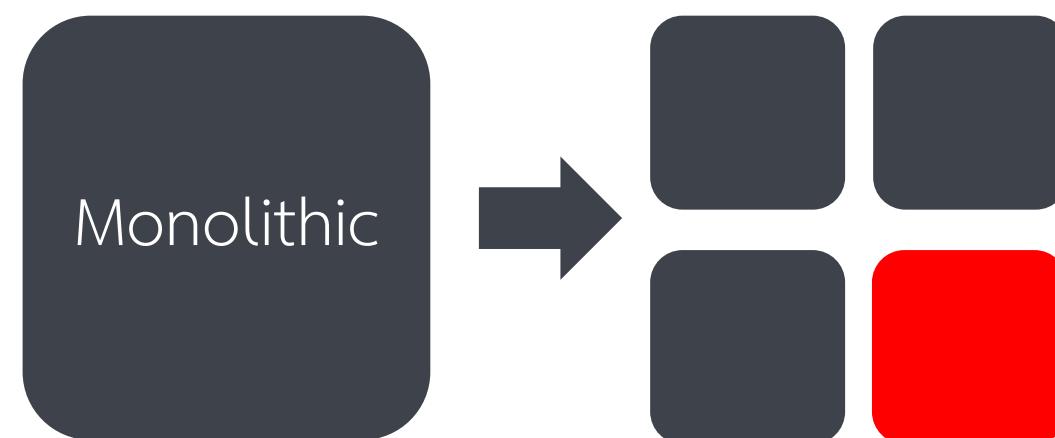


# ประโยชน์ของการออกแบบ Microservice

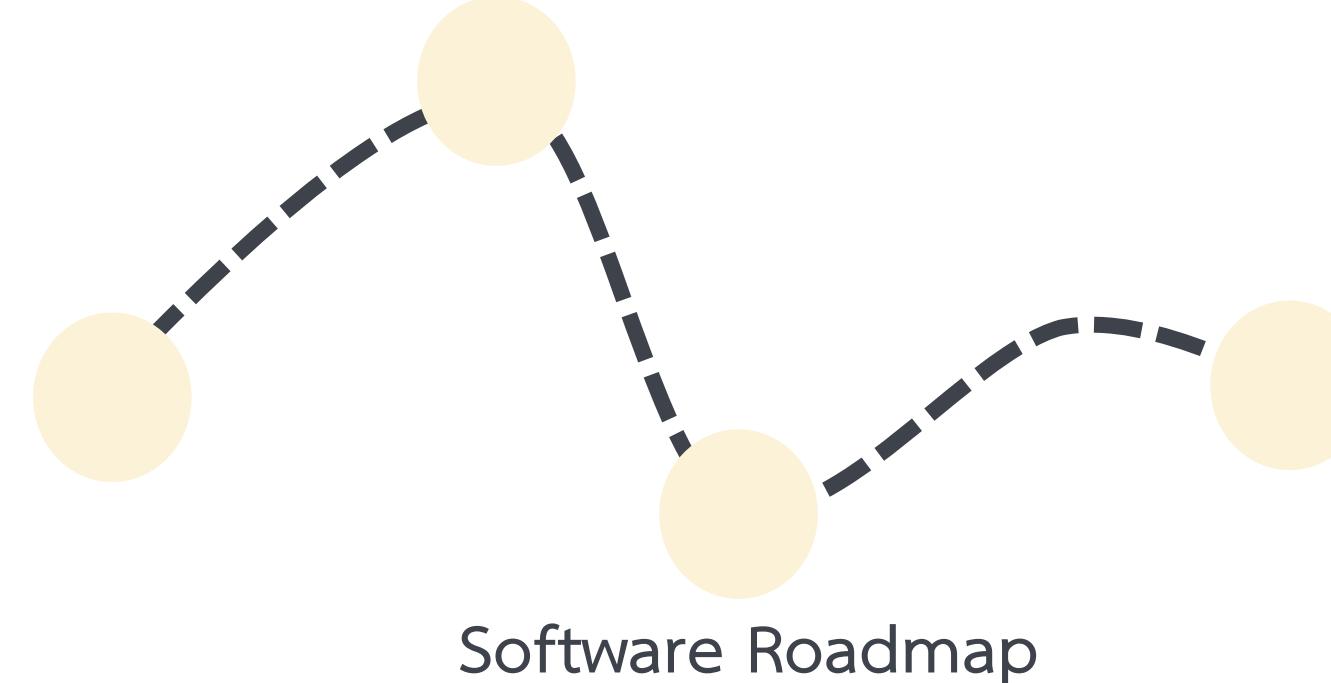
## (1) ความเป็นอิสระในการ Develop และการ Deploy



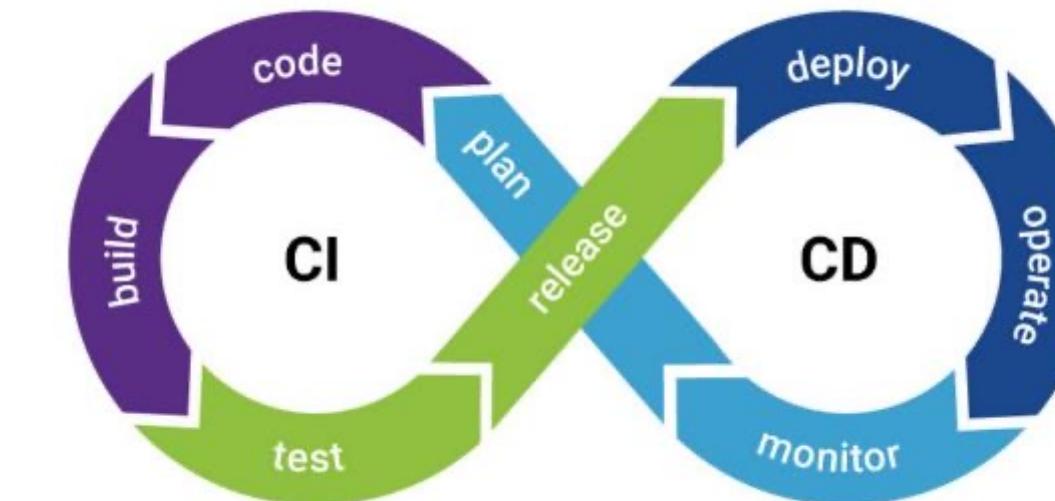
ทีมต่าง ๆ สามารถทำงาน**แบบขนานได้**



นักพัฒนาโฟกัสเพียงงานเล็ก ๆ ไม่กี่งาน (**Small components**)



แต่ละทีมสามารถมี **Software Roadmap** เป็นของตัวเอง



อ้างอิง <https://journeyofquality.com/2020/11/20/23-ci-cd-tools/>

สามารถพัฒนา ทดสอบ และตรวจสอบได้อย่างรวดเร็ว



# ประโยชน์ของการออกแบบ Microservice

## (2) การ Delivery ได้อย่างรวดเร็วและเป็นประจำ

- **Delivery**

คือ การนำ Code ที่ผ่านการตรวจสอบ (Review) และไปยัง Git ที่ทำหน้าที่เป็น Version Control โดยสิ่งที่จัดเก็บบน Git จะเรียกว่า Repository หรือสามารถกล่าวได้ว่า การที่พร้อมจะนำ Code ไป Deploy ที่ Production ได้

- **Deployment**

คือ การนำ Code จาก Repository ที่ผ่านการ Review และไป Deploy บน Production



อ้างอิง <https://blog.cloudhm.co.th/ci-cd/>

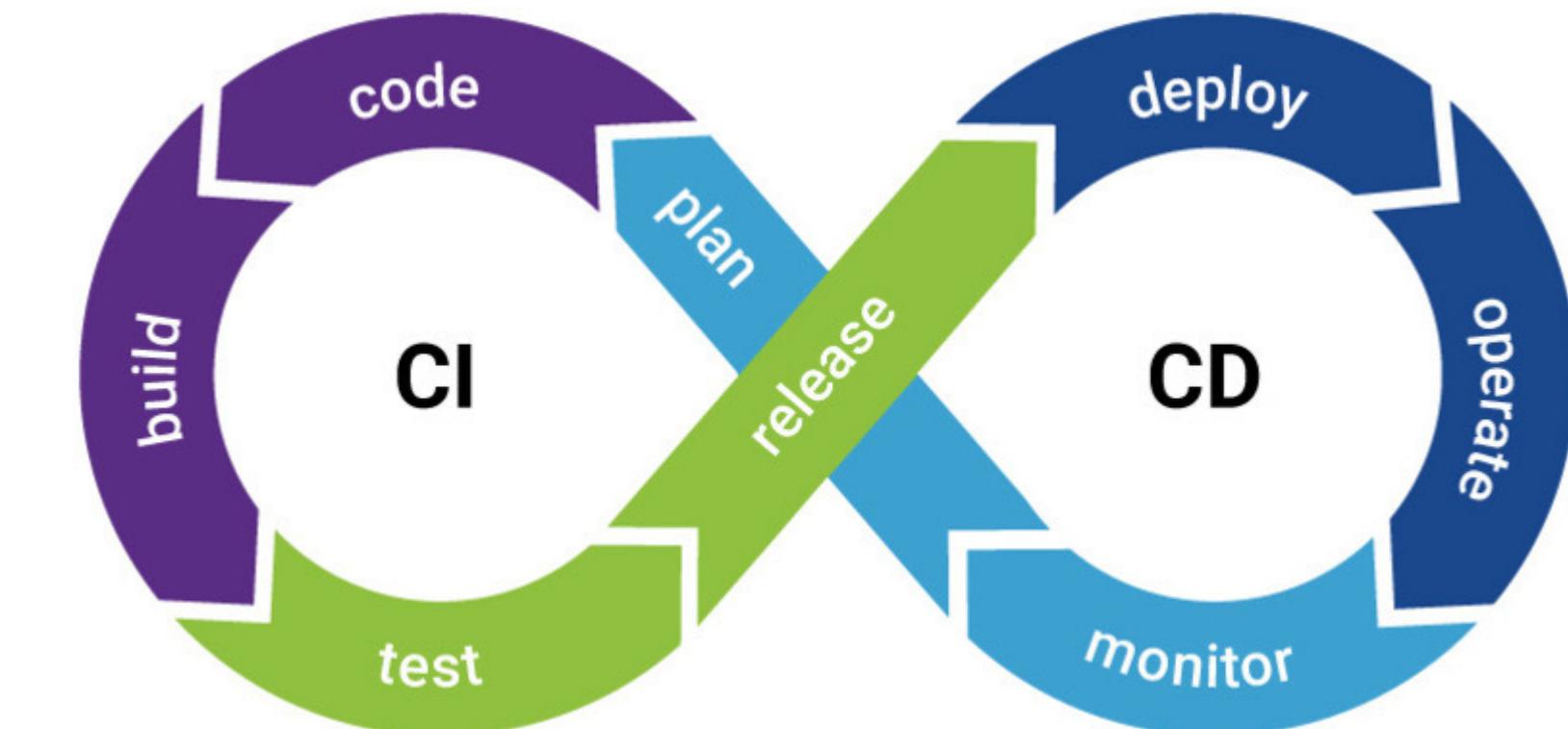


# ประโยชน์ของการออกแบบ Microservice

## (2) การ Delivery ได้อย่างรวดเร็วและเป็นประจำ

เนื่องจากแต่ละ Service ได้พัฒนาจากทีมขนาดเล็กและมีปริมาณงานไม่มากทำให้การ Integration และ Delivery ได้อย่างอิสระ รวดเร็ว และต่อเนื่อง โดยอาศัยแนวคิด Continuous Integration / Continuous Delivery Pipeline ส่งผลให้แต่ละ service มี version ที่แตกต่างกันได้อาทิเช่น

- Service Process Payments มี version 2
- Service Booking Flights มี version 7
- Service Update Customers Profiles มี version 4



อาจอ้าง <https://journeyofquality.com/2020/11/20/23-ci-cd-tools/>

นอกจากนี้ ทำให้ (1) รับ feedback ได้เร็ว (2) ชี้วัดต้นทุนและประสิทธิภาพของทีมได้ (3) Deploy เฉพาะ Service ที่แก้ไขได้

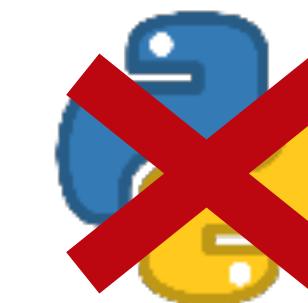


# ประโยชน์ของการออกแบบ Microservice

## (3) เลือกเทคโนโลยีได้เหมาะสมกับงาน

### Monolithic Application

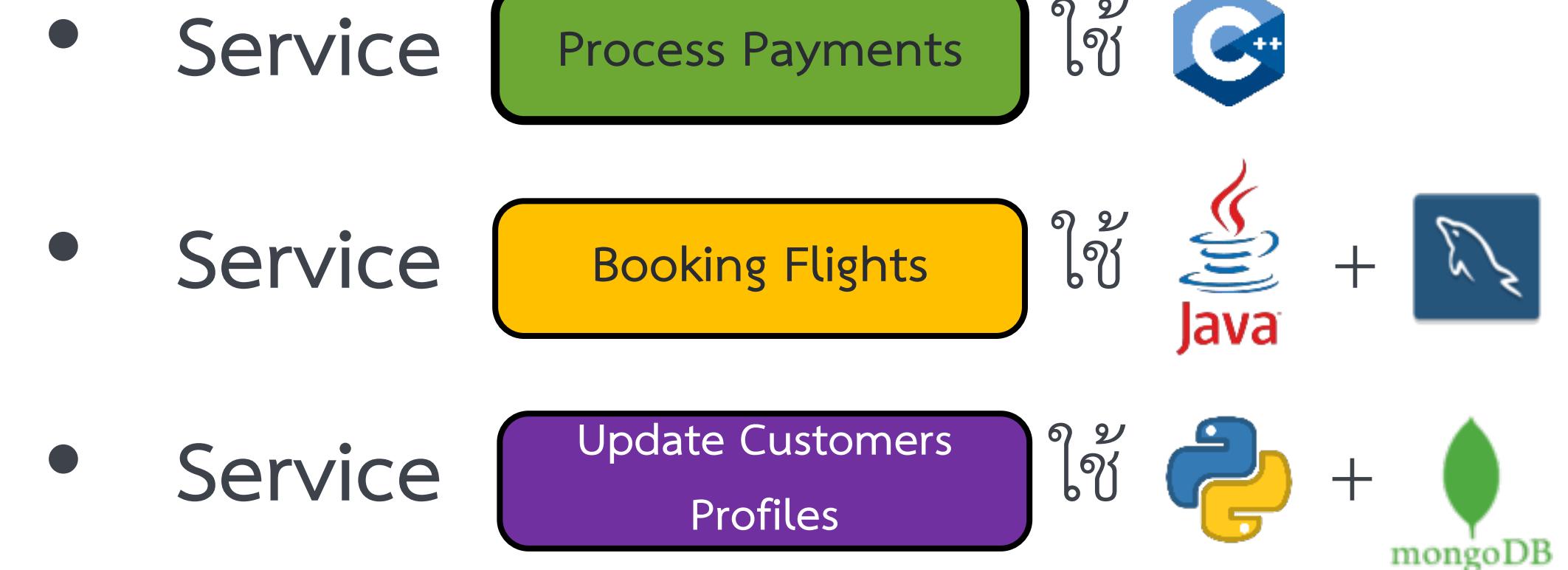
- Code – ใช้เพียงภาษาเดียวในการพัฒนา



- Balance - ระหว่างความสามารถของเทคโนโลยี และความปราณາของระบบ

### Microservice Application

- Code – ใช้หลากหลายภาษาในการพัฒนา
- สามารถเลือกใช้เทคโนโลยีให้เหมาะสมกับ Business Logic ของแต่ละ Service เช่น



