



Monolith Decomposition

Java Accelerator 7

Lesson 5.4





WELCOME

Learning Outcomes

By the end of this lesson, you will be able to:



Explain the principle of independent deployability.



Compare and contrast monoliths versus microservices.

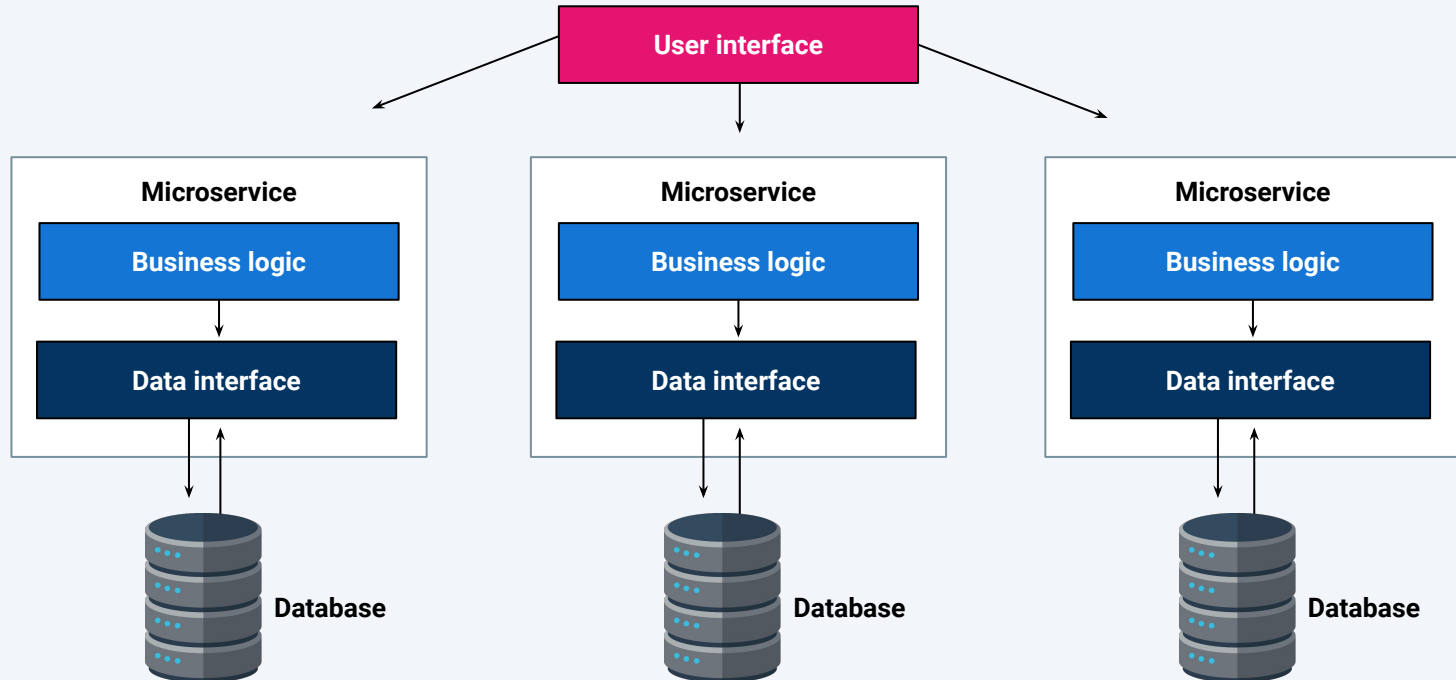


Explain the patterns of decomposition.

Independent Deployability

Independent Deployability

The principle of independent deployability means that each microservice must be deployable completely independently of all other microservices.





The key point is not that it is possible to deploy a microservice independently but that this is the normal way you deploy changes.

Database Considerations

Independent deployability isn't just something you think about when designing microservices. It should be THE way changes to your microservices are deployed.

It's still important to design your microservices in a way that supports this principle:



Services are loosely coupled.



Well-defined interfaces exist between services.



Database Considerations

Database Considerations

Imagine that we have an e-commerce site where we are selling products.

