Microservice Architecture

COMP2511, CSE, UNSW

These lecture slides are from the book “*Head First Software Architecture*”,

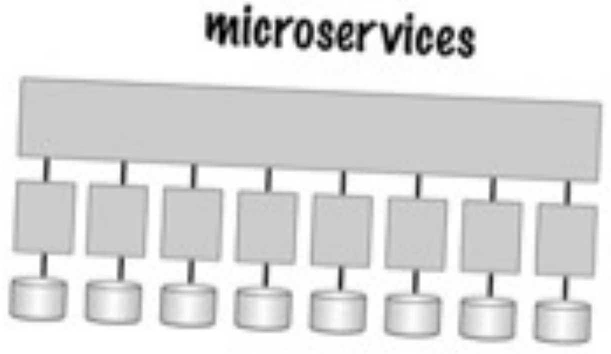
by Raju Gandhi, Mark Richards, Neal Ford, O'Reilly Media, Inc., March 2024

Introduction to Microservices

- ❖ Microservices are single-purpose, independently deployed units.
- ❖ Ideal for environments requiring frequent changes and scalability.

Examples:

- ○ Netflix's streaming services
- ○ Amazon's product catalogue.

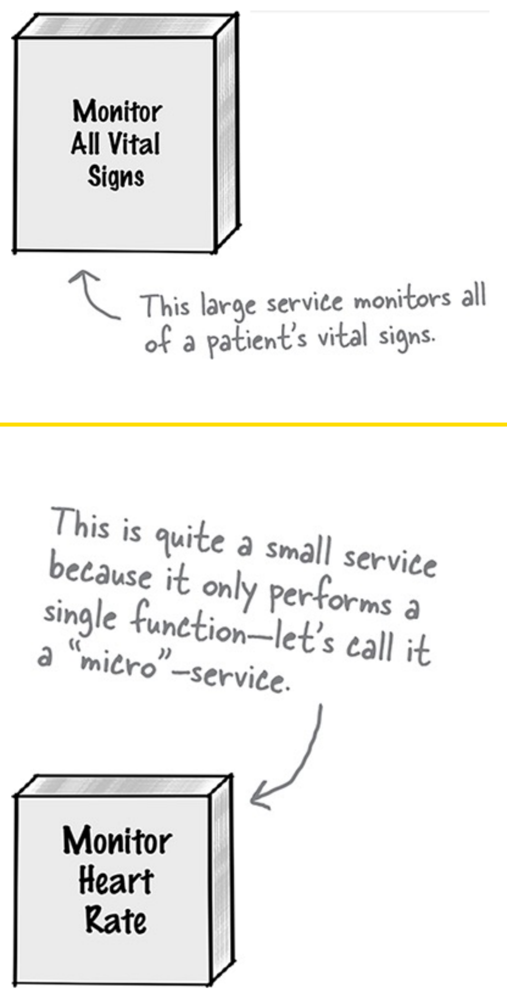


Defining Microservices

❖ Performs one specific function exceptionally well.

Examples:

- Dedicated microservice like "*Monitor Heart Rate.*"
- "Authenticate User" service, "*Generate Invoice*" service.
- "User Profile Management" service.
- "Shopping Cart" service.
- "Notification and Alert" service.
- "Recommendation Engine" service (e.g., Netflix recommendations).


Monitor All Vital Signs
This large service monitors all of a patient's vital signs.
This is quite a small service because it only performs a single function—let's call it a "micro"-service.
Monitor Heart Rate

Exercise: Define Microservices

Identify single-purpose microservices below:

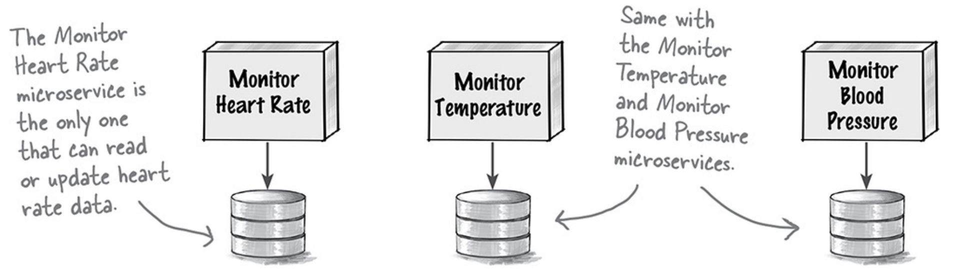
- ❑ Add a movie to your personal “to watch” list
- ❑ Pay for an order using your credit card
- ❑ Generate sales-forecasting and financial-performance reports
- ❑ Submit and process a loan application to get that new car you’ve always wanted
- ❑ Determining the shipping cost for an online order

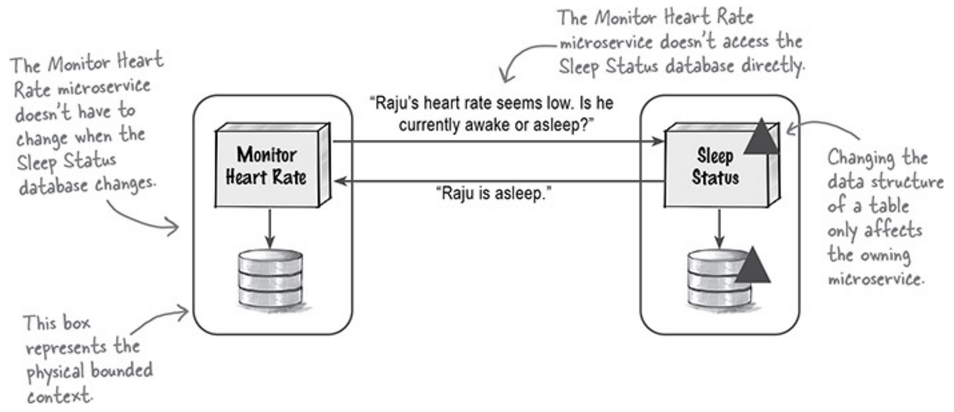
Key Characteristics of Microservices

- ❖ Own their own data (Physical bounded contexts).
- ❖ Direct data access restricted to owning microservice.

Examples:

- Order service maintains its own order history database.
- Inventory service owns and manages product availability data.
- Payment service manages transaction records independently.
- User Authentication service securely stores user credentials separately.