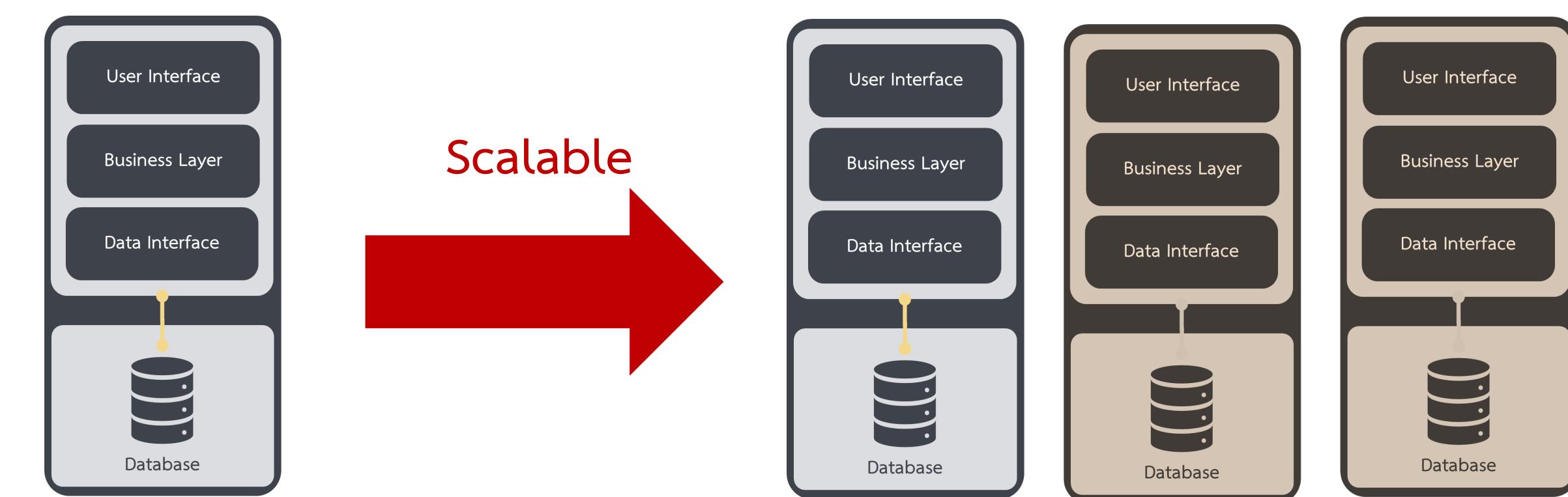


# ประโยชน์ของการออกแบบ Microservice

## (4) ระบบสามารถปรับขนาดได้อย่างละเอียด

ความสามารถในการปรับขนาดได้

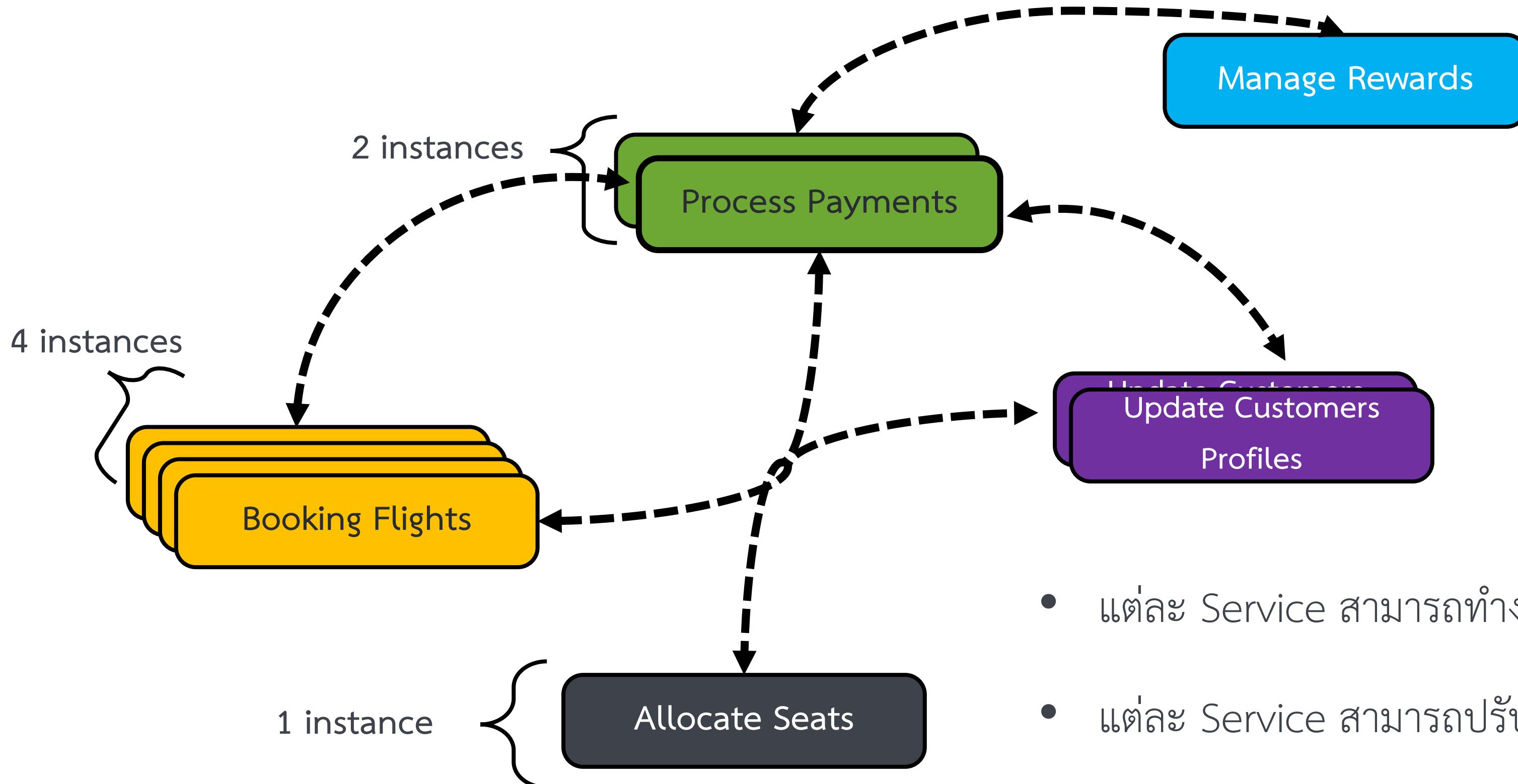
- สามารถปรับเพิ่มลดทรัพยากร (Resource) อย่างรวดเร็วและต่อเนื่องตามความต้องการ (Demand) โดยไม่กระทบต่อประสิทธิภาพการทำงานของระบบ
- ไม่ต้องมาคาดการณ์ปริมาณงานที่จะใช้งาน ในระบบจัดการแบบ Adaptive ซึ่งอาจจะอาศัยเทคโนโลยี Cloud ร่วมด้วยเพื่อการจัดการโครงสร้างพื้นฐาน (Infrastructure) ได้อย่างยึดหยุ่น แต่ถ้าเป็น Monolithic Application





# ประโยชน์ของการออกแบบ Microservice

## (4) ระบบสามารถปรับขนาดได้อย่างละเอียด



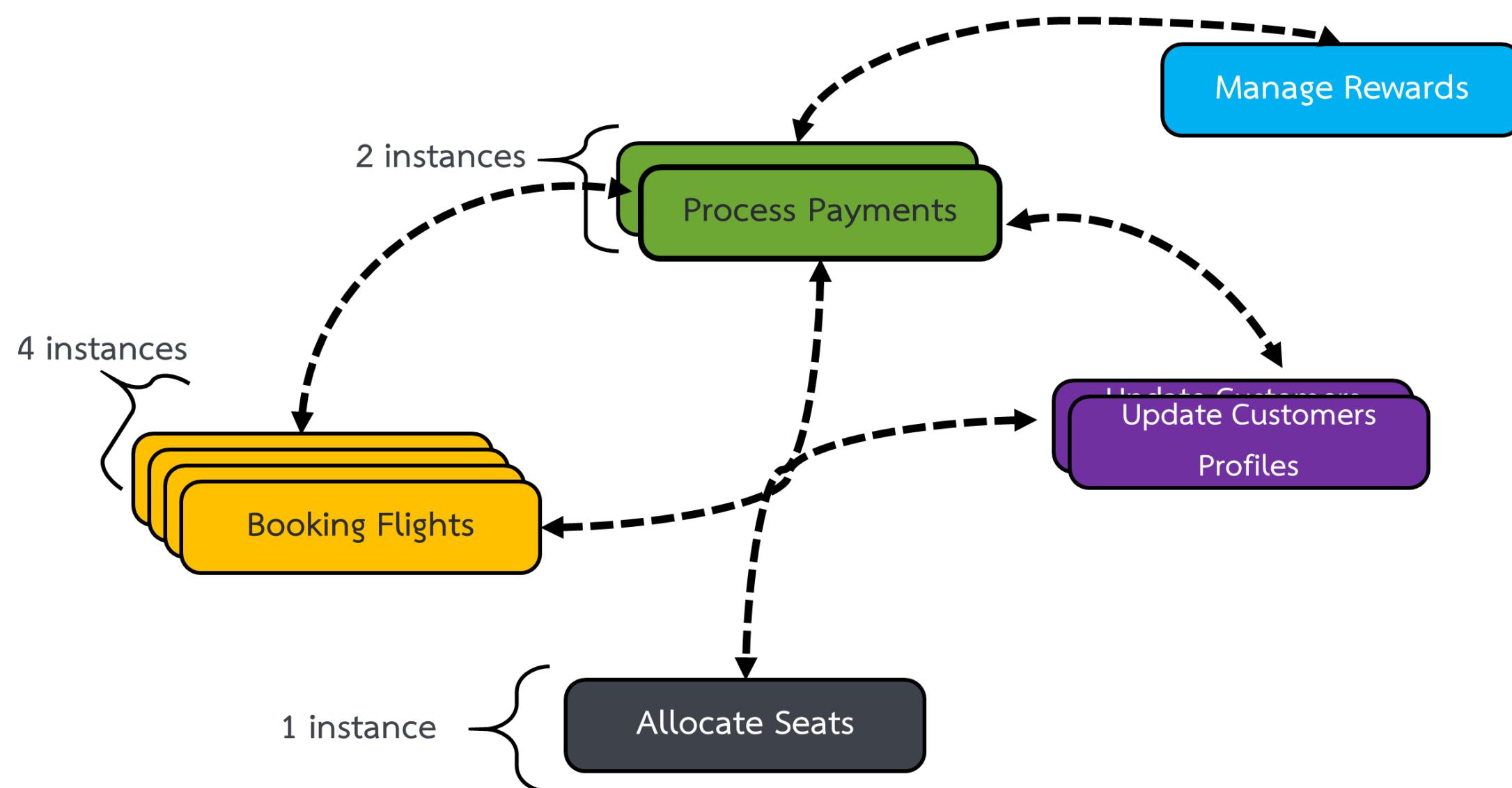
- แต่ละ Service สามารถทำงานบนทรัพยากรเฉพาะที่เหมาะสมกับ Business Logic นั้นได้
- แต่ละ Service สามารถปรับขนาดเฉพาะตัวได้ ซึ่งอาจจำความเฉพาะของข้อมูล
- ก่อให้เกิด “**MORE OPTIMIZED SCALING**”



# สิ่งที่ต้องคำนึงเมื่อใช้ Microservice

- **Complex Architecture**

แต่ละ service จะมีความ simpler และ smaller ขณะที่ภาพรวมของสถาปัตยกรรมจะซับซ้อนขึ้นเรื่อยๆ เนื่องจากการพัฒนาระบบแบบ Microservice นั้น



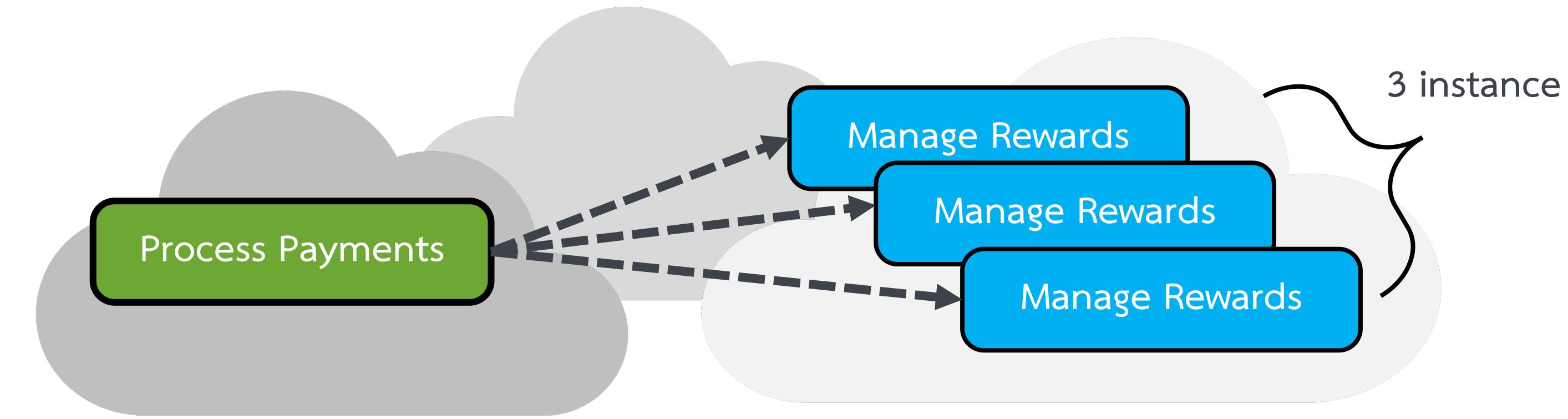
- แต่ละ service จะทำงานแบบกระจาย (distributed architecture)
- การจัดการสื่อสารระหว่าง Service ได้
- การจัดการช่องทางและรูปแบบการสื่อสาร

ดังนั้น ภาระงานและความซับซ้อนของระบบจะย้ายไปอยู่ที่ 3 ปัจจัยข้างต้น



# สิ่งที่ต้องคำนึงเมื่อใช้ Microservice

- Service Discovery



เมื่อสอง service ต้องการคุยกันโดยตรงแบบ request - response ซึ่งไม่มี message broker มาเป็นตัวกลาง ซึ่งในสภาพแวดล้อมแบบ cloud นั้นจะสามารถสร้าง instance ของ service อัตโนมัติได้ตาม traffic demand

คำถามคือว่า เมื่อมี instance ใหม่เพิ่มมาหรือถูกลบออกแล้ว service อื่น ๆ จะรู้ได้อย่างไร โดยธรรมชาติของ instance จะมี IP address และ port number แบบ dynamic



# สิ่งที่ต้องคำนึงเมื่อใช้ Microservice

- **Networking**

แต่ละ service จะสื่อสารกันผ่าน APIs ดังนั้น ถ้าเครือข่ายล้ม ระบบจะไม่สามารถทำงานได้ นอกจากนี้ ต้องค่อยส่งเกตุ bandwidth ของเครือข่ายว่ามีความพอเหมาะหรือไม่ และความเร็วในการตอบสนอง (latency) ว่าช้าไปไหม

- **Monitoring**

สังเกตุสถานะ เหตุการณ์ และข้อมูลต่าง ๆ ของ service จาก telemetry data เช่น metric, logs, health เพื่อดูว่า service นั้นยังทำงานได้หรือไม่ หรือมีแนวโน้มว่าจะล้ม