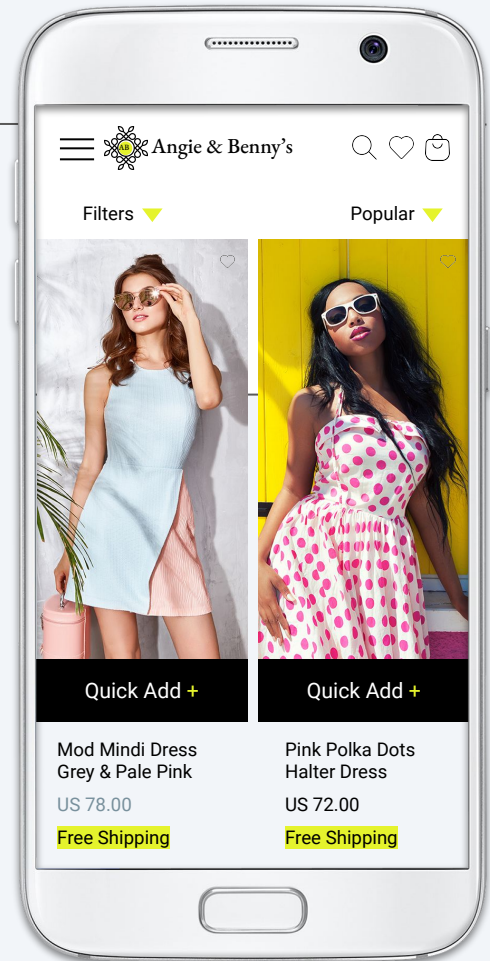
We have designed this site using microservices:

Catalog Microservice

Keeps track of all of the products we sell and their prices, and an Invoicing microservice lets customers place orders.

Invoicing Microservice

Allows customers to place orders.





**How do we work with shared
data in microservices?**

Database Considerations

Ways we can work with shared data in microservices:

01

Each microservice owns its own tables but could let other microservices read from them.

02

Each microservice could provide a copy of its data for other microservices to use.

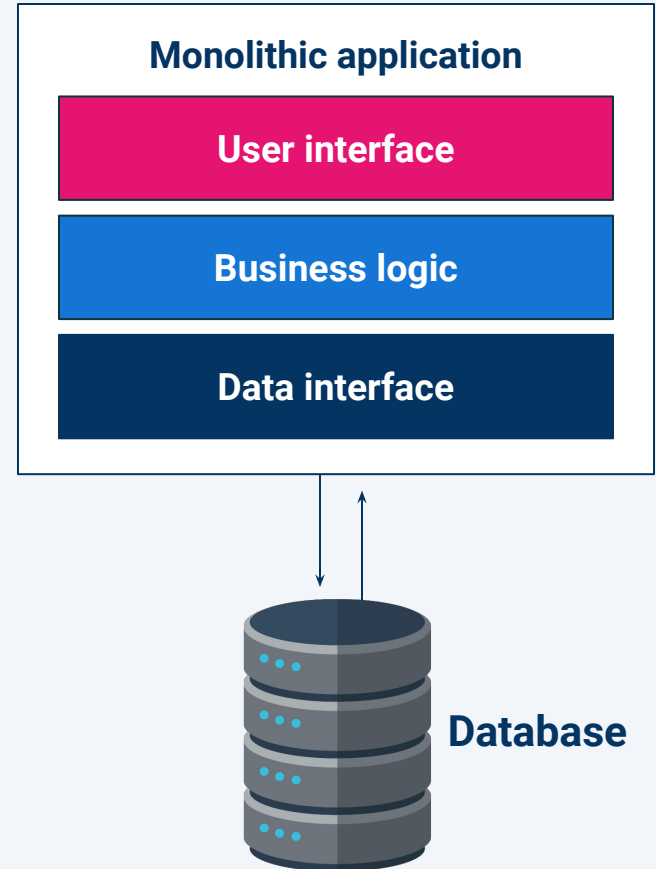
03

Each microservice could provide an interface for other microservices to access data.