The Monitor Heart Rate microservice is the only one that can read or update heart rate data.
Monitor Heart Rate
Monitor Temperature
Same with the Monitor Temperature and Monitor Blood Pressure microservices.
Monitor Blood Pressure

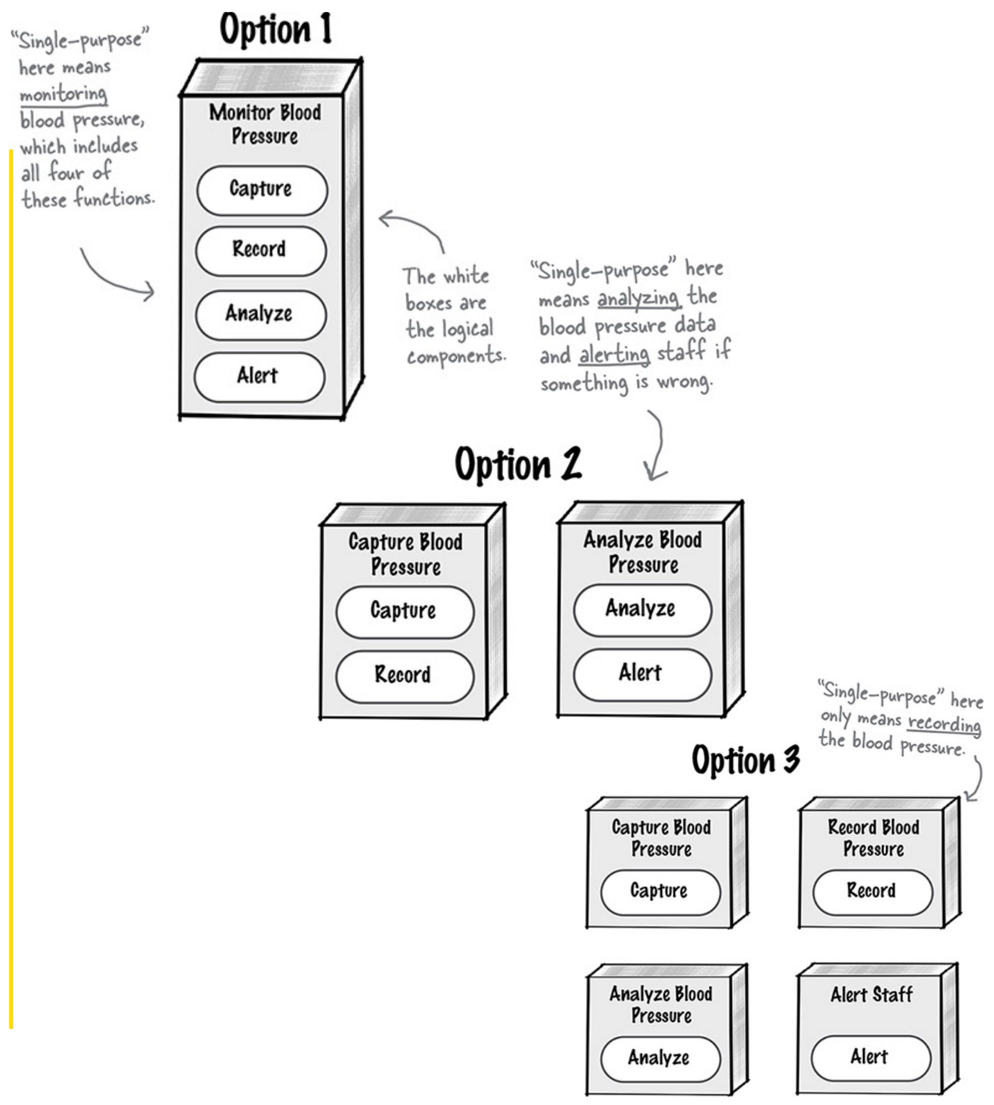

The Monitor Heart Rate microservice doesn't have to change when the Sleep Status database changes.
The Monitor Heart Rate microservice doesn't access the Sleep Status database directly.
"Raju's heart rate seems low. Is he currently awake or asleep?"
"Raju is asleep."
Monitor Heart Rate
Sleep Status
Changing the data structure of a table only affects the owning microservice.
This box represents the physical bounded context.

Determining Granularity

- ❖ Granularity: The scope of a microservice's responsibility.
- ❖ Avoid too fine-grained ("Grains of Sand" antipattern).

Examples:

- Single microservice handling payment transactions.
- A microservice dedicated to shipping and tracking orders.
- Product review and rating as a distinct service separate from product information.
- User notification service isolated from user profile management



"Single–purpose" here means monitoring blood pressure, which includes all four of these functions.
Option 1
Monitor Blood Pressure
Capture
Record
Analyze
Alert
The white boxes are the logical components.
"Single–purpose" here means analyzing the blood pressure data and alerting staff if something is wrong.
Option 2
Capture Blood Pressure
Capture
Record
Analyze Blood Pressure
Analyze
Alert
"Single–purpose" here only means recording the blood pressure.
Option 3
Capture Blood Pressure
Capture
Record Blood Pressure
Record
Analyze Blood Pressure
Analyze
Alert Staff
Alert

Granularity Disintegrators
(Reasons to Make Services Smaller)

Cohesion: Functions within a service should be closely related.

- Payment processing separate from user authentication.

Fault Tolerance: Separating unstable functions for better reliability.

- Isolating an unstable email notification service.

Access Control: Easier management of sensitive data.

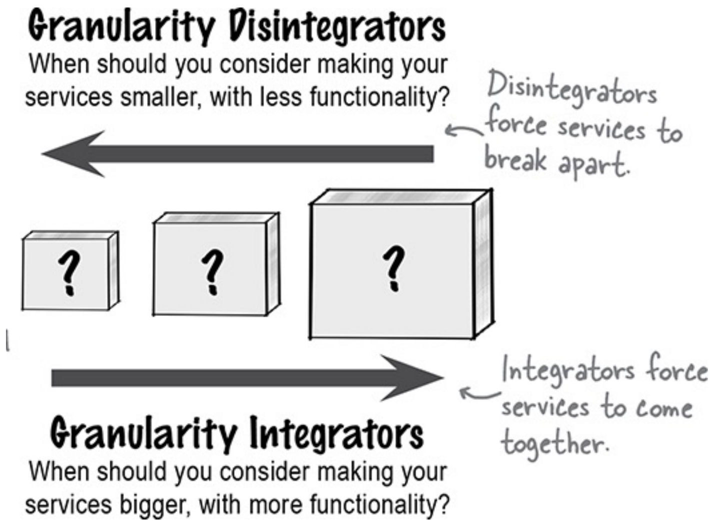
- Isolating financial data access.