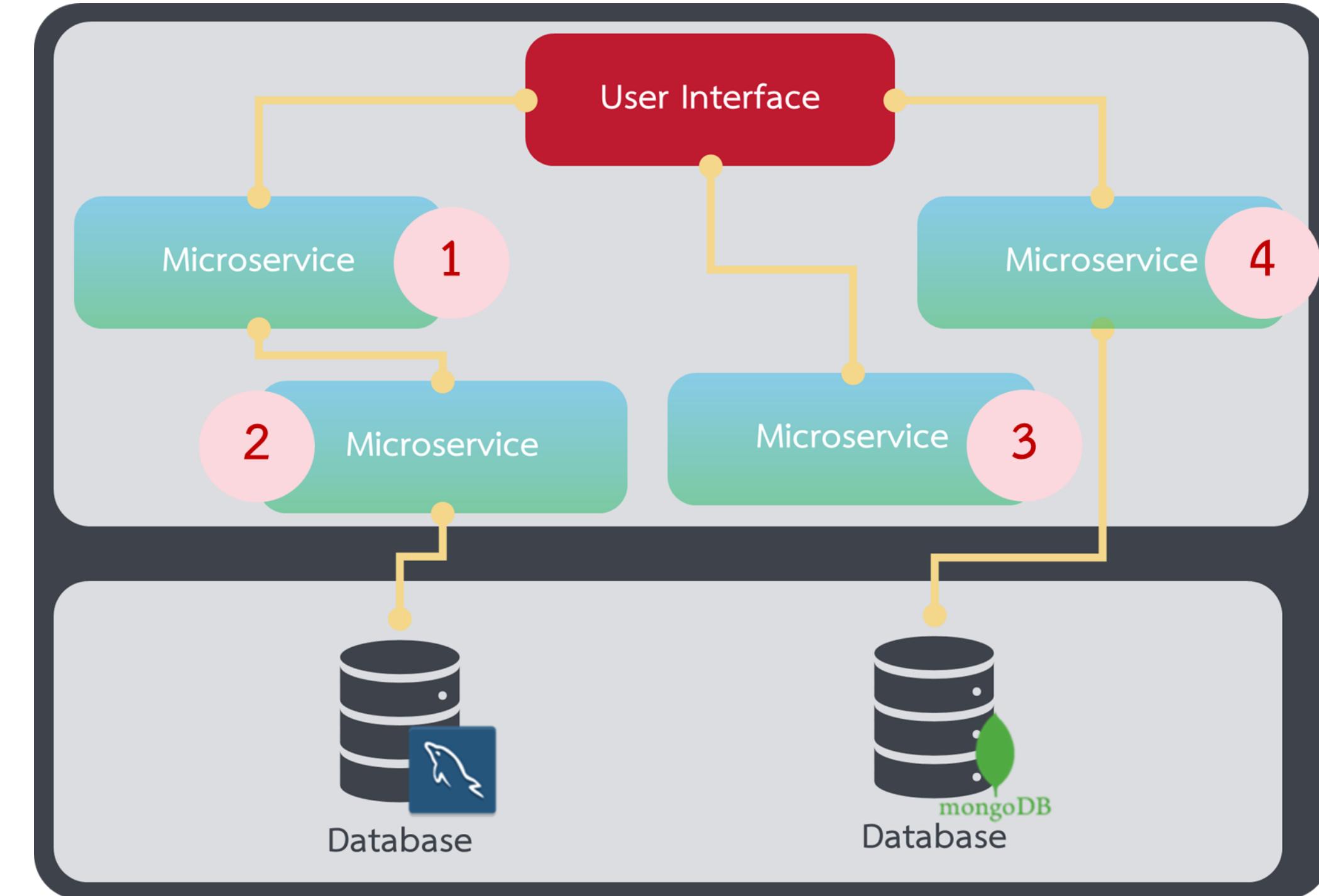
- **Resiliency**

“DON’T DESIGN FOR FAILURE, DESIGN FOR RECOVERY” ดังนั้น ต้องระบุ single-point-of-failure ให้ได้



# สรุปประโยชน์ของ Microservice

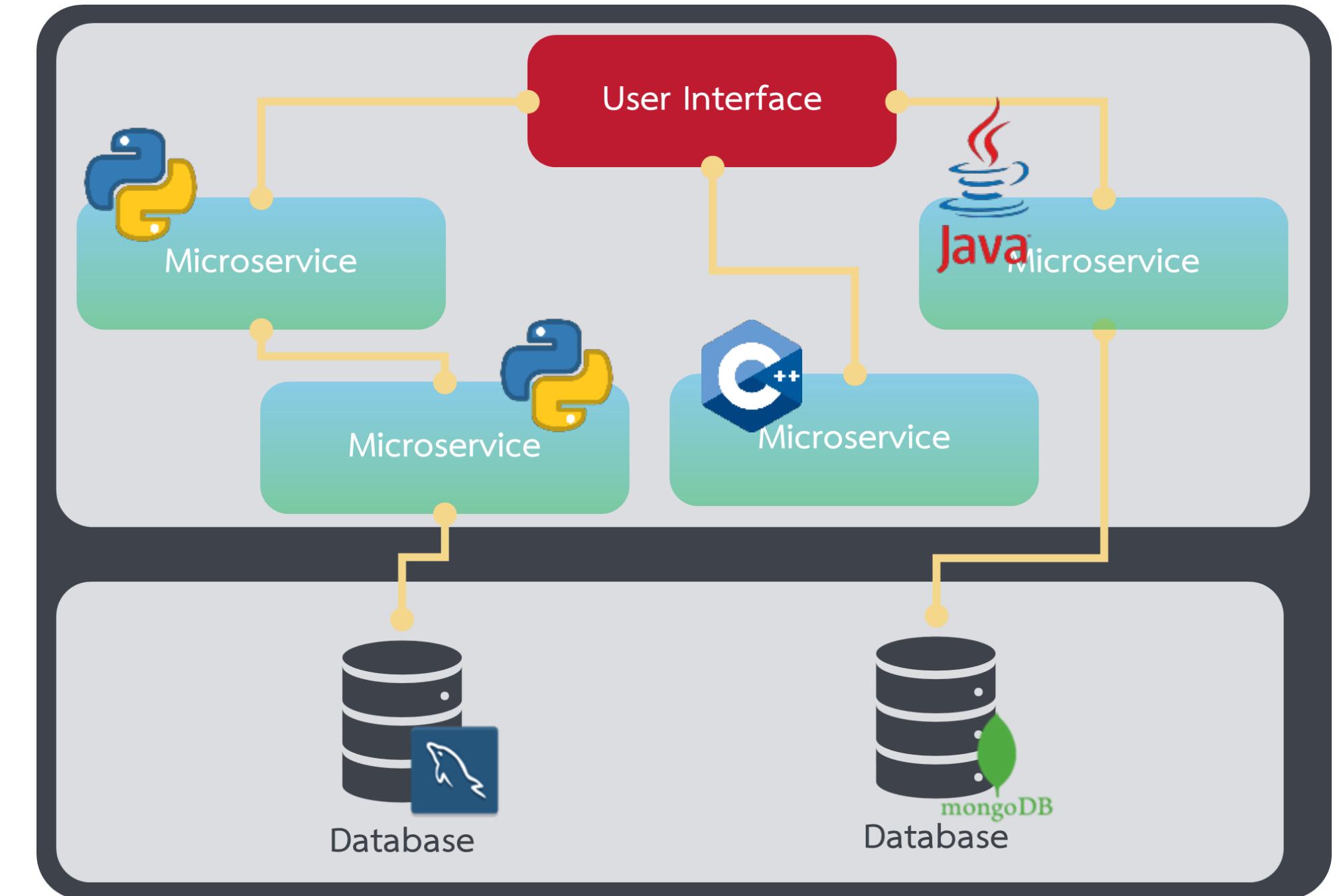
นอกจากนี้ ระบบที่ได้รับการพัฒนาบนหลักการ Microservice ยังจะมีข้อดีดังต่อไปนี้



- **Loosely coupled**
  - ✓ Service (3) และ (4) มีความเป็นอิสระทั้งงานและข้อมูล
  - ✓ Service (1) และ (2) มีการเชื่อมโยงกัน “ถ้า (2) ล้ม (1) ก็จะล้มตาม” ถ้า Service (2) เป็นงานทางด้าน Infrastructure กรณีจะมีปัญหาน้อยกว่า ถ้าเป็น Business Logic



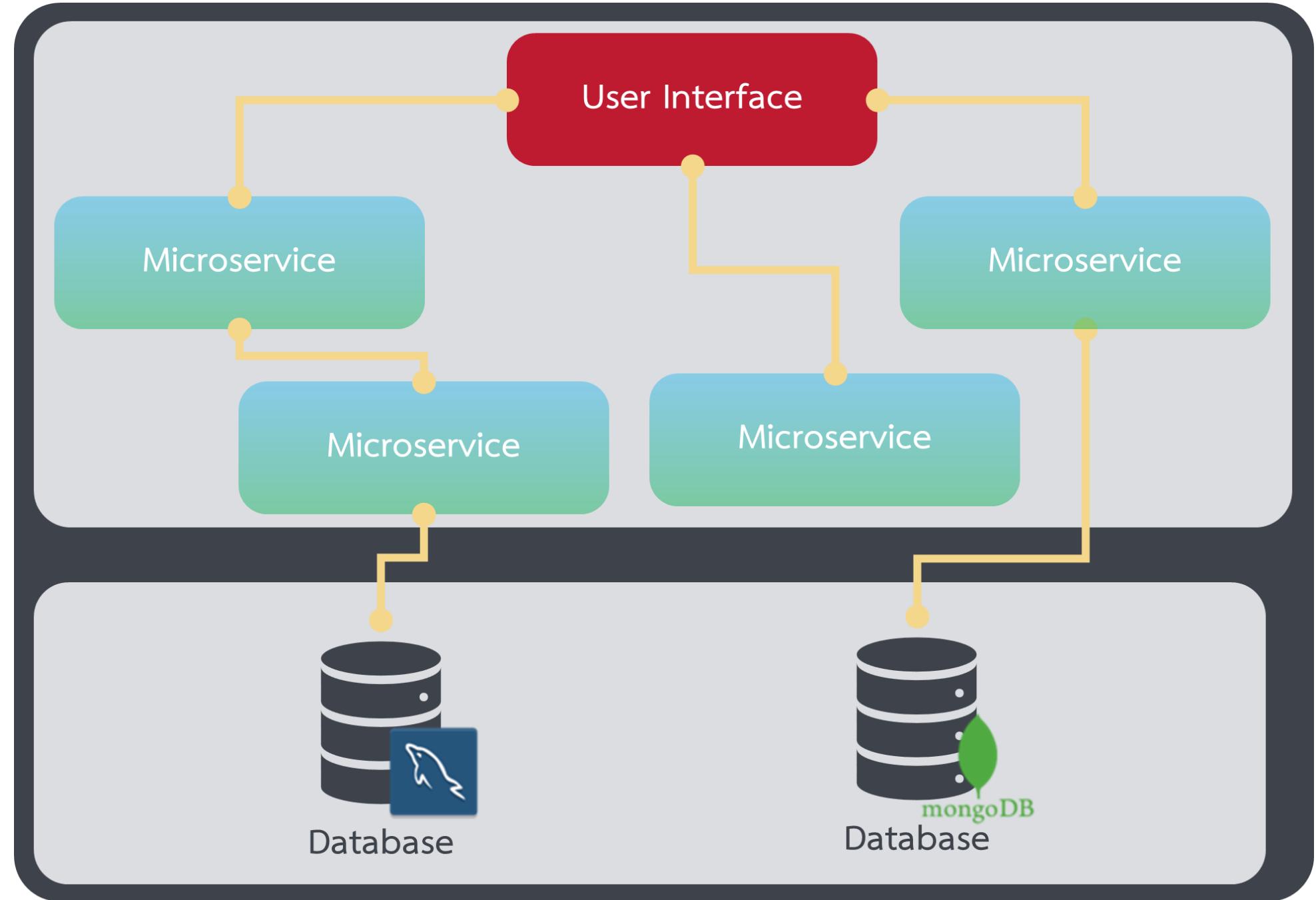
# สรุปประโยชน์ของ Microservice



- Technology flexibility แต่ละ Service ไม่จำเป็นต้องพัฒนาหรือใช้เทคโนโลยีเดียวกัน



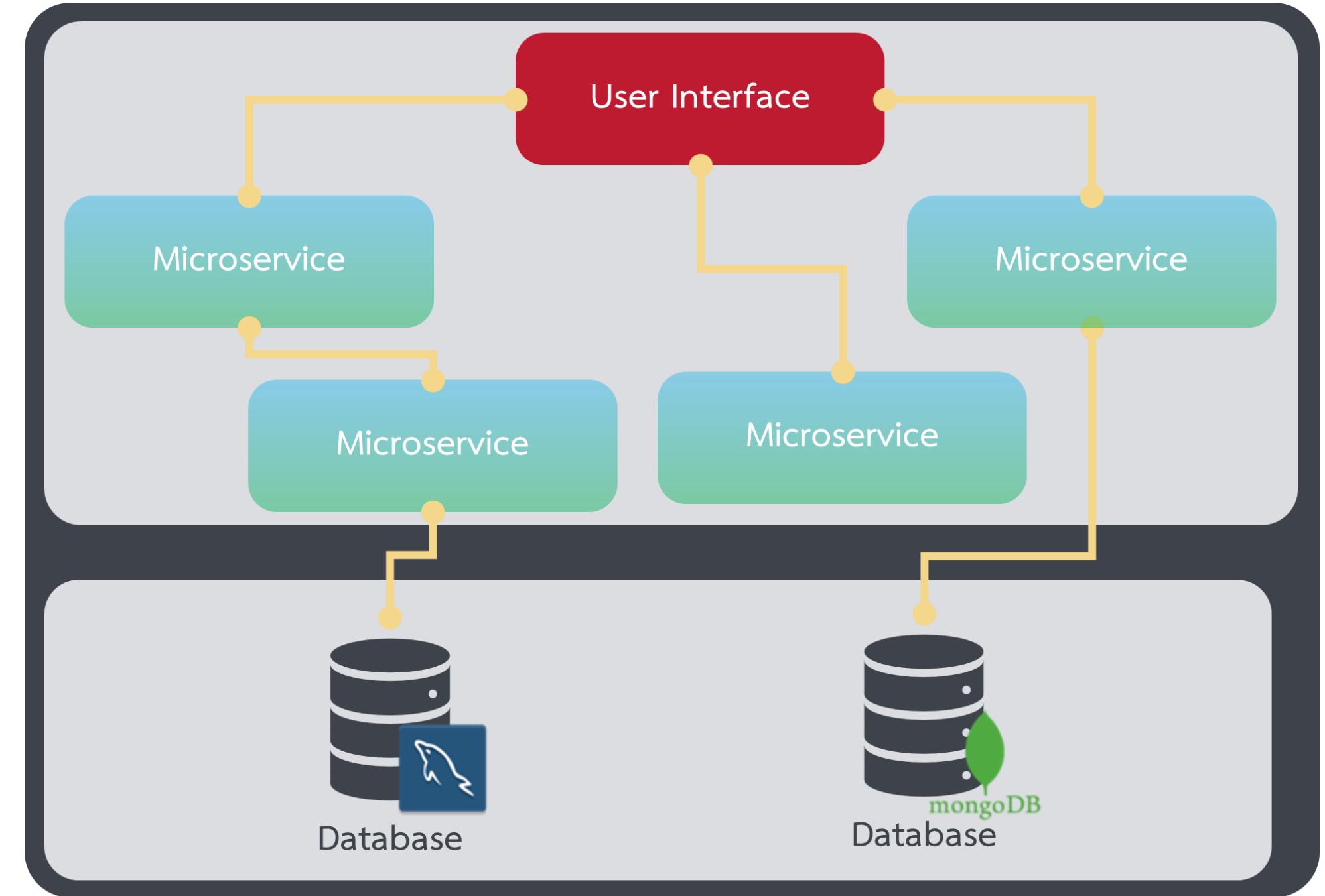
# สรุปประโยชน์ของ Microservice



- Focus on business first ให้ความสำคัญกับความต้องการของภาคธุรกิจเป็นสำคัญ รวดเร็วต่อการปรับเปลี่ยน
- Good software quality เนื่องจากแต่ละ Service มีขอบเขตงานและหน้าที่ความรับผิดชอบที่ชัดเจน