Monoliths vs. Microservices

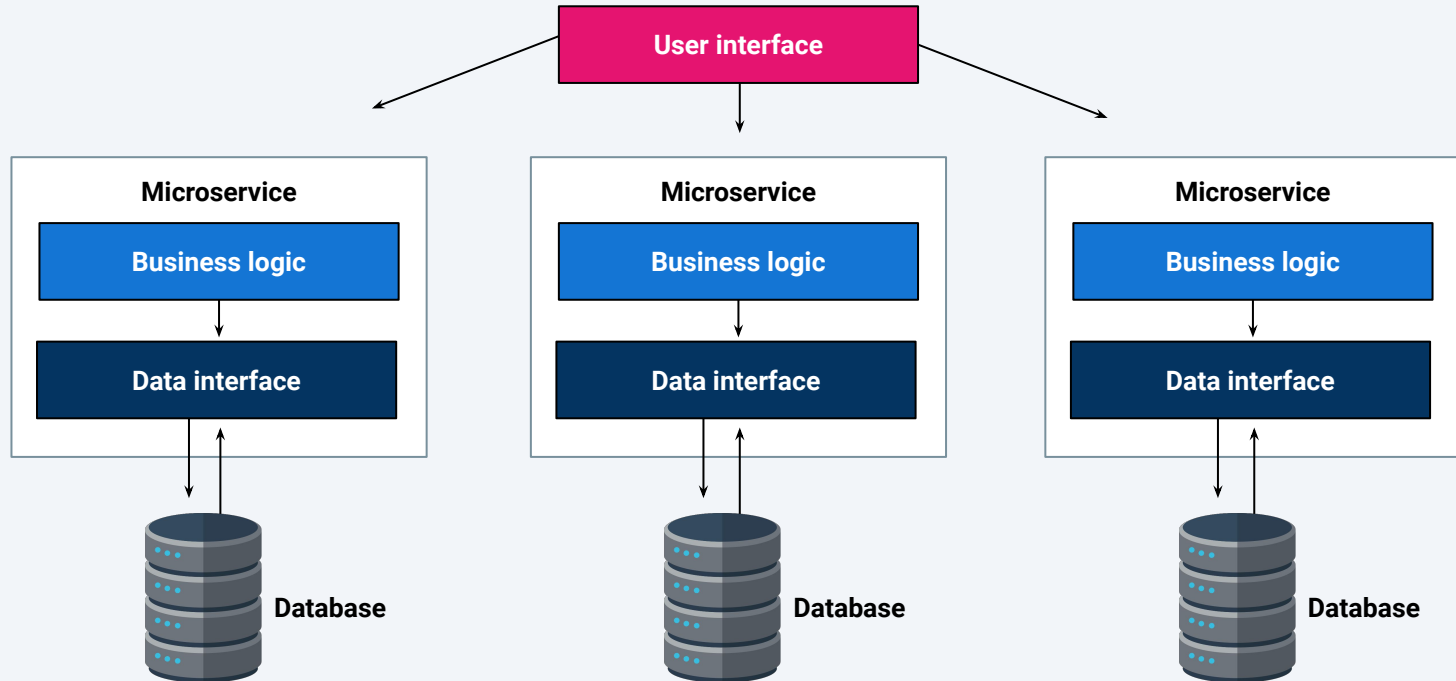
Monolithic Architecture

In a monolithic application, everything is owned by the monolith. The monolith owns and is responsible for all of the entities and the data that represents those entities.



Microservice Architecture

With a microservice architecture, we are splitting a monolith into separate and independent microservices.



Microservice Architecture

01

Each microservice will own and be responsible for only a portion of the functionality.

02

We must decide which functionality each microservice provides and which entities it needs to support that functionality.

03

As we define what each microservice provides, we are defining boundaries around the functionality, entities, and data that a microservice owns.

Service Boundaries

Key points in thinking about service boundaries:



Each microservice has its own database.



Each microservice should have only the data it needs.



Each microservice has all of the data it needs.