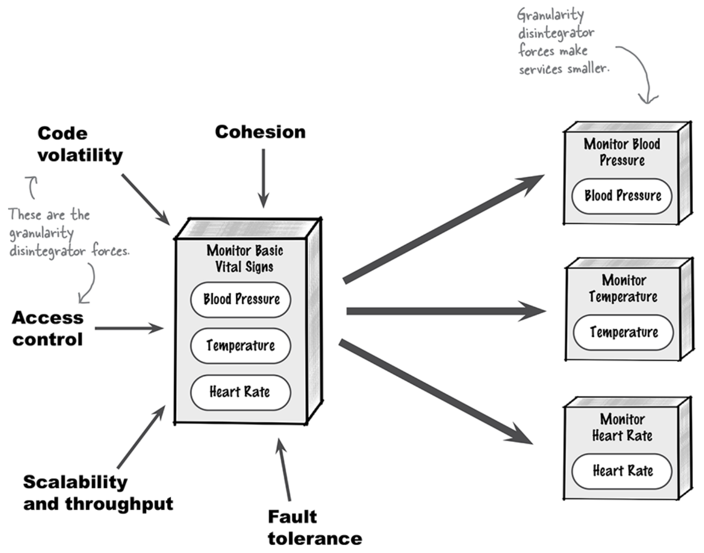
Code Volatility: Isolating frequently changing parts.

- User interface components separated from stable backend logic.

Scalability: Independent scaling for high-demand components.

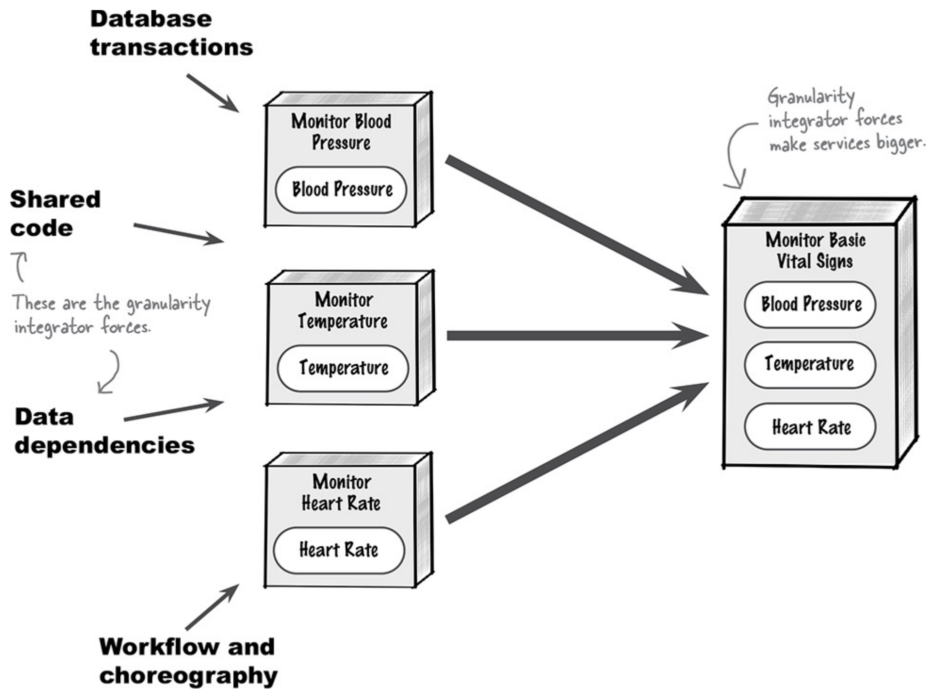
- High-traffic "search" feature isolated for scaling.


Granularity Disintegrators
When should you consider making your services smaller, with less functionality?
Disintegrators force services to break apart.
?
?
?
Integrators force services to come together.
Granularity Integrators
When should you consider making your services bigger, with more functionality?