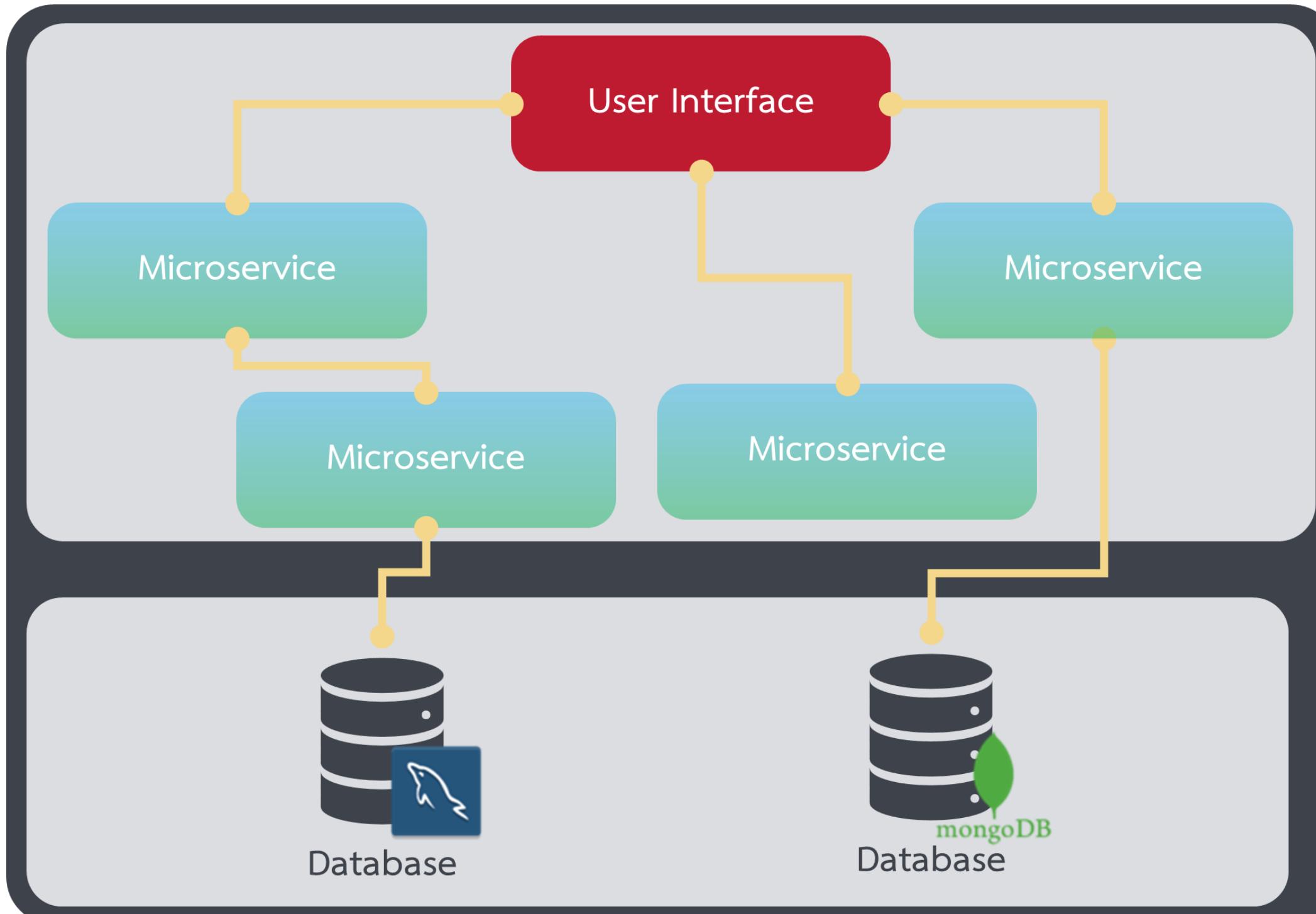
# สรุปประโยชน์ของ Microservice



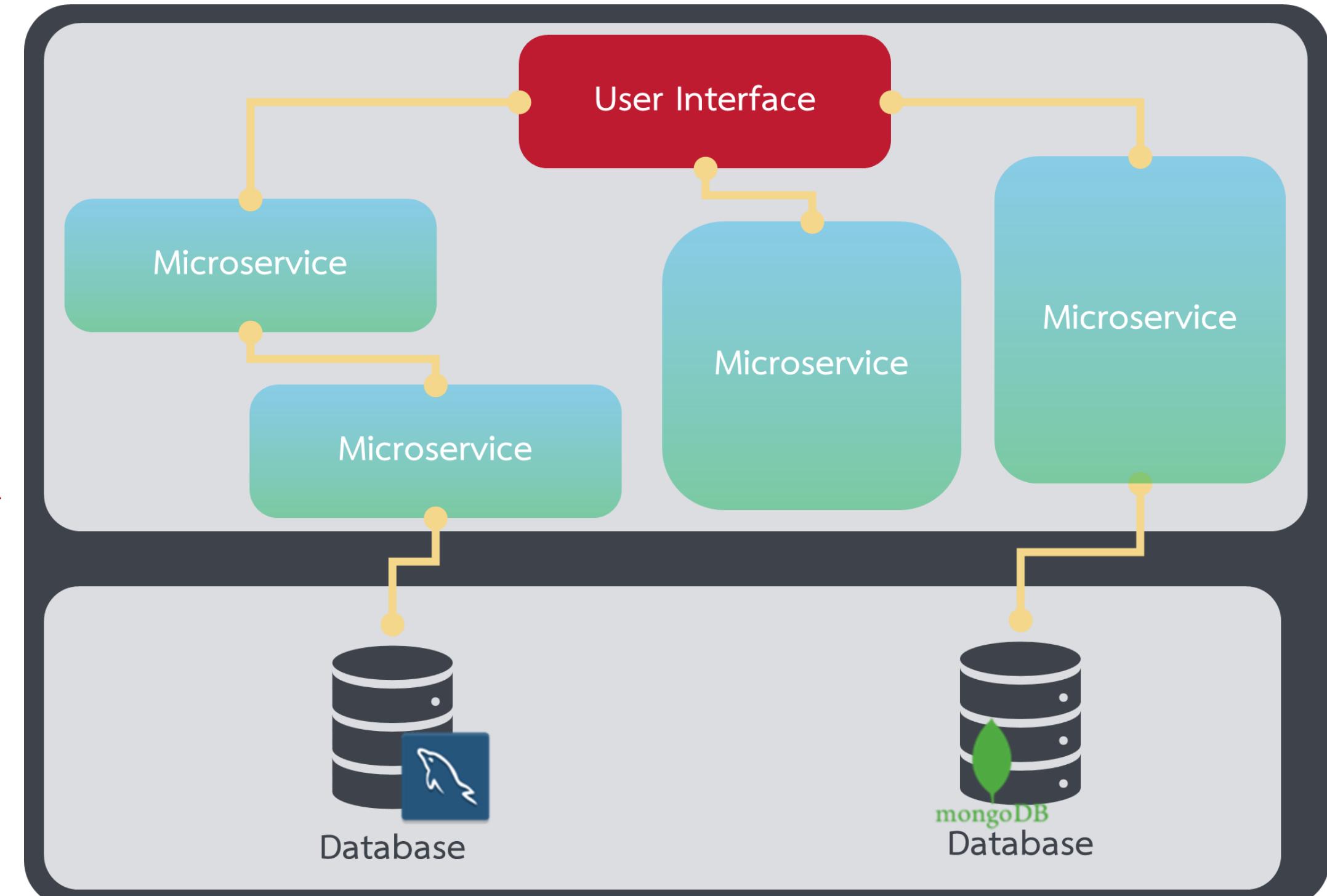
- Easy to understand and develop by small team เนื่องจากมีขอบเขตที่จำกัดและงานที่ชัดเจน ทำให้ง่ายต่อการทำความเข้าใจ
- Deploy independently สามารถ Deploy แต่ละ Service ได้อย่างอิสระและไม่ขึ้นกับ Service อื่น แต่ต้องพิจารณาและทดสอบก่อน Deploy ตาม Contract Test



# สรุปประโยชน์ของ Microservice



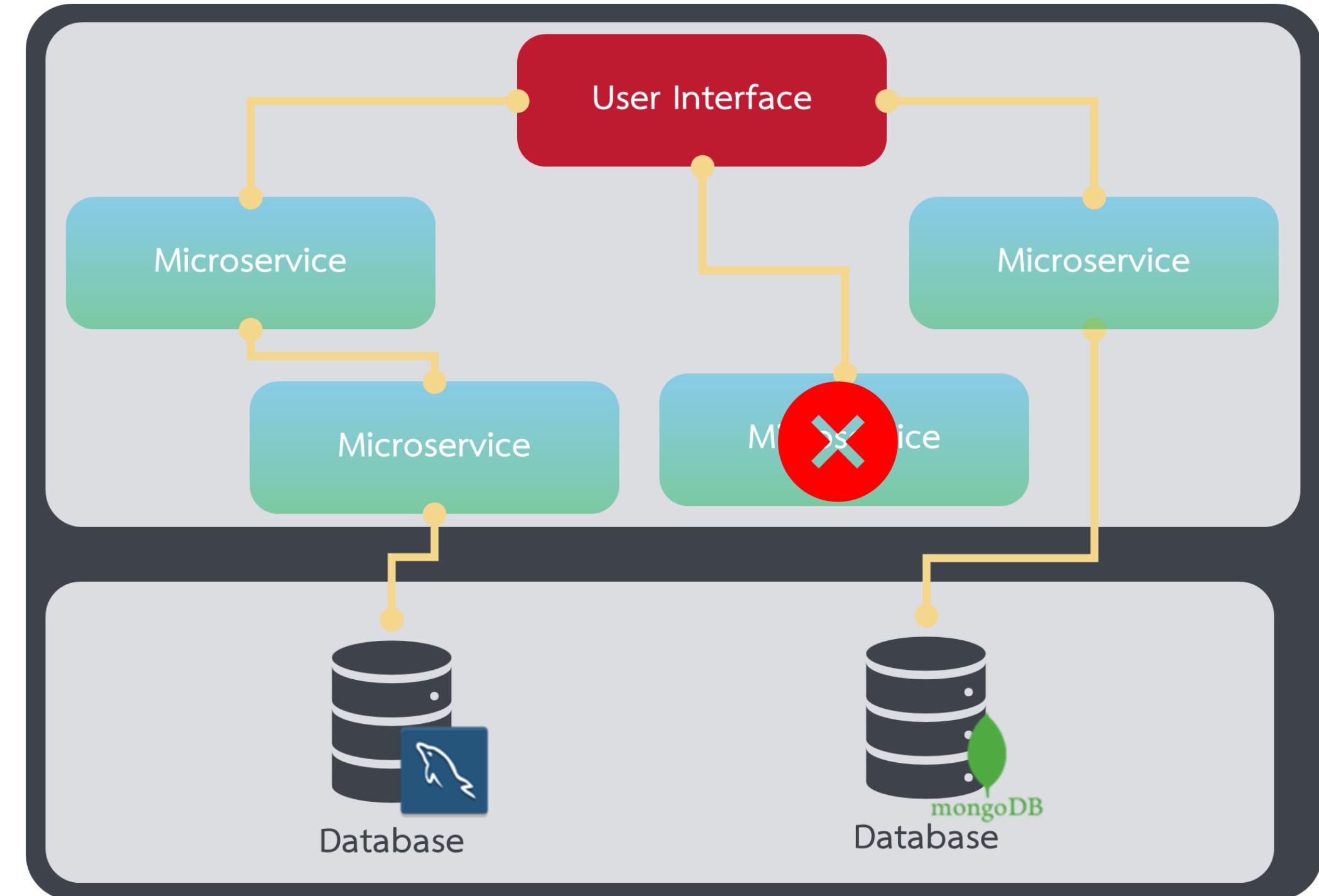
Scalable



- Scale independently เมื่อ Service ได้ต้องการเพิ่มขนาด เพื่อรับบริการที่มากขึ้น จะสามารถขยายเฉพาะ Service นั้นได้โดยไม่กระทบต่อระบบ



# สรุปประโยชน์ของ Microservice

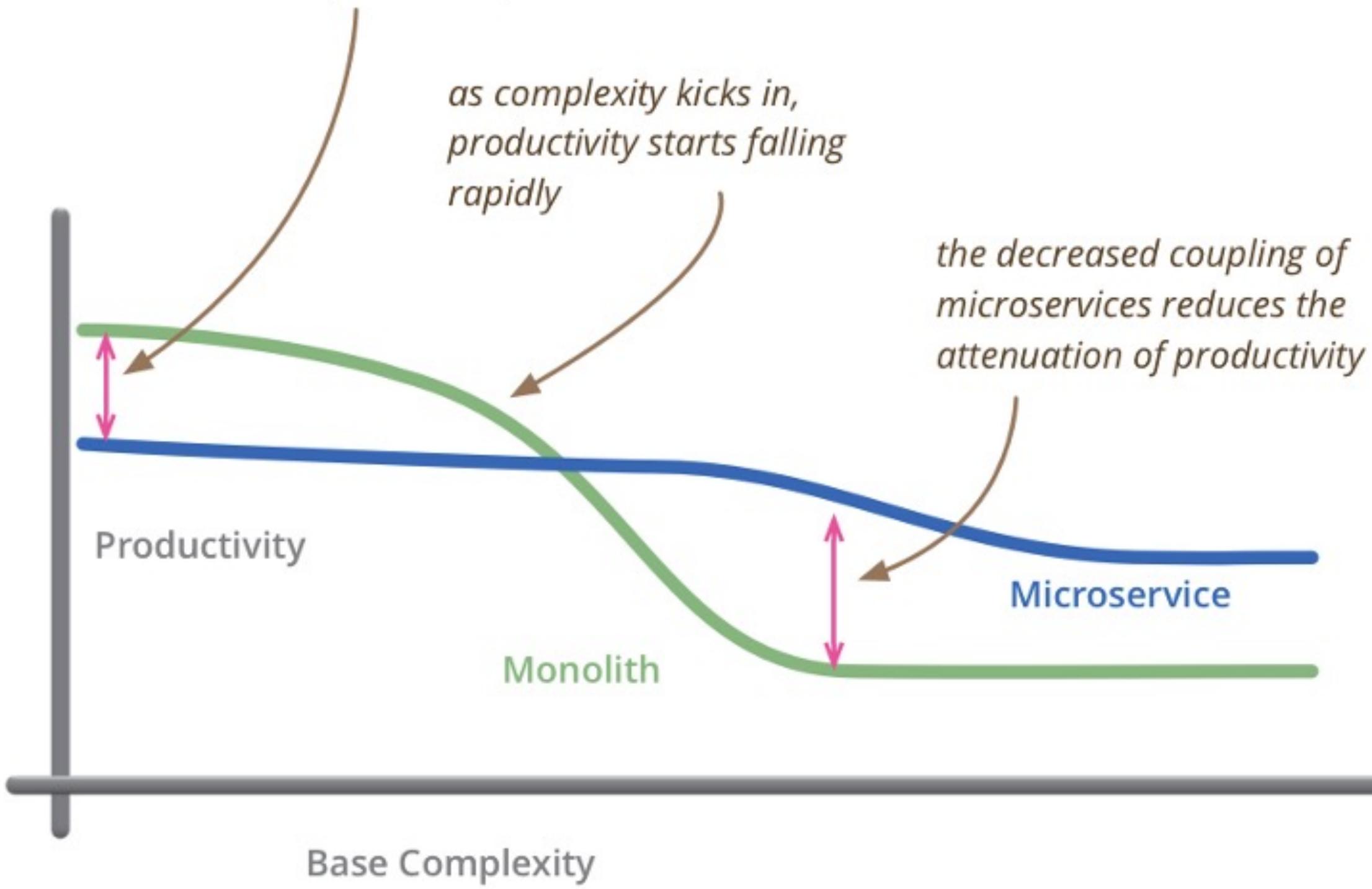


- Reliability เมื่อบาง Service ขัดข้อง ระบบยังสามารถทำงานได้ ไม่ใช่ล้มทั้งระบบ



# ข้อเสียของการออกแบบ Microservice

*for less-complex systems, the extra baggage required to manage microservices reduces productivity*