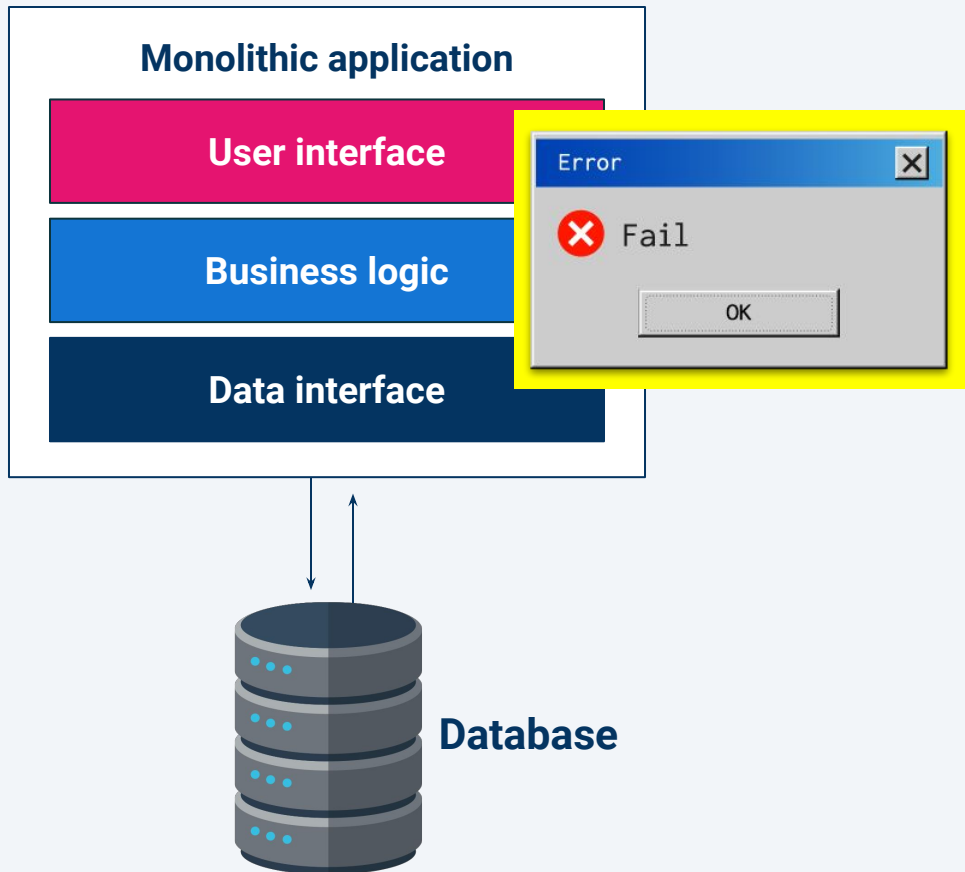
Each microservice is the single source of truth for the data it owns.

Monitoring

Monitoring

In a monolithic application, monitoring is usually straightforward.

The monolith typically writes a log for events. Errors in a monolith can be more obvious in the sense that they have a bigger effect on the application. A severe error might bring the whole application down.





The challenge is often to log enough information so that we can identify where inside the monolith the error has happened, and why.

Monitoring Challenges with Microservice

1

With a microservice architecture, we are splitting a monolith into separate microservices. Each microservice is deployed independently. Already this means we have more places to look.

2

Another challenge is relating data between logs. We might have to use the `productId` from an event in the Invoicing log to relate it back to an event in the Catalog log.

“We replaced our monolith with microservices so that every outage could be more like a murder mystery.”

—Honest Status Page



Reimplementing, Refactoring, Recopying

Reimplementing, Refactoring, Recopying

How do we make changes to the code in a monolithic application?
At a high level, we have three approaches we can take:

01

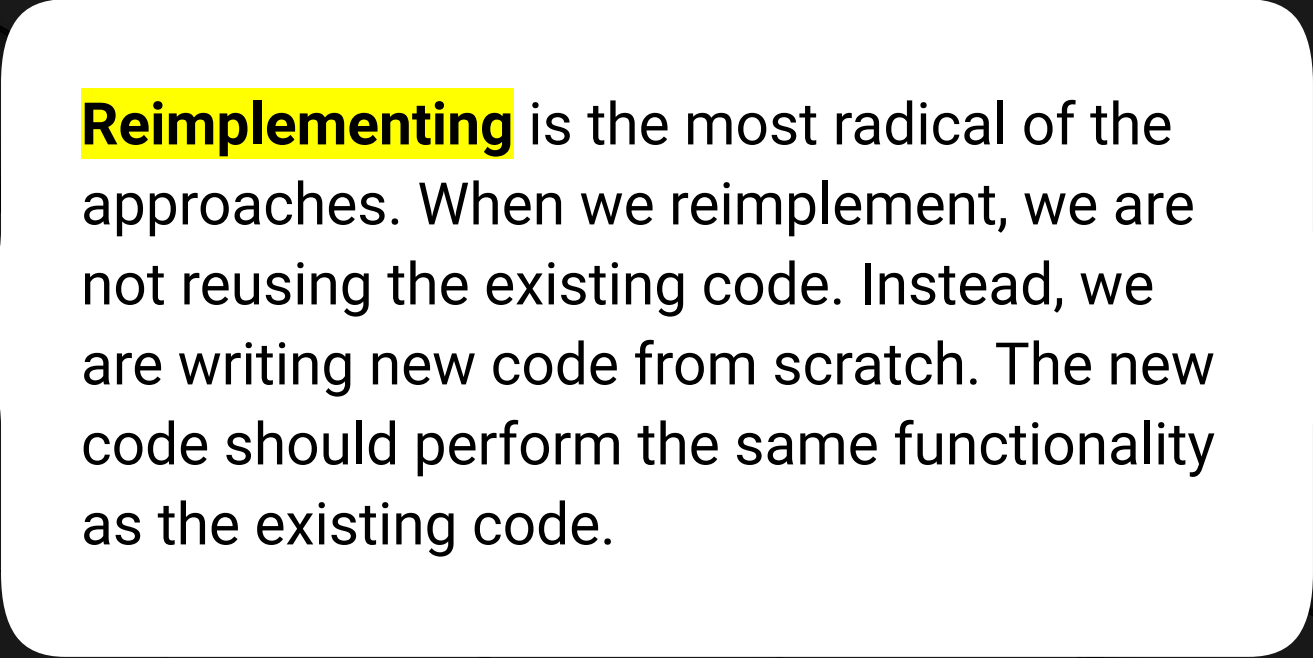
Reimplementing

02

Refactoring

03

Recopying



Reimplementing is the most radical of the approaches. When we reimplement, we are not reusing the existing code. Instead, we are writing new code from scratch. The new code should perform the same functionality as the existing code.



Time to Code

Reimplementing

Suggested Time:

Reimplementing

Pros

We're not locked into a particular technology or programming language or solution.

We can choose whatever makes sense to provide the functionality.

With hindsight, we can make more informed design decisions.

The existing code grew and changed over time. Because we know now what the code is doing, we can make better design decisions and reduce technical debt.

Reimplementing might be the only option.

Depending on how the existing code is structured, it might not be possible to cleanly separate out the functionality.