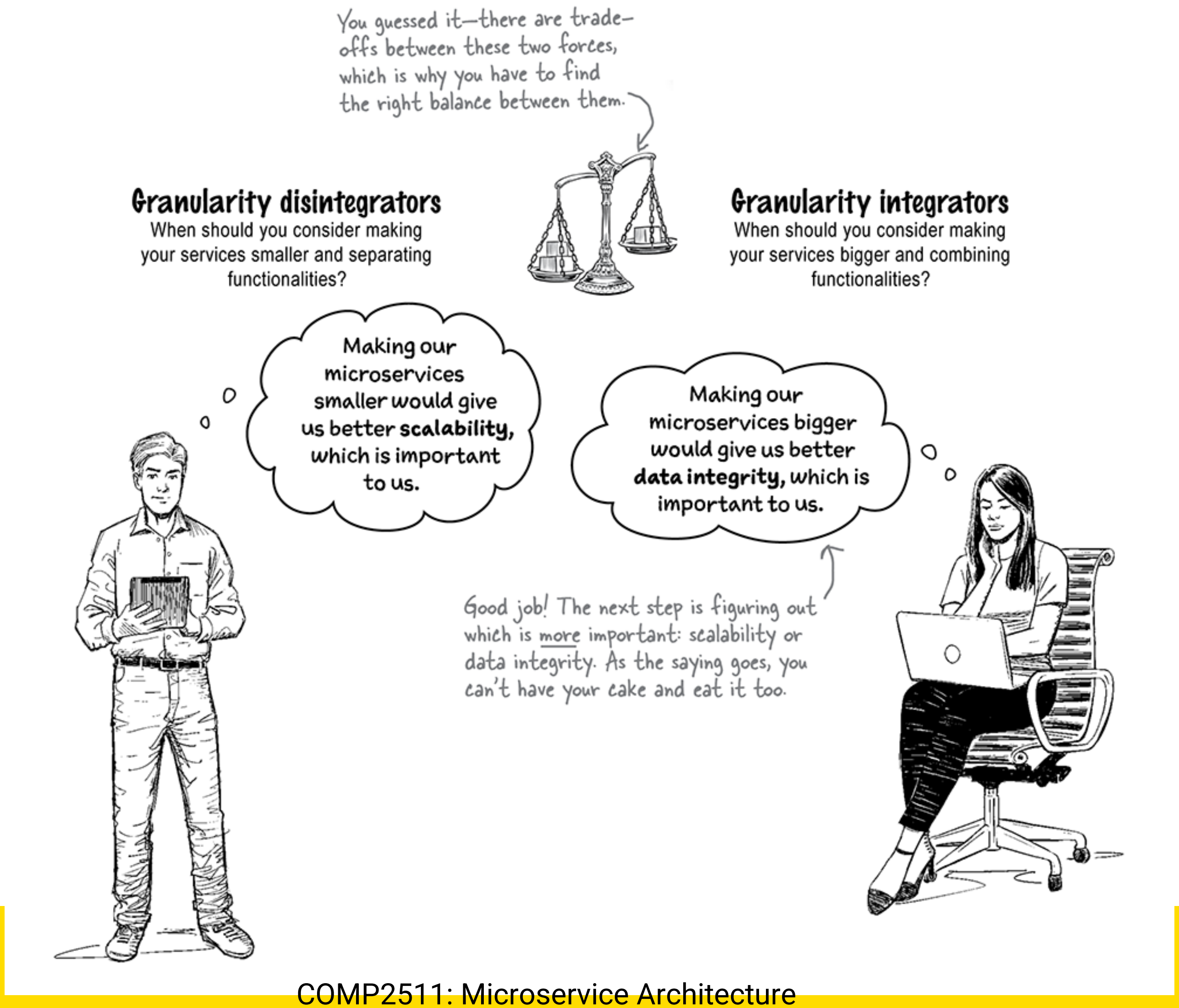


Granularity Integrators
(Reasons to Make Services Larger)

- Database Transactions: Easier to manage single commit/rollback operations.
  - Order creation and inventory deduction in one service.
- Data Dependencies: Maintain tightly coupled data together.
  - User profiles and preferences managed together.
- Workflow Efficiency: Reduce excessive inter-service communication.
  - Checkout service combining cart, pricing, and payment functionalities.

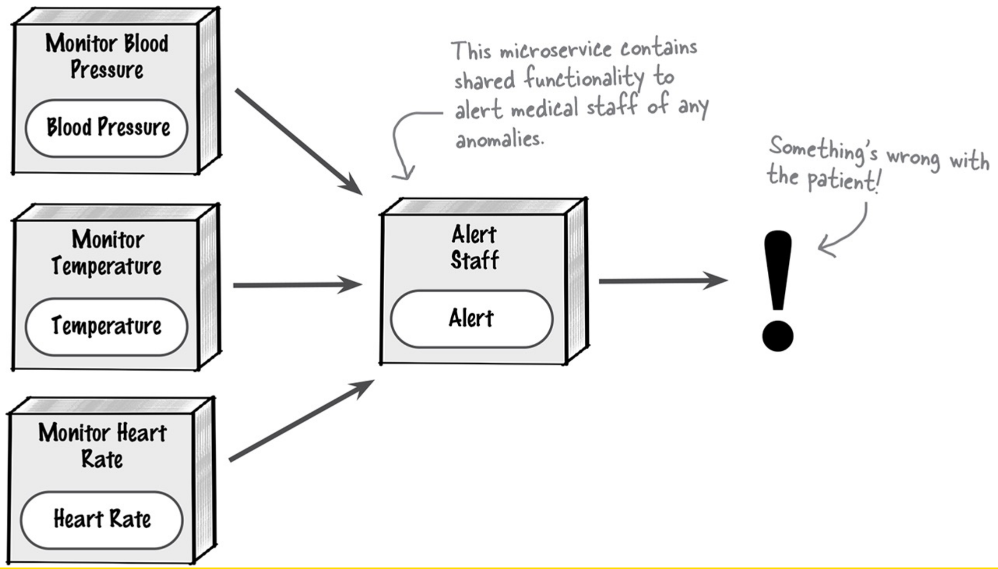


It’s about a right balance!


You guessed it—there are trade-offs between these two forces, which is why you have to find the right balance between them.
Granularity disintegrators When should you consider making your services smaller and separating functionalities?
Granularity integrators When should you consider making your services bigger and combining functionalities?
Making our microservices smaller would give us better **scalability,** which is important to us.
Making our microservices bigger would give us better **data integrity,** which is important to us.
Good job! The next step is figuring out which is more important: scalability or data integrity. As the saying goes, you can't have your cake and eat it too.

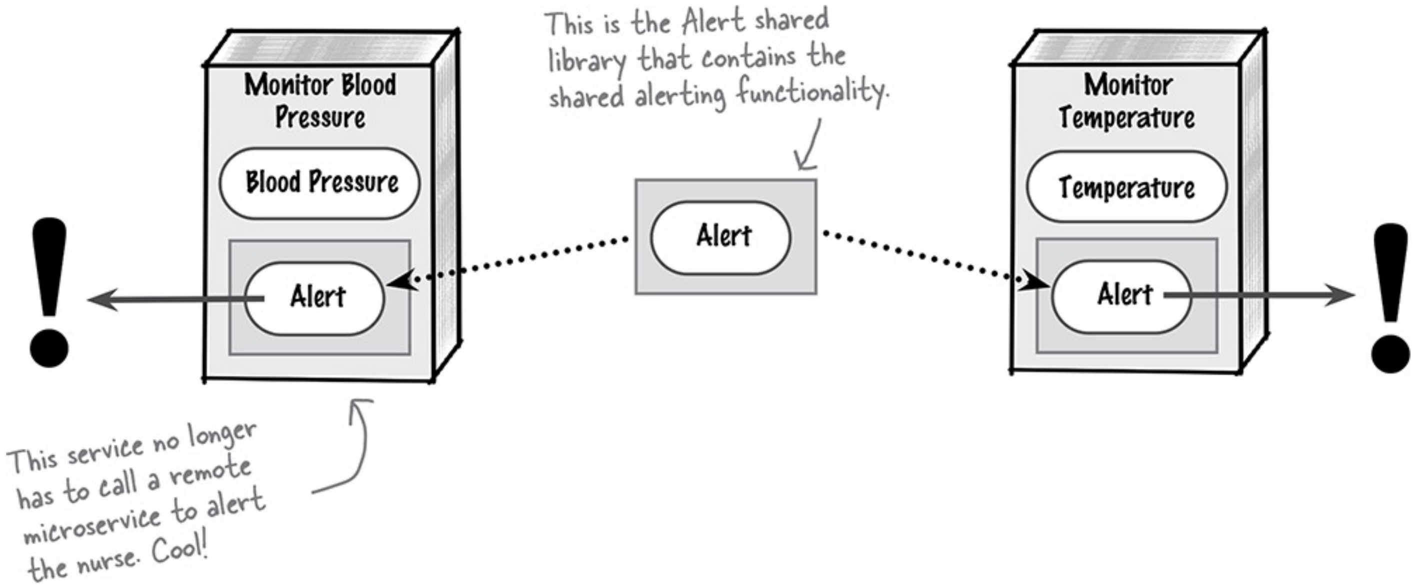
Sharing Functionality

- Shared Services: Standalone microservices accessed remotely.
  - Authentication service used by multiple microservices.
  - Shared alert functionality in *MonitorMe* medical alerts


Monitor Blood Pressure
Blood Pressure
Monitor Temperature
Temperature
Monitor Heart Rate
Heart Rate
This microservice contains shared functionality to alert medical staff of any anomalies.
Alert Staff
Alert
Something's wrong with the patient!
!