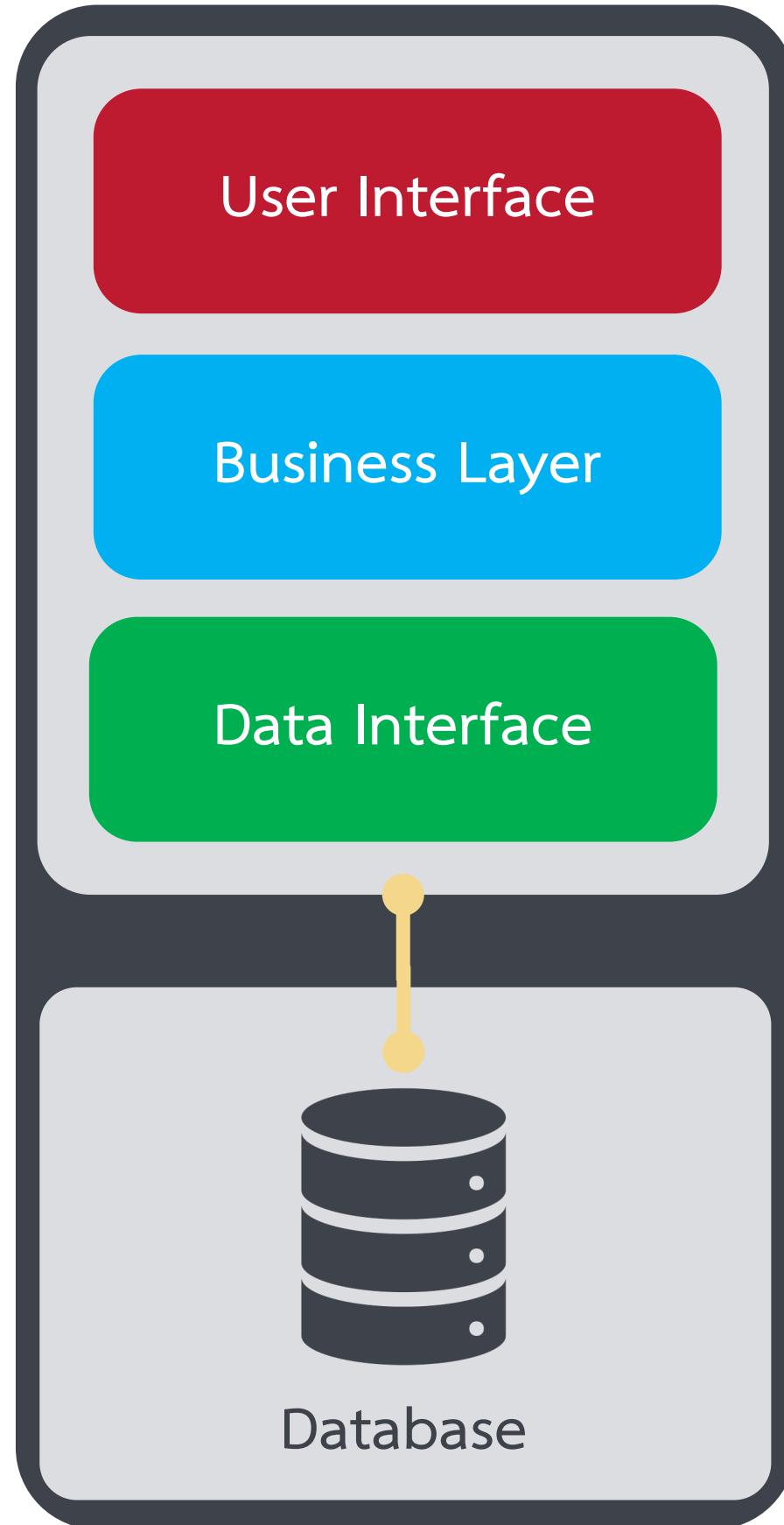
Integration testing ค่อนข้างเกิดยาก แต่อาศัยการทดสอบแบบใหม่มาช่วย คือ contract testing

การ deploy ค่อนข้างวุ่นวายกว่า เพราะต้อง deploy หลายรอบ ขณะที่ แบบ monolithic จะ deploy เพียงครั้งเดียว

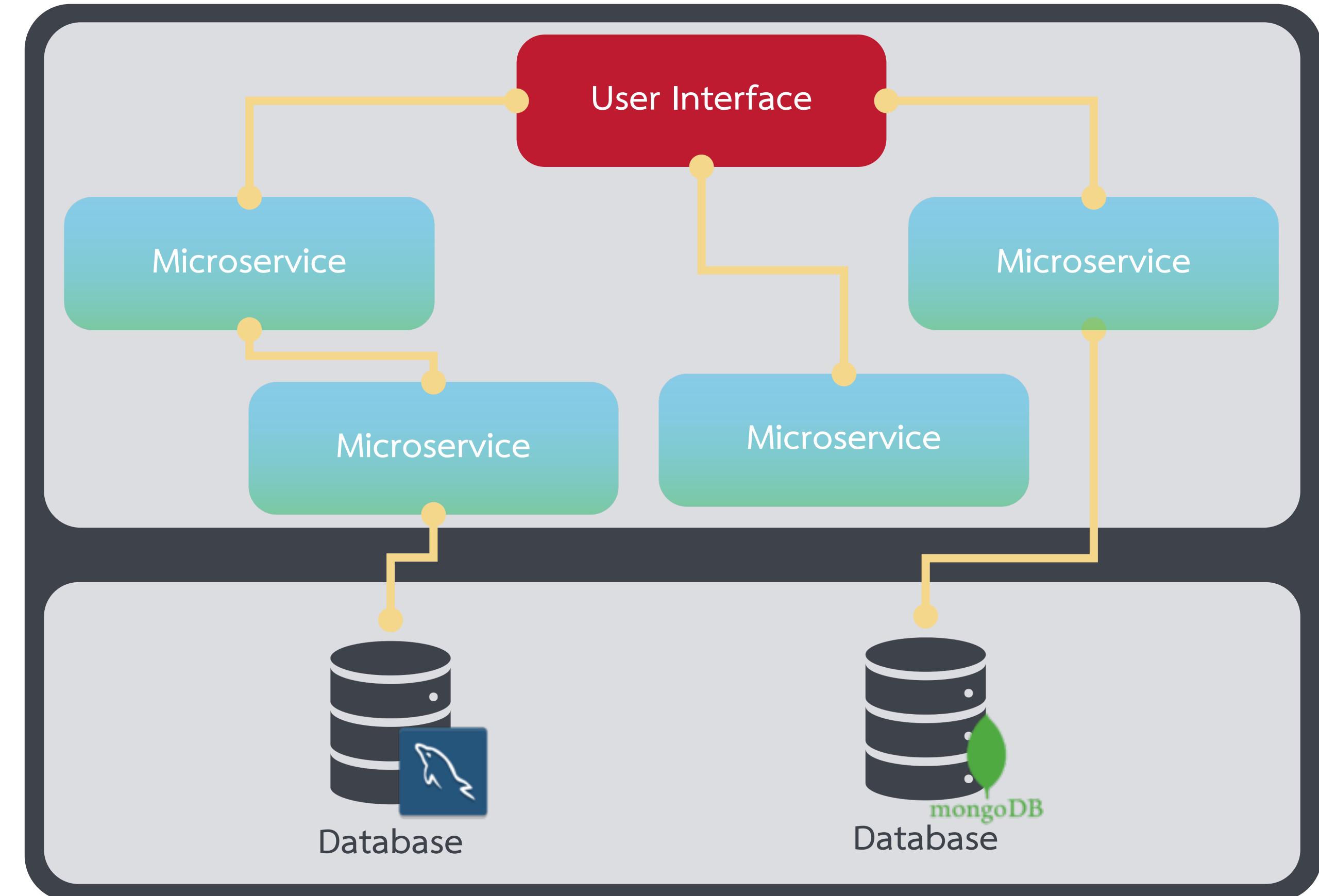
ต้นทุกสูงกว่าแบบ monolithic เนื่องจากมีทีมของตัวเอง มีฐานข้อมูลของตนเอง มีสภาพแวดล้อมของตัวเอง มีเทคโนโลยีของตนเอง (Hardware และ Software)



# Monolithic vs Microservice



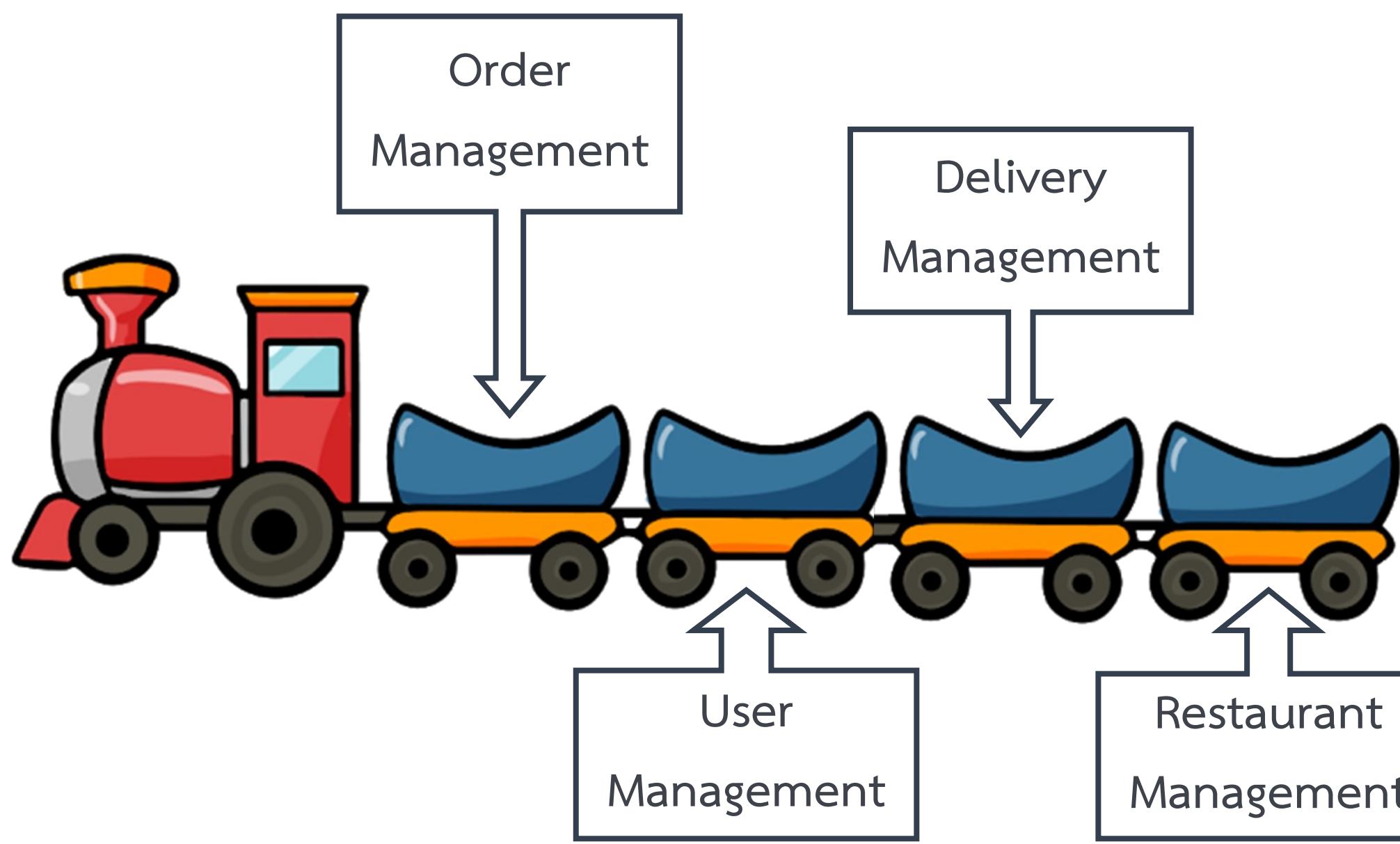
Monolithic



Microservice

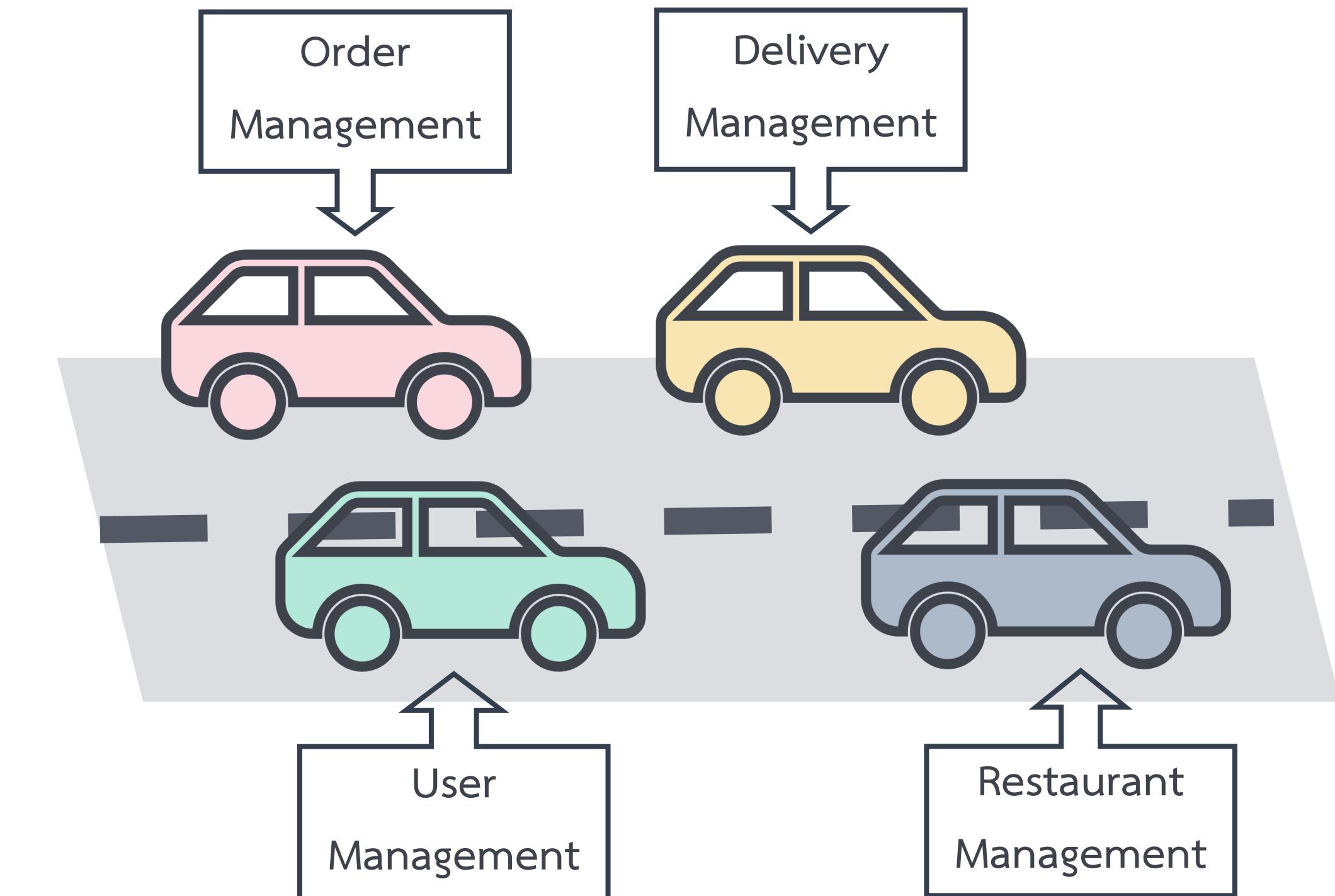


# Monolithic vs Microservice



## Monolithic

มีลักษณะการประมวลผลแบบ Centralization (คือ การรวมทุก Logic เข้าไว้ด้วยกัน)