Cons

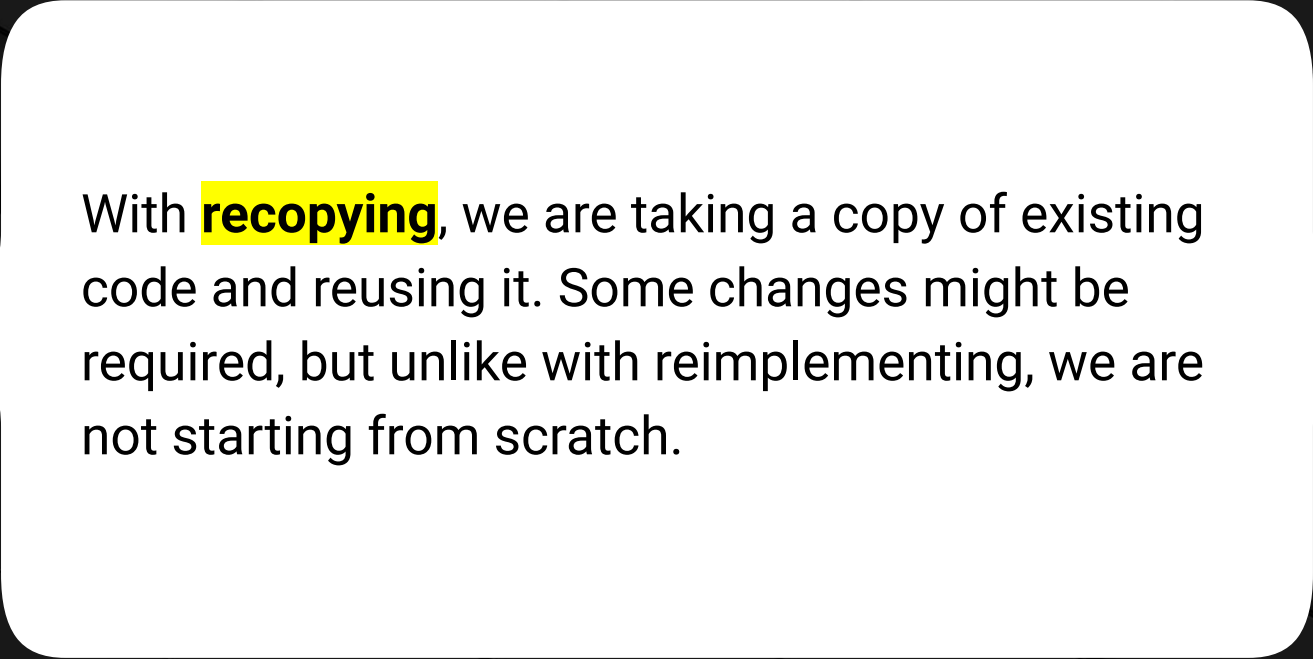
It costs time and money to rewrite code.

Rewriting code means that you're spending time and money to write code that has already been written.

You might introduce new bugs.

The existing code has been tested by QA and by running in production with real data and real users.

It might not be possible to rewrite one part of the application without rewriting all of it.



With **recopying**, we are taking a copy of existing code and reusing it. Some changes might be required, but unlike with reimplementing, we are not starting from scratch.



Time to Code

Recopying

Suggested Time:

Recopying

Pros

You have less code to write.

Code reuse saves time and money.

The code already exists, and it works.

Working code has been tested and does what you expect.

The code is written in a language and with tools that you already understand.

Less learning curve and guesswork involved.