Sharing Functionality

- Shared Libraries: Embedded at compile-time, deployed with each service.
  - Logging and error handling libraries.


This is the Alert shared library that contains the shared alerting functionality.
Monitor Blood Pressure
Blood Pressure
Alert
Alert
Monitor Temperature
Temperature
Alert
!
!
This service no longer has to call a remote microservice to alert the nurse. Cool!

Shared Services vs. Shared Libraries

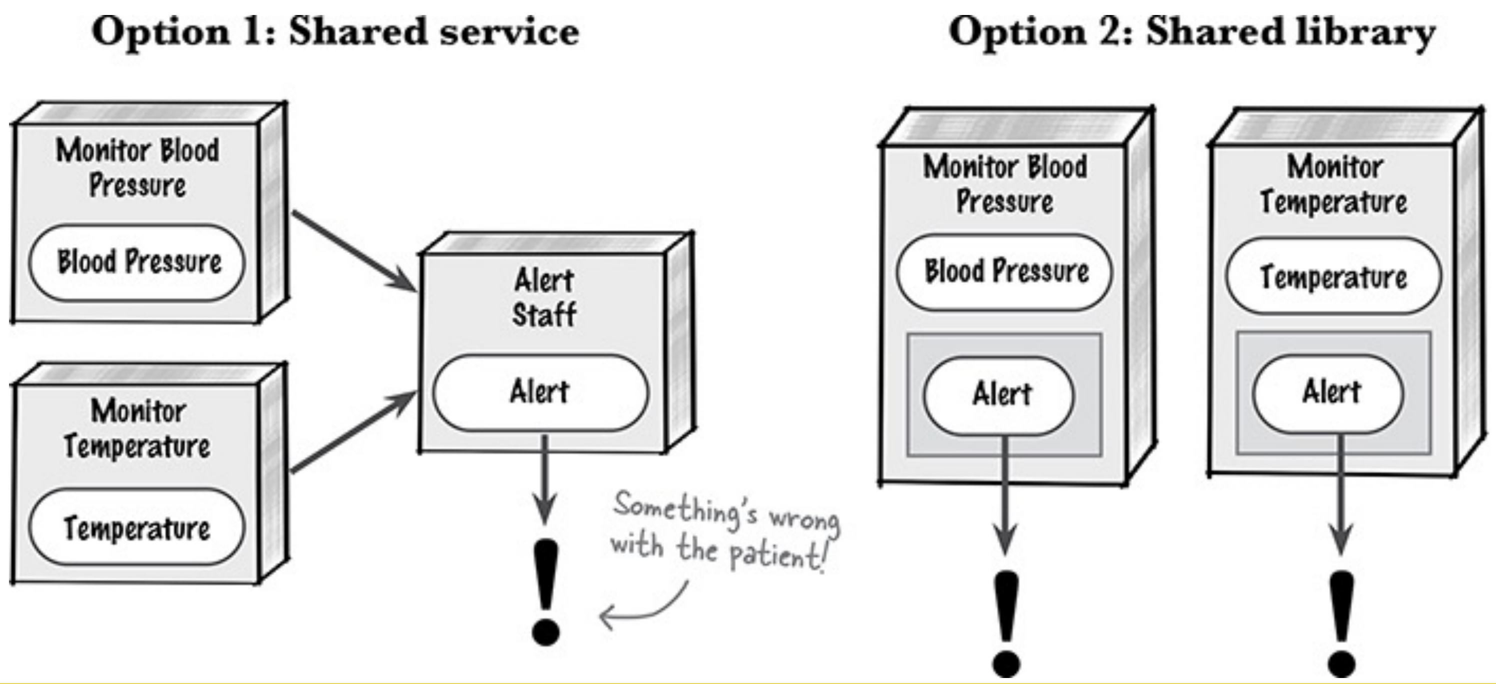
- Services: Agile, suitable for diverse environments, slower, less fault-tolerant.
  - Central user authentication service.

- Libraries: Faster, scalable, robust, but challenging dependency management.
  - JSON parsing libraries used across multiple microservices.

Exercise

Should the alert functionality in *MonitorMe* be a shared library or a shared service?

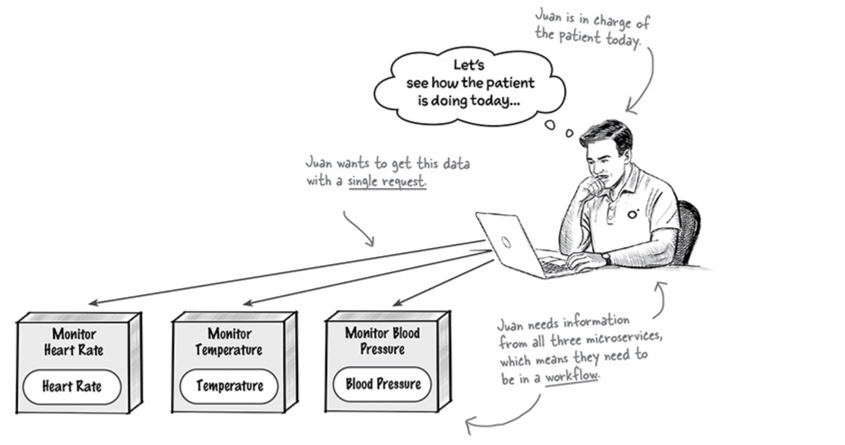
- Justify your decision.