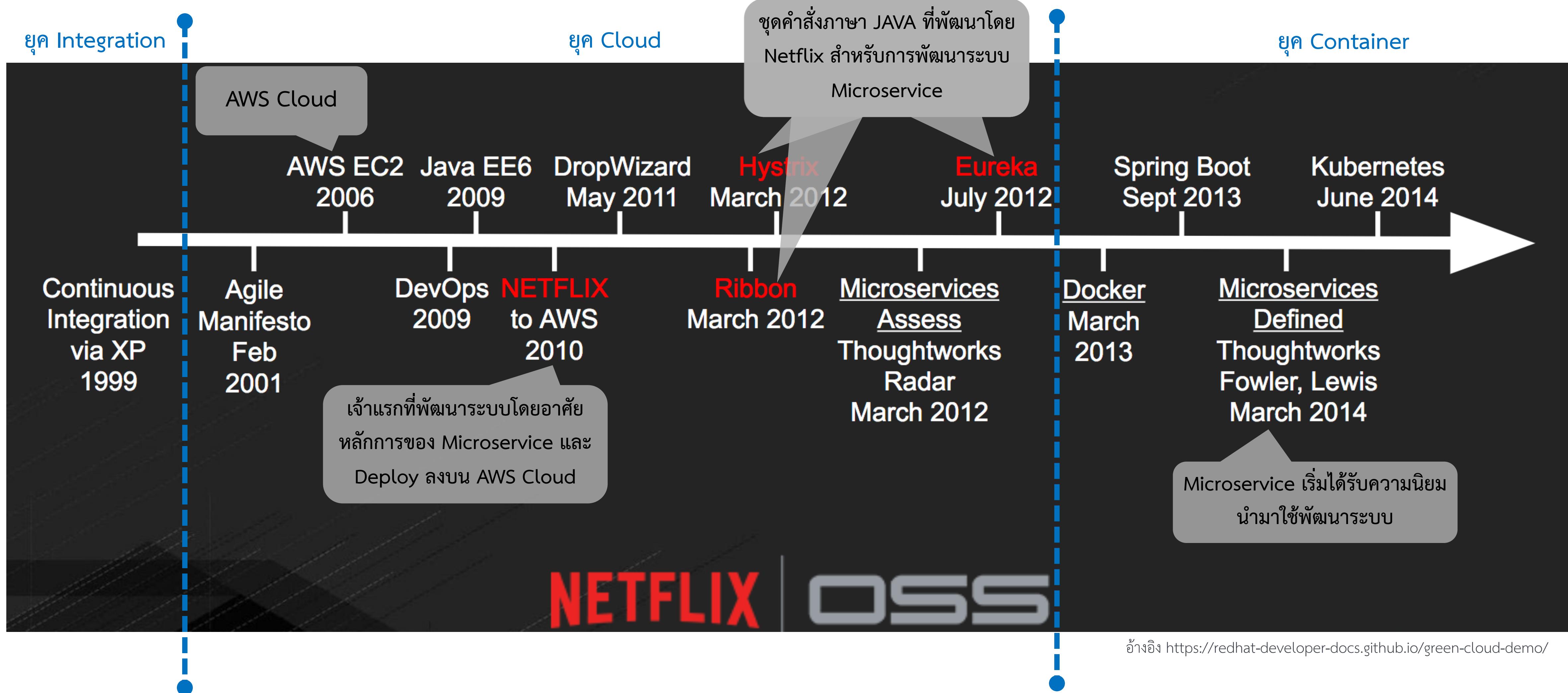


## Microservice

มีลักษณะการประมวลผลแบบ Distribution (คือ แยก Service ตาม Logic)

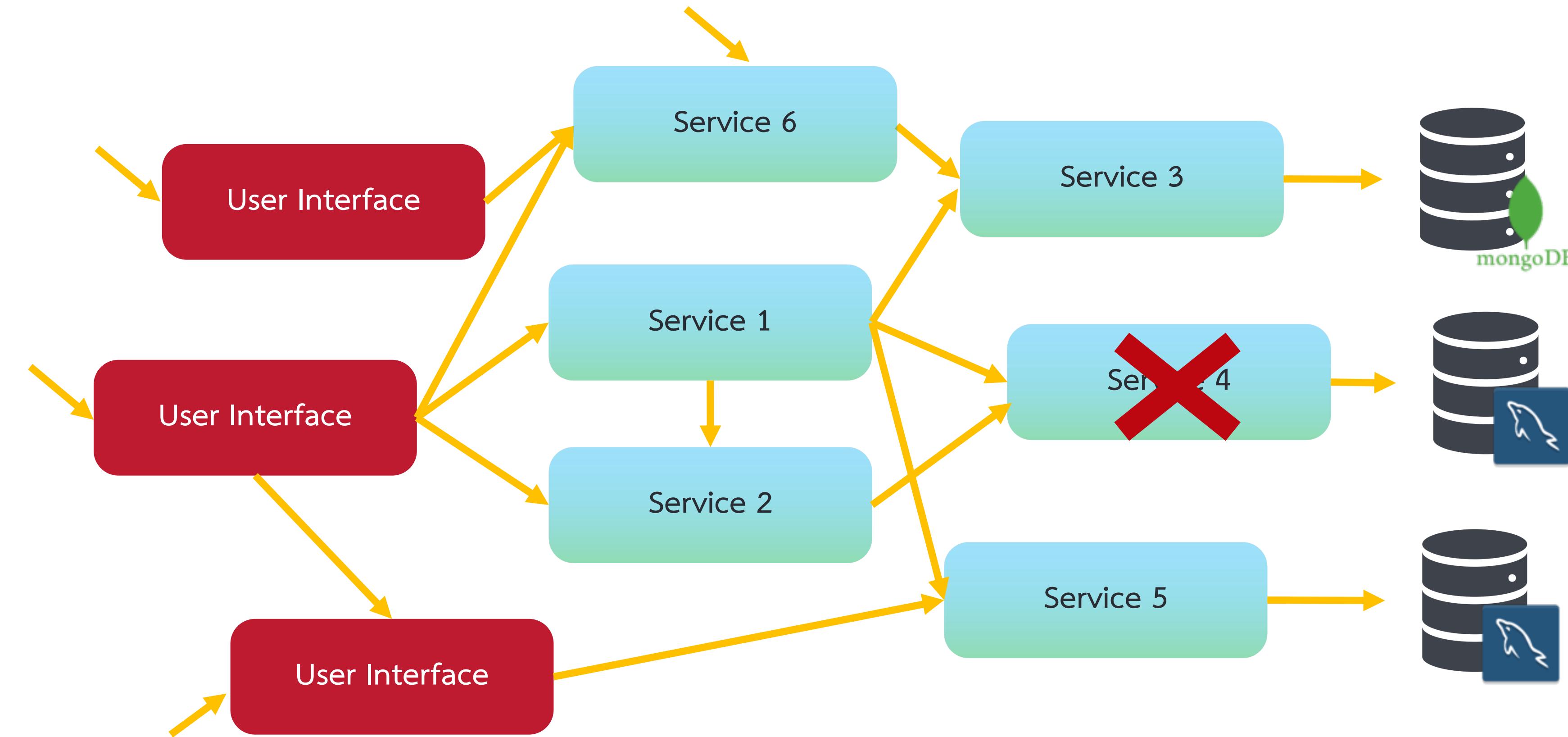


# วิวัฒนาการของ Microservice





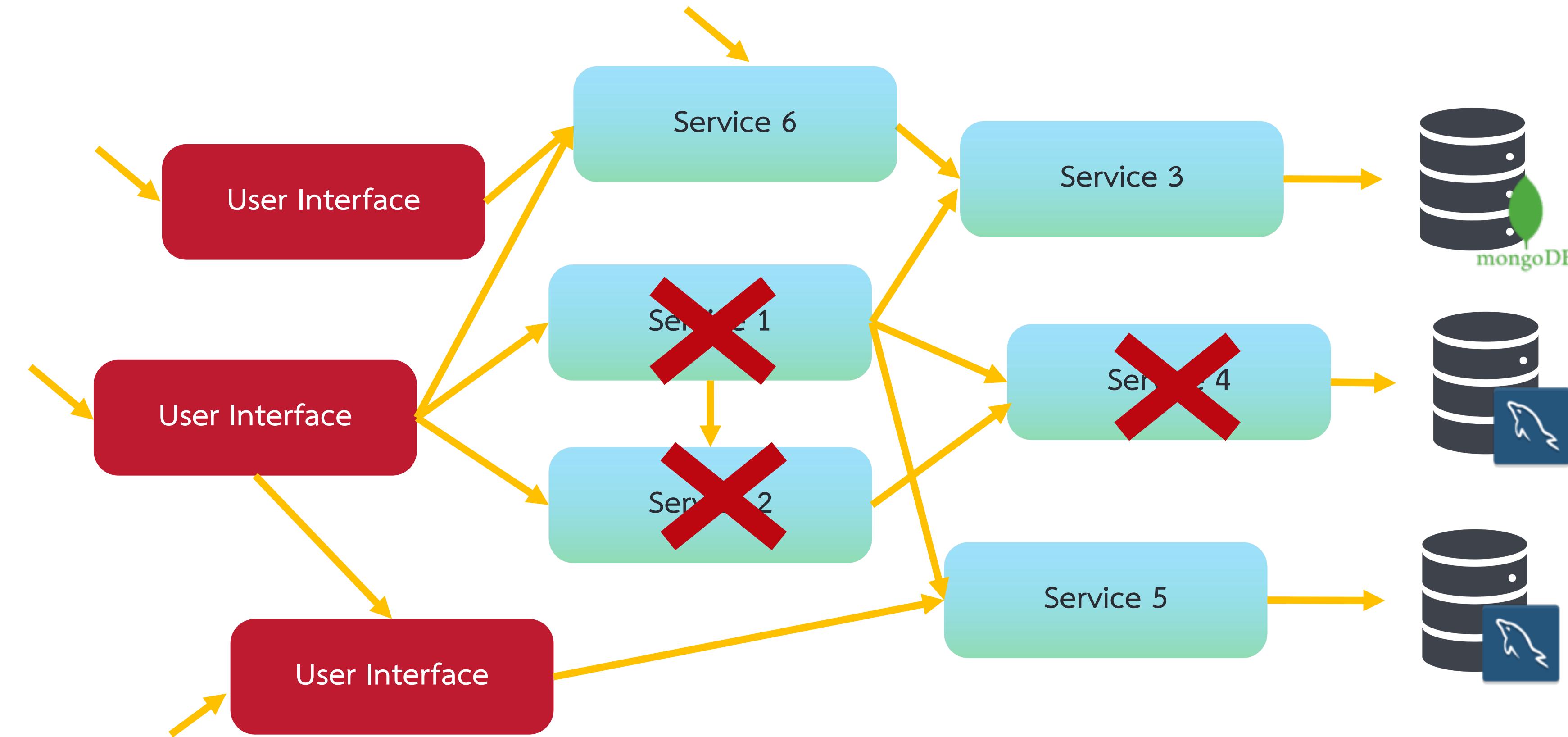
# Cascading Failure



Failure คือ เมื่อ Service บางตัวล้มแล้ว Service ที่เกี่ยวข้องหรือสัมพันธ์กันจะทยอยล้มตามกันไปหากไม่มีการจัดการ Cascading Failure



# Cascading Failure



Failure คือ เมื่อ Service บางตัวล้มแล้ว Service ที่เกี่ยวข้องหรือสัมพันธ์กันจะทยอยล้มตามกันไปหากไม่มีการจัดการ Cascading Failure



# Microservice Architecture

**“DON’ T DESIGN FOR FAILURE, DESIGN FOR RECOVERY”**

ระบบที่ไม่ล้ม คือ (1) ระบบที่ไม่เสร็จ หรือ (2) ไม่มีคนใช้งาน ดังนั้น ระบบเราสามารถล้มได้เสมอ นักศึกษาจึงควรวางแผน Recovery เสมอในการออกแบบ

- การมองแบบที่ 1 คือ ระบบล้มไปแล้ว

ต้องระบุสาเหตุให้ได้ว่าล้มเพราะอะไร → หาทาง Recovery ขึ้นมา ตัวอย่างเช่น ไฟไหม้อาจจะสั่งให้ตัว Stand by ทำงานแทน

- การมองแบบที่ 2 คือ ระบบยังไม่ล้ม แต่มีวิ่งเวลาจะล้ม

คาดการณ์ได้ว่าจะเกิดโอกาสล้ม → เตรียมแผนรับมือ

อาจจะคาดการณ์จาก latency ว่าซ้ำไปใหม่, metric, logs, service health เป็นต้น



# របន់គិតថាមរយៈ Netflix

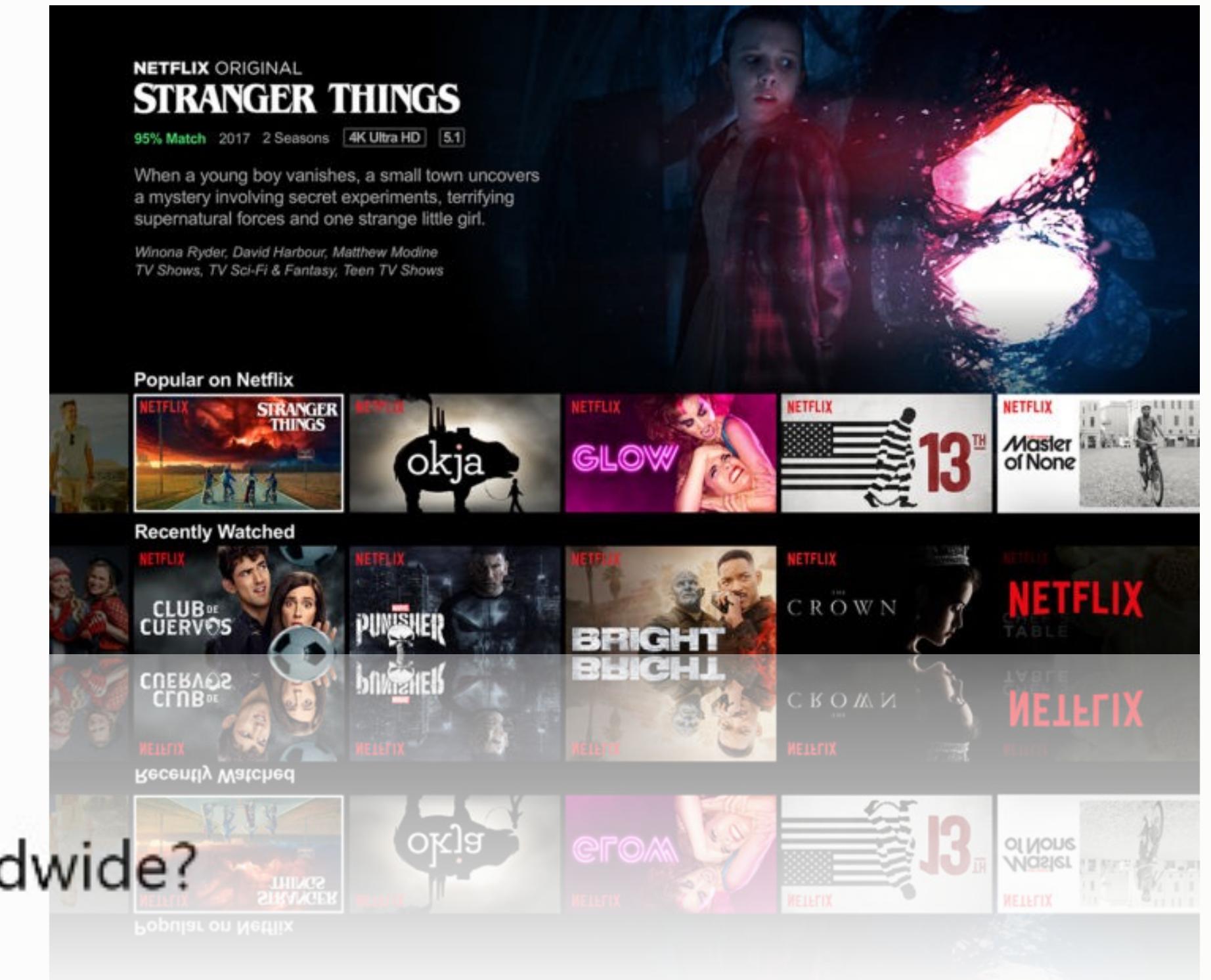
# **Netflix** – Industry Case Study

- About Netflix

- A media-service provider and production company
  - Subscription-based streaming service
  - **148M** paid subscribers! (2019)