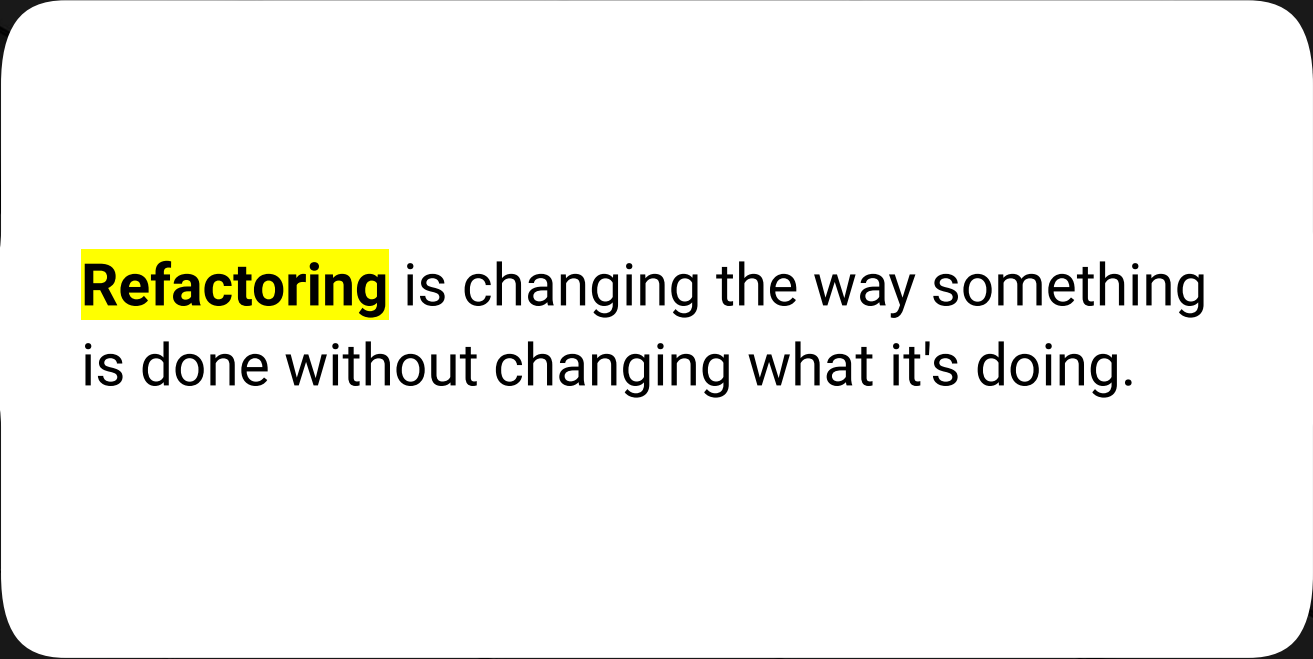
Cons

You're locked into that technology.

Copying the code means you're keeping the technology stack that supports the code.

You're inheriting all of the existing bugs, flaws, and technical debt.

You get the existing code as is.



Refactoring is changing the way something is done without changing what it's doing.

Refactoring

As an example, imagine that we have a function that returns `true` if the first number is less than the second number, but returns `false` otherwise:

```
public boolean min(int num1, int num2) {  
    if (num1 < num2) {  
        return true;  
    } else if (num1 == num2) {  
        return false;  
    } else {  
        return false;  
    }  
}
```

Refactoring

With refactoring, we could change the code to something like this:

```
public boolean min(int num1, int num2) {  
    return num1 < num2;  
}
```




Time to Code

Refactoring

Suggested Time:

Refactoring

Pros

You have less code to write.

Code reuse saves time and money.

The code already exists, and it works.

Working code has been tested and does what you expect.

The code is written in a language and with tools that you already understand.

Less learning curve and guesswork involved.

Cons

You're locked into that technology.

Copying the code means you're keeping the technology stack that supports the code.

You're inheriting all of the existing bugs, flaws, and technical debt.

You get the existing code as is.

Patterns of Decomposition

Decomposition

We use **decomposition** to break down a problem down into smaller, more manageable parts.



Pattern recognition

Once a problem is broken down, we use **pattern recognition** to find similarities and patterns among the smaller parts. This helps us solve the problem more efficiently.



Branch by Abstraction

With the **branch by abstraction** pattern, we'll have two implementations of the same functionality in the same codebase at the same time.

1

The existing implementation will be the active implementation that is used by clients while we develop a new implementation.

2

When the new implementation is ready, we will make it active, and it will replace the existing implementation.

Branch by Abstraction Pattern

To use the branch by abstraction pattern, we'll follow these steps:

01

Create an abstraction for the functionality to replace.

02

Change clients of the existing functionality to use the abstraction.