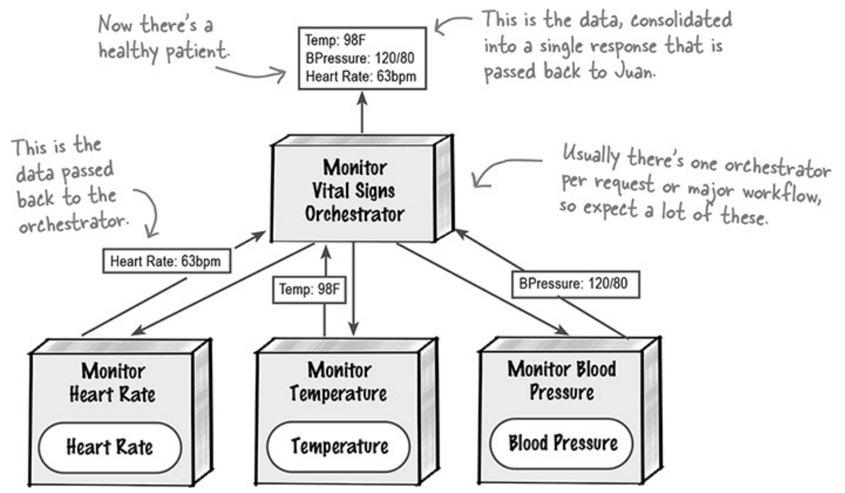
Workflow Management: Orchestration

- Central orchestration manages workflow, akin to a symphony conductor.
  - Pros: Centralized management, clear state/error handling.
  - Cons: Bottlenecks, high coupling, performance concerns.

- Example:
  - Centralised order processing orchestrating payment, inventory, and shipment services.



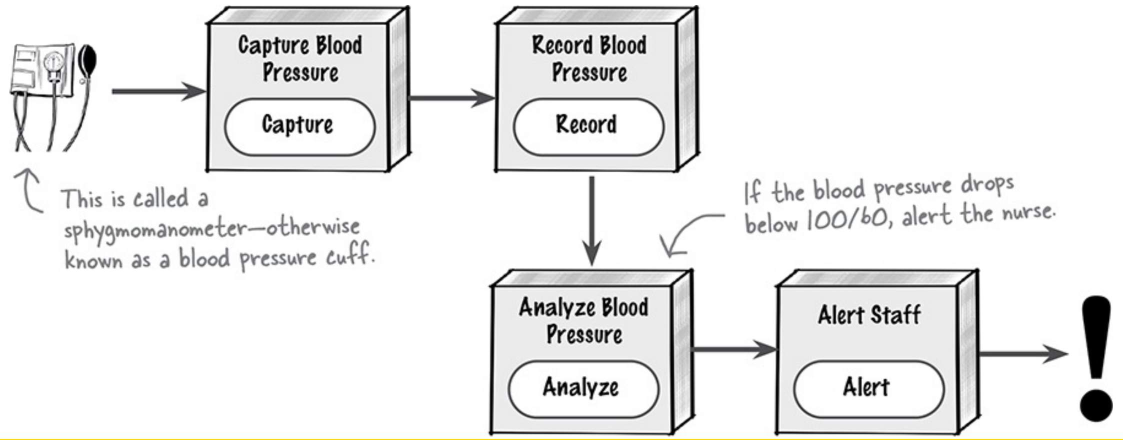
Juan is in charge of the patient today.
Let's see how the patient is doing today...
Juan wants to get this data with a single request.
Monitor Heart Rate
Heart Rate
Monitor Temperature
Temperature
Monitor Blood Pressure
Blood Pressure
Juan needs information from all three microservices, which means they need to be in a workflow.



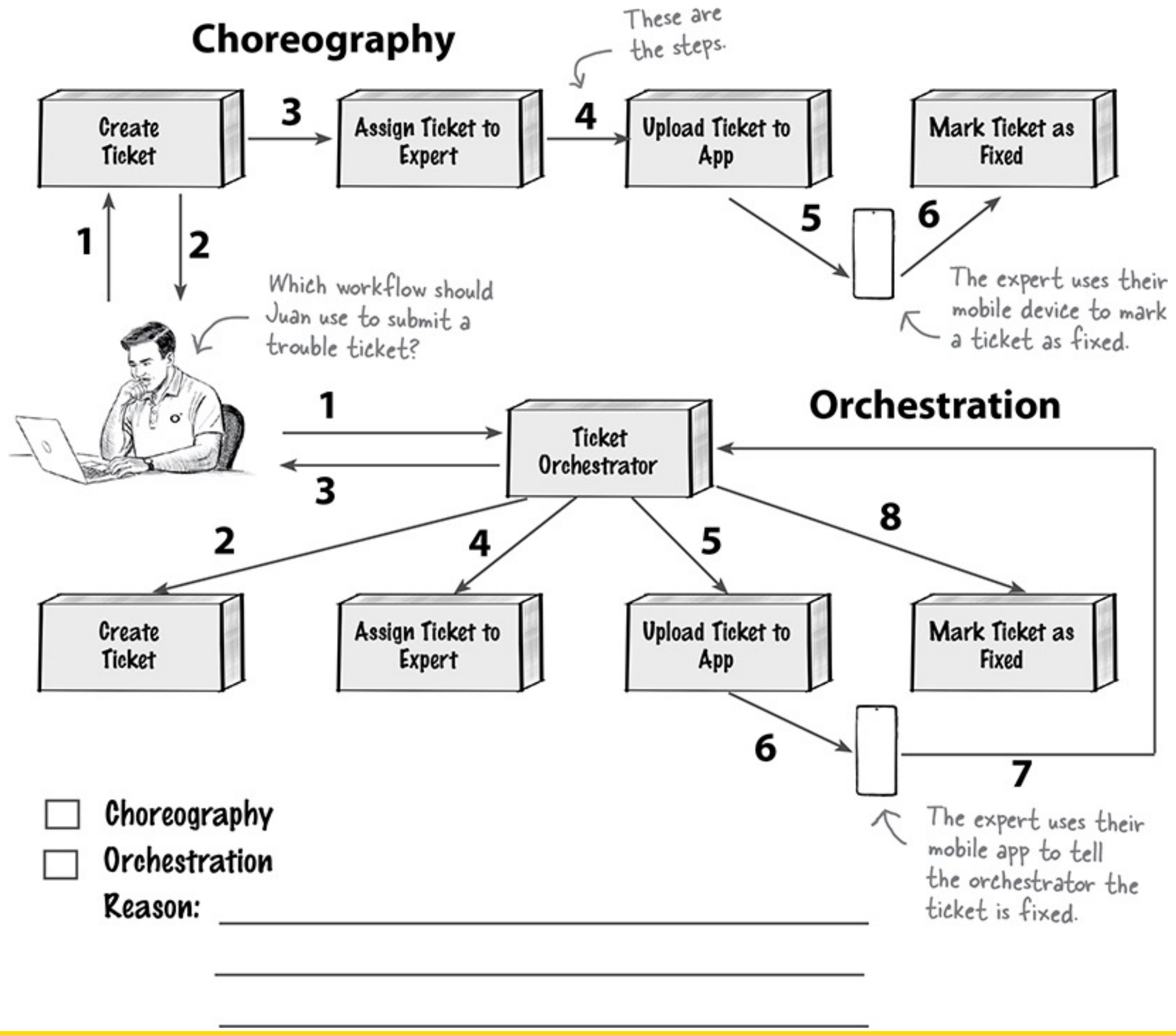
Now there's a healthy patient.
Temp: 98F
BPressure: 120/80
Heart Rate: 63bpm
This is the data, consolidated into a single response that is passed back to Juan.
This is the data passed back to the orchestrator.
Monitor Vital Signs Orchestrator
Usually there's one orchestrator per request or major workflow, so expect a lot of these.
Heart Rate: 63bpm
Temp: 98F
BPressure: 120/80
Monitor Heart Rate
Heart Rate
Monitor Temperature
Temperature
Monitor Blood Pressure
Blood Pressure