- Software

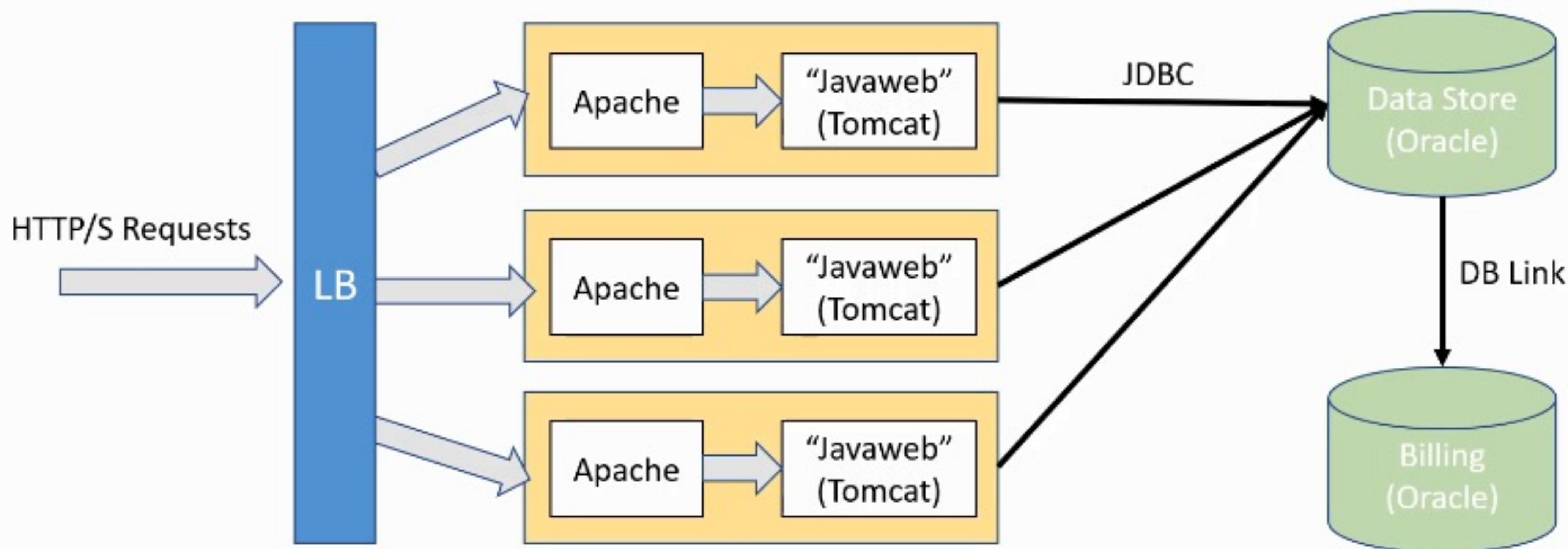
- What is needed for providing high-quality service worldwide
  - Industry pioneer with micro-services
  - **Monolithic application** architecture was not good enough...





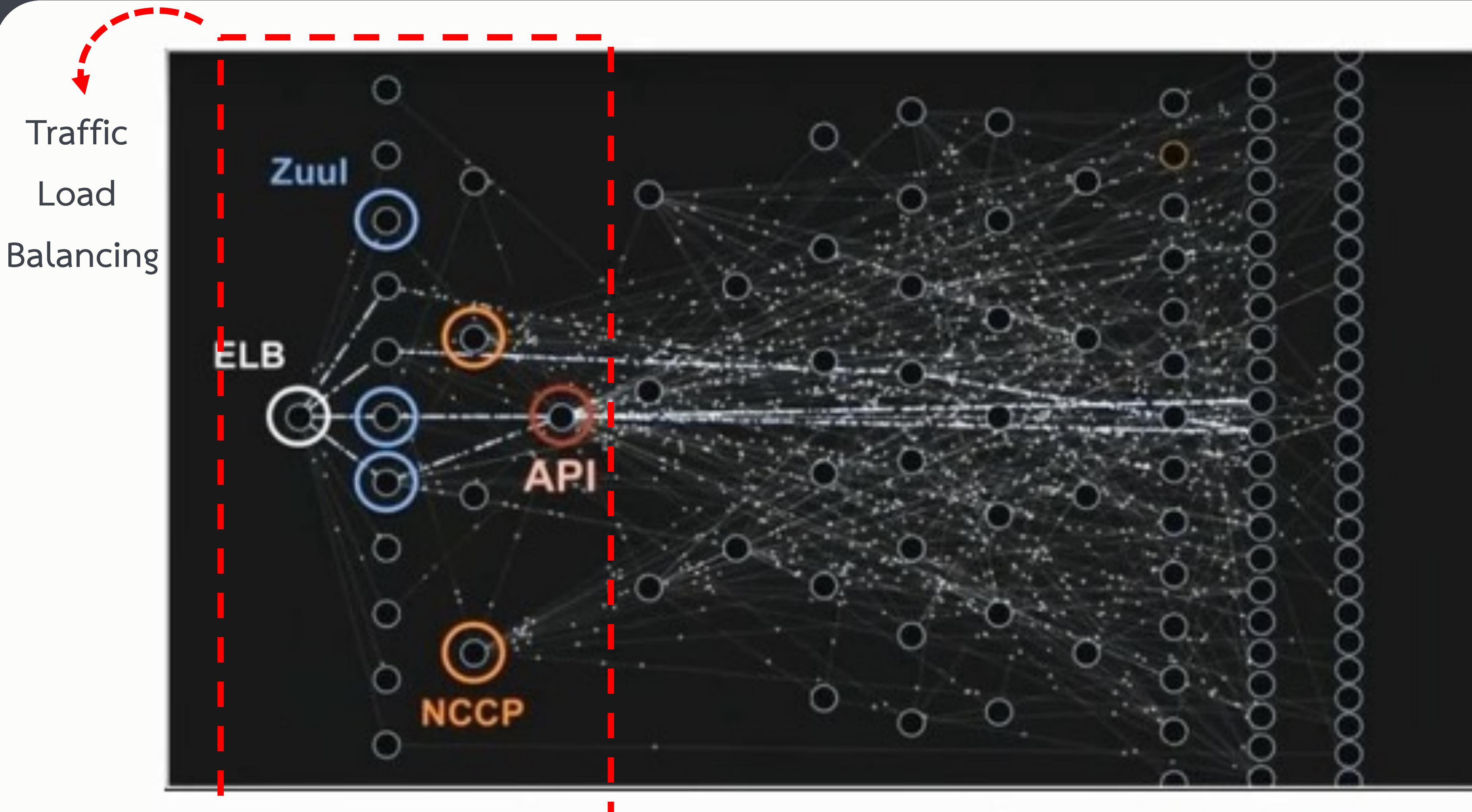
# กรณีศึกษา Netflix

- **Monolithic Architecture (~Data Center - 2000)**





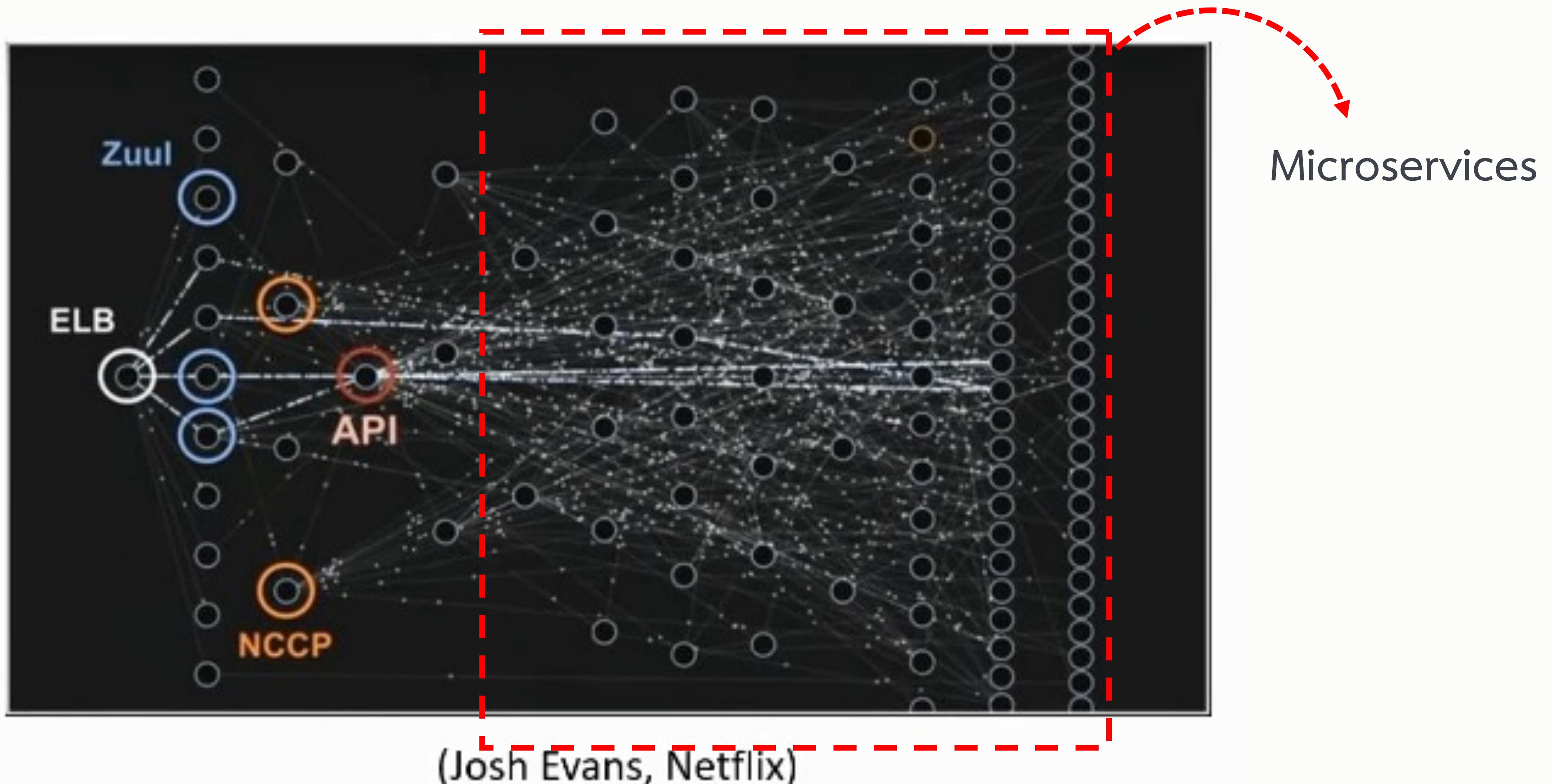
# กรณีศึกษา Netflix



(Josh Evans, Netflix)