Workflow Management: Choreography

- Peer-to-peer service communication, like coordinated dance.
  - Pros: Scalable, loosely coupled, high responsiveness.
  - Cons: Complex error and state management.
- Example:
  - Event-driven updates between cart, inventory, and shipping services in an e-commerce site.


Capture Blood Pressure
Capture
Record Blood Pressure
Record
This is called a sphygmomanometer—otherwise known as a blood pressure cuff.
If the blood pressure drops below 100/60, alert the nurse.
Analyze Blood Pressure
Analyze
Alert Staff
Alert
!

Exercise



☐ Choreography

☐ Orchestration

Reason: ______________________________

____

____

Advantages of Microservices

- Maintainability, Testability, Deployability, Evolvability.
- Exceptional scalability and fault tolerance.

- Examples:
  - Continuous deployment at Spotify
  - Scalable services at Netflix

Limitations of Microservices

- Complexity, especially in workflow management.
- Performance issues due to inter-service communications.

- Example:
  - Increased latency in highly interactive systems like gaming or real-time analytics platforms.

Balancing Microservices Architecture

- Decision criteria:
  - Business agility
  - Complexity handling
  - Team structure

- Optimal balance between granular control and practical maintainability.

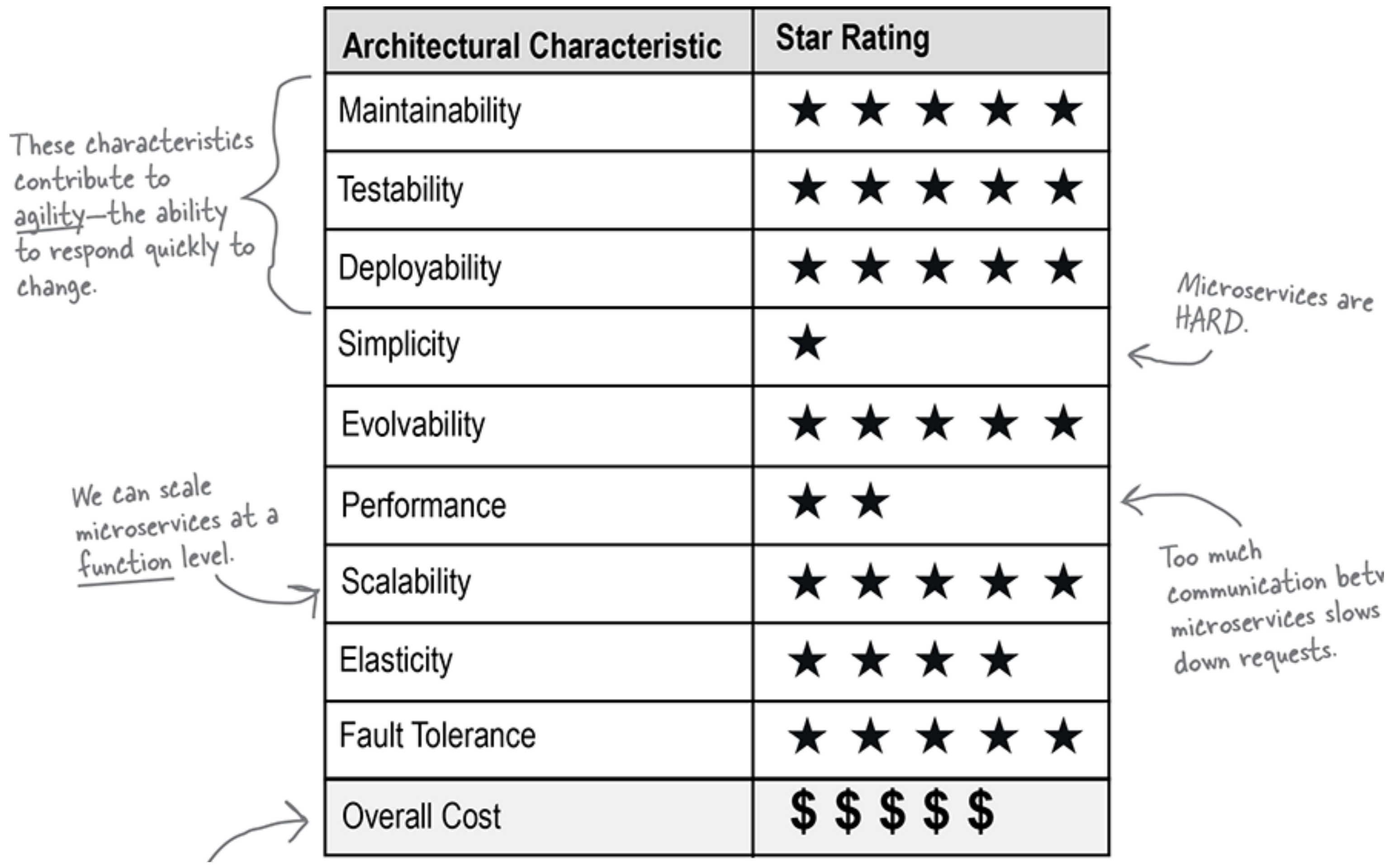
- Example:
  - Amazon's product catalog services balancing granularity and maintainability.