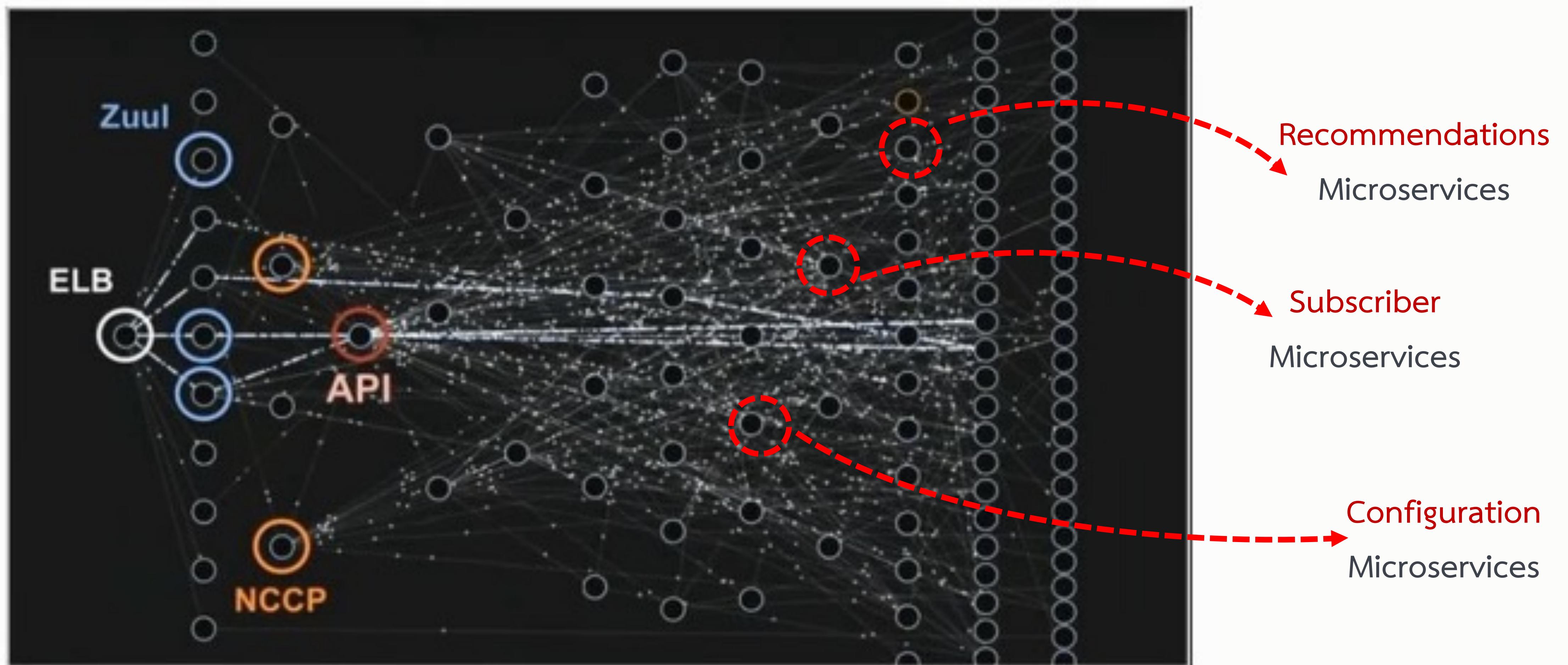


# กรณีศึกษา Netflix

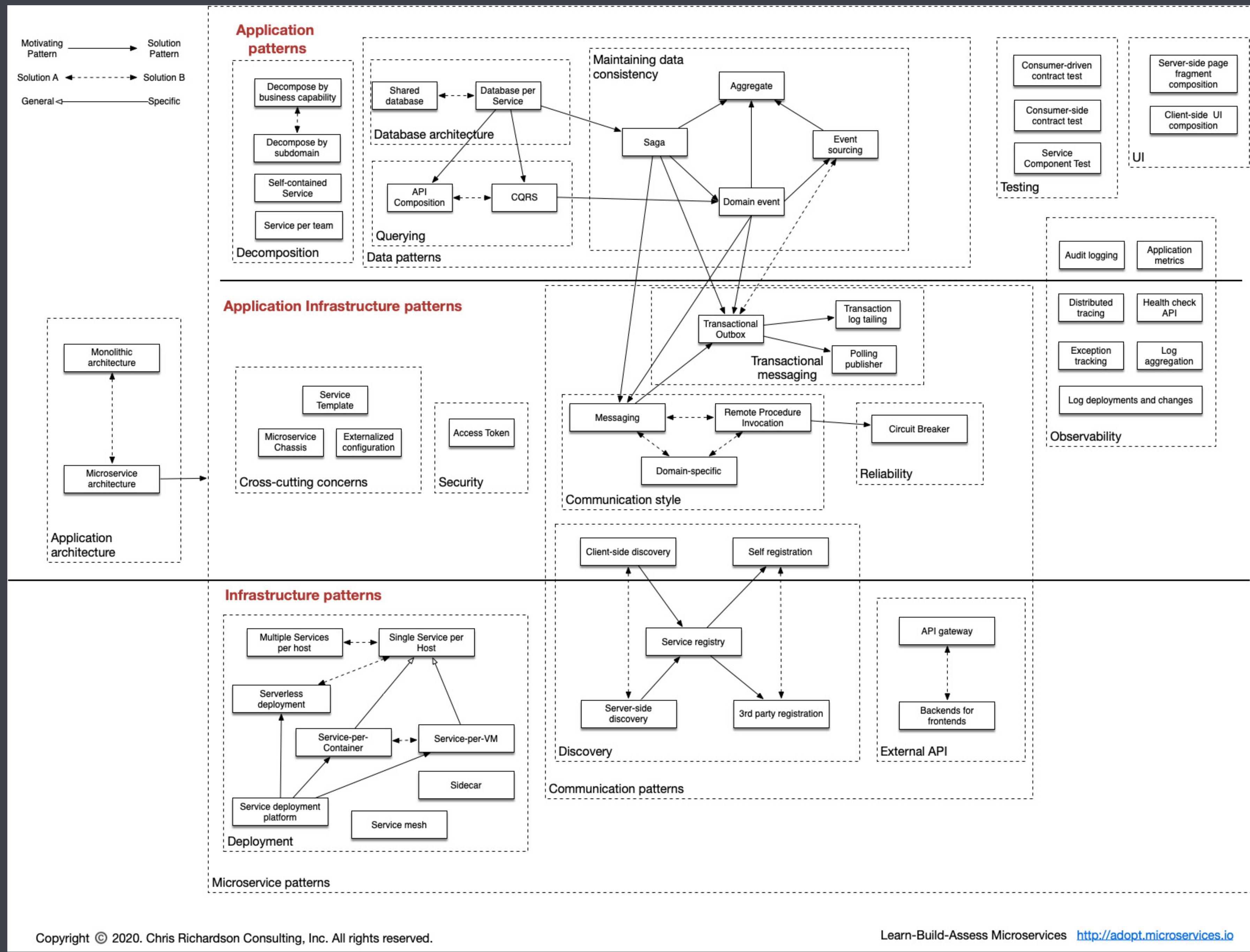


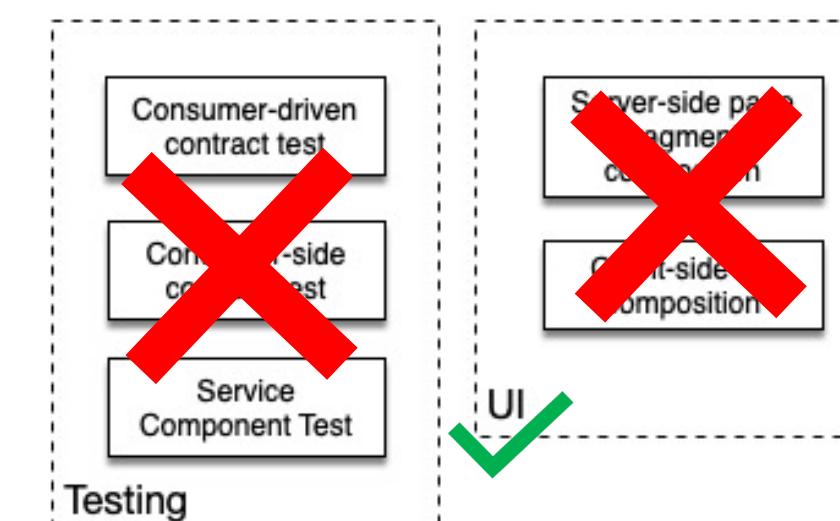
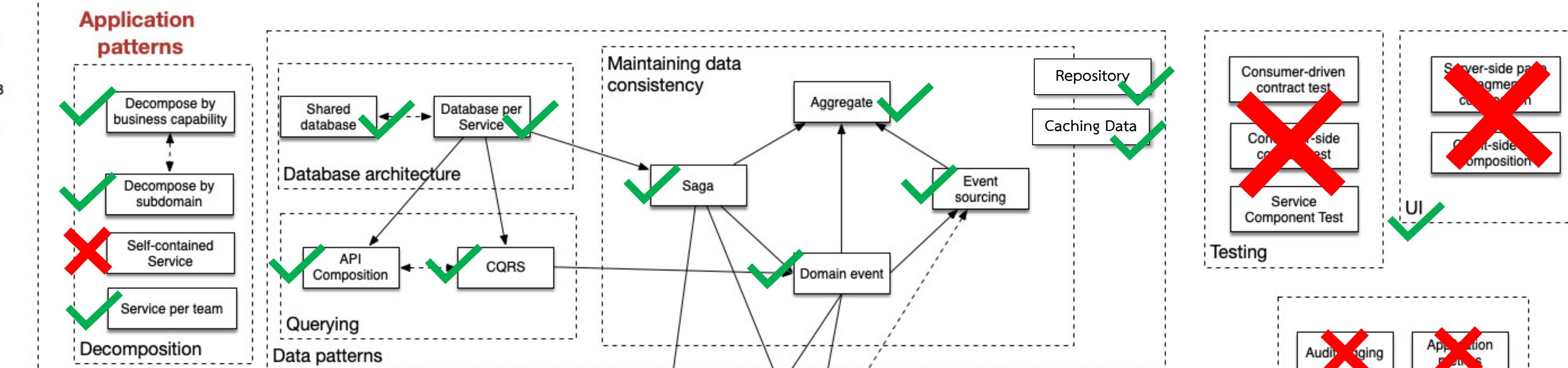
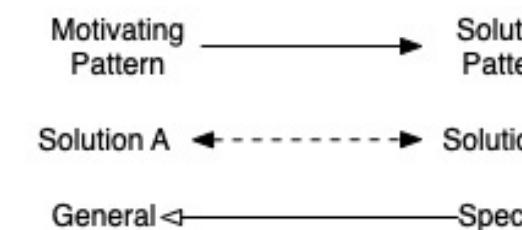


# กรณีศึกษา Netflix

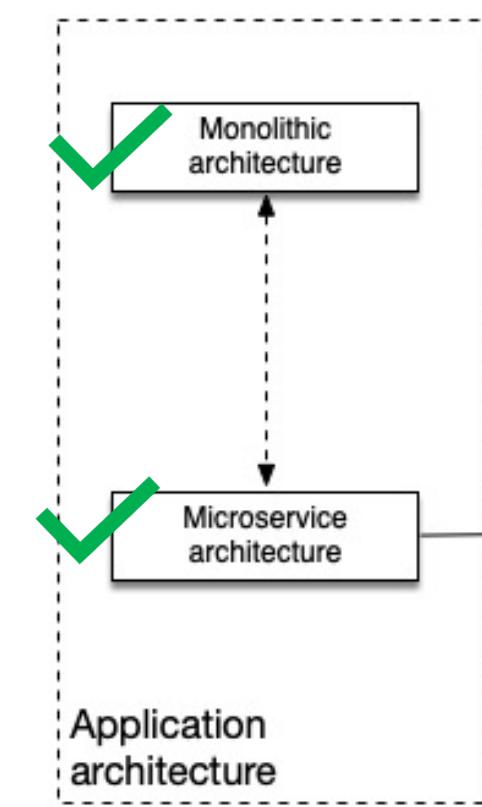


(Josh Evans, Netflix)

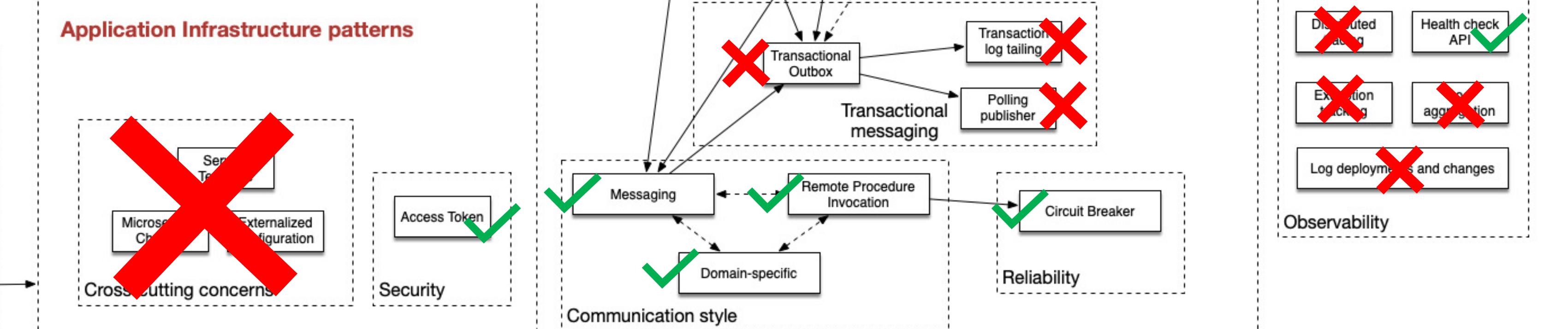




Auditing  
Application metrics

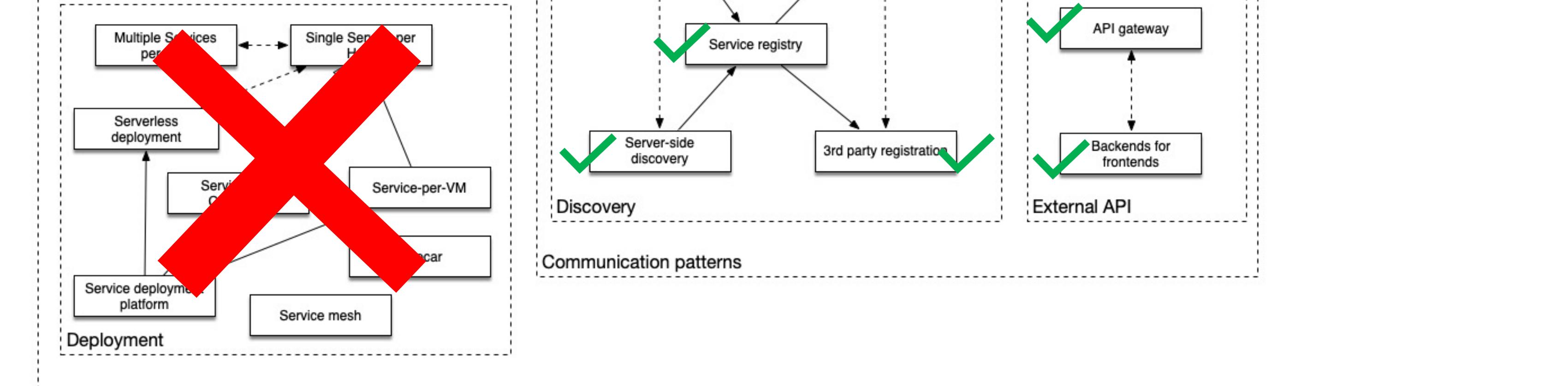


### Application Infrastructure patterns



Application architecture

### Infrastructure patterns

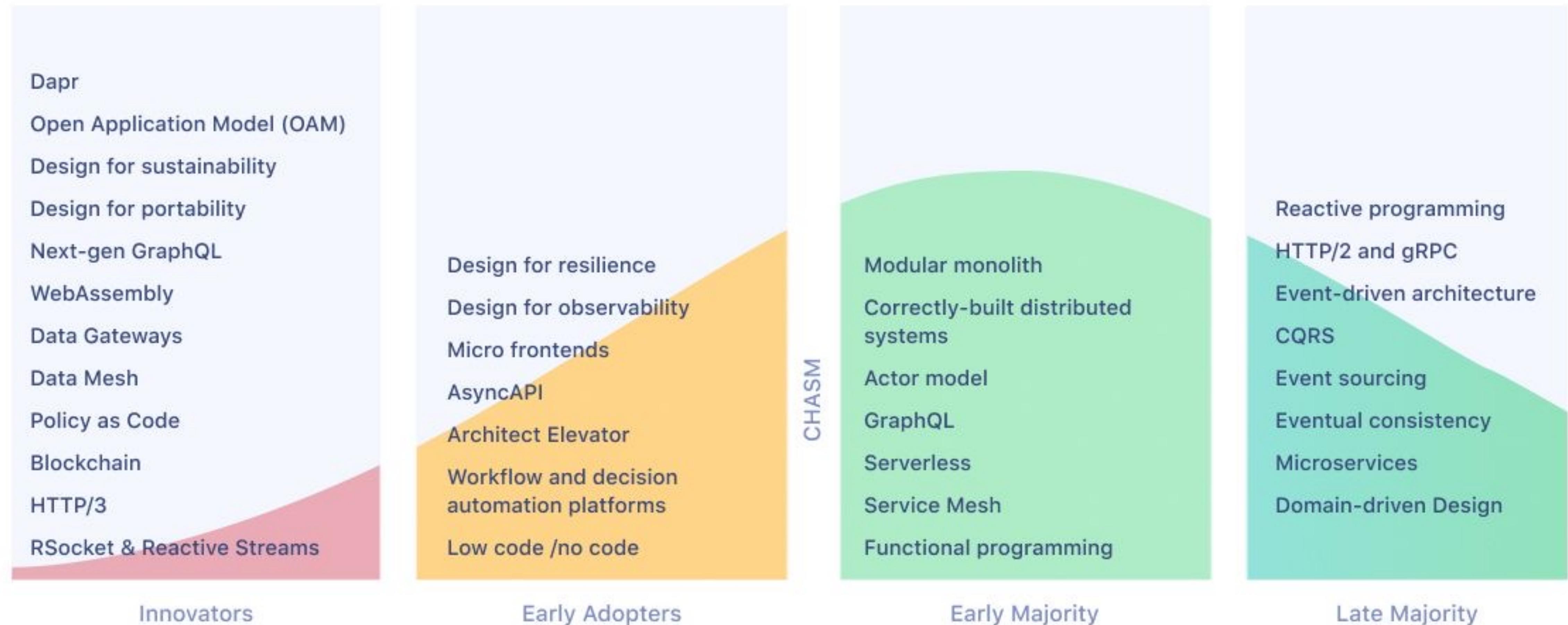


Microservice patterns

## Software Development Architecture and Design 2021 Graph

<http://infoq.link/architecture-trends-2021>

**InfoQ**



## Software Development Architecture and Design 2022 Graph

<http://infoq.link/architecture-trends-2022>

**InfoQ**