Case Study - StayHealthy MonitorMe

- Successful real-world implementation of microservices.
- *Insights*: Balance granularity, effectively manage shared resources.
- Continuous focus on agility and operational stability.

- Example:
  - Reliable and scalable health monitoring system for critical patient data.

Microservices Star Ratings



Architectural Characteristic	Star Rating
Maintainability	★ ★ ★ ★ ★
Testability	★ ★ ★ ★ ★
Deployability	★ ★ ★ ★ ★
Simplicity	★
Evolvability	★ ★ ★ ★ ★
Performance	★ ★
Scalability	★ ★ ★ ★ ★
Elasticity	★ ★ ★ ★
Fault Tolerance	★ ★ ★ ★ ★
Overall Cost	$ $ $ $ $

These characteristics contribute to agility—the ability to respond quickly to change.

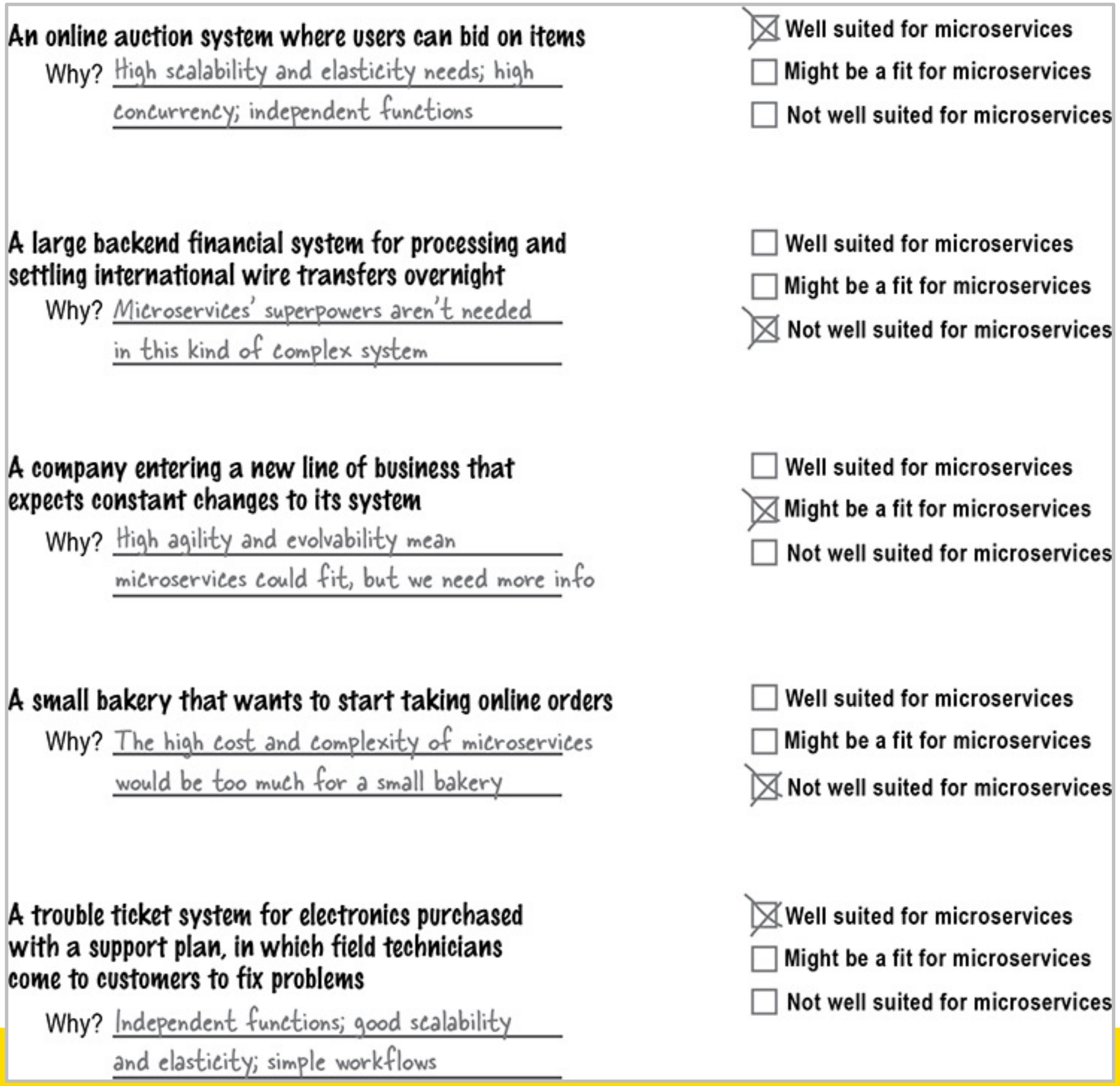
Microservices are HARD.

We can scale microservices at a function level.

Too much communication betw microservices slows down requests.

Exercise

Which of the following systems might be well suited for the microservices architectural style, and why?



An online auction system where users can bid on items

Why? High scalability and elasticity needs; high concurrency; independent functions

- ☒ Well suited for microservices
- ☐ Might be a fit for microservices
- ☐ Not well suited for microservices

A large backend financial system for processing and settling international wire transfers overnight

Why? Microservices' superpowers aren't needed in this kind of complex system

- ☐ Well suited for microservices
- ☐ Might be a fit for microservices
- ☒ Not well suited for microservices

A company entering a new line of business that expects constant changes to its system

Why? High agility and evolvability mean microservices could fit, but we need more info

- ☐ Well suited for microservices
- ☒ Might be a fit for microservices
- ☐ Not well suited for microservices

A small bakery that wants to start taking online orders

Why? The high cost and complexity of microservices would be too much for a small bakery

- ☐ Well suited for microservices
- ☐ Might be a fit for microservices
- ☒ Not well suited for microservices

A trouble ticket system for electronics purchased with a support plan, in which field technicians come to customers to fix problems

Why? Independent functions; good scalability and elasticity; simple workflows

- ☒ Well suited for microservices
- ☐ Might be a fit for microservices
- ☐ Not well suited for microservices

Summary

- Microservices offer high flexibility but involve significant complexity.
- Requires crucial granularity and communication decisions.
- Evaluate and manage trade-offs carefully.

- Example:
  - Transitioning from monoliths to microservices at Uber.