03

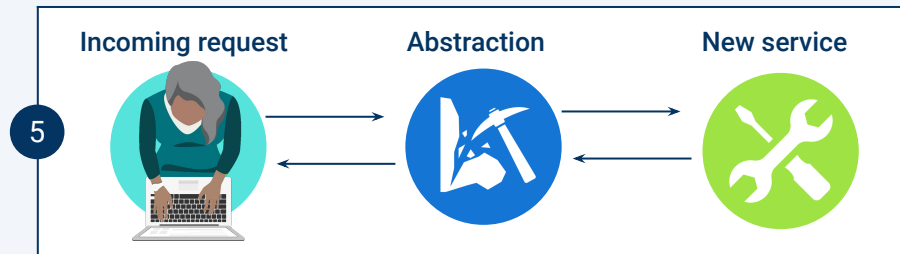
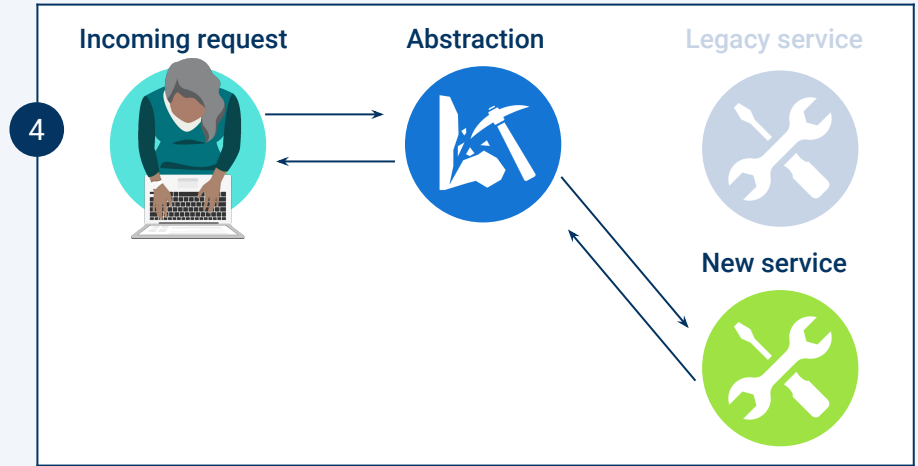
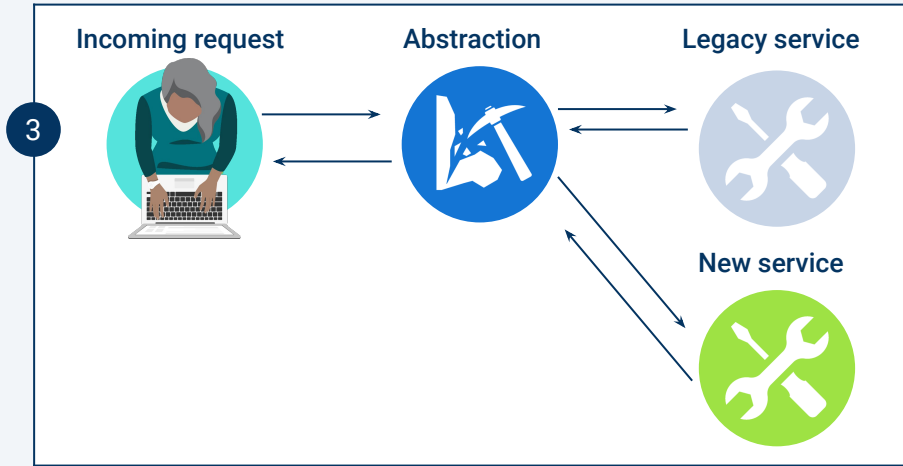
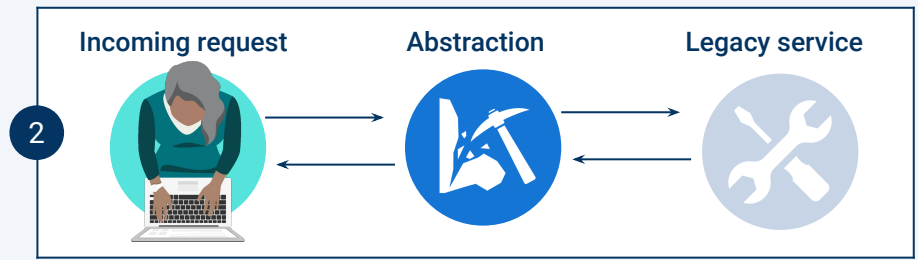
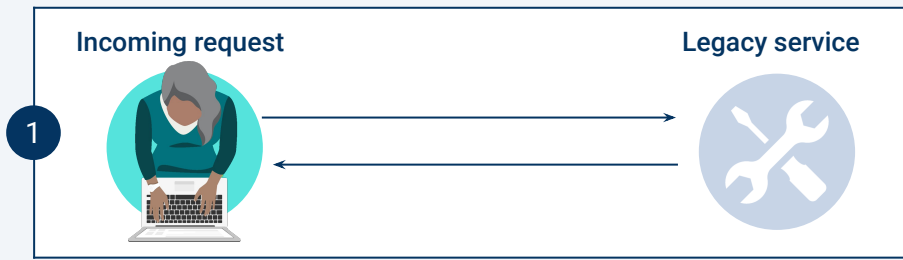
Create a new implementation of the abstraction.

04

When ready, switch clients to use the new implementation.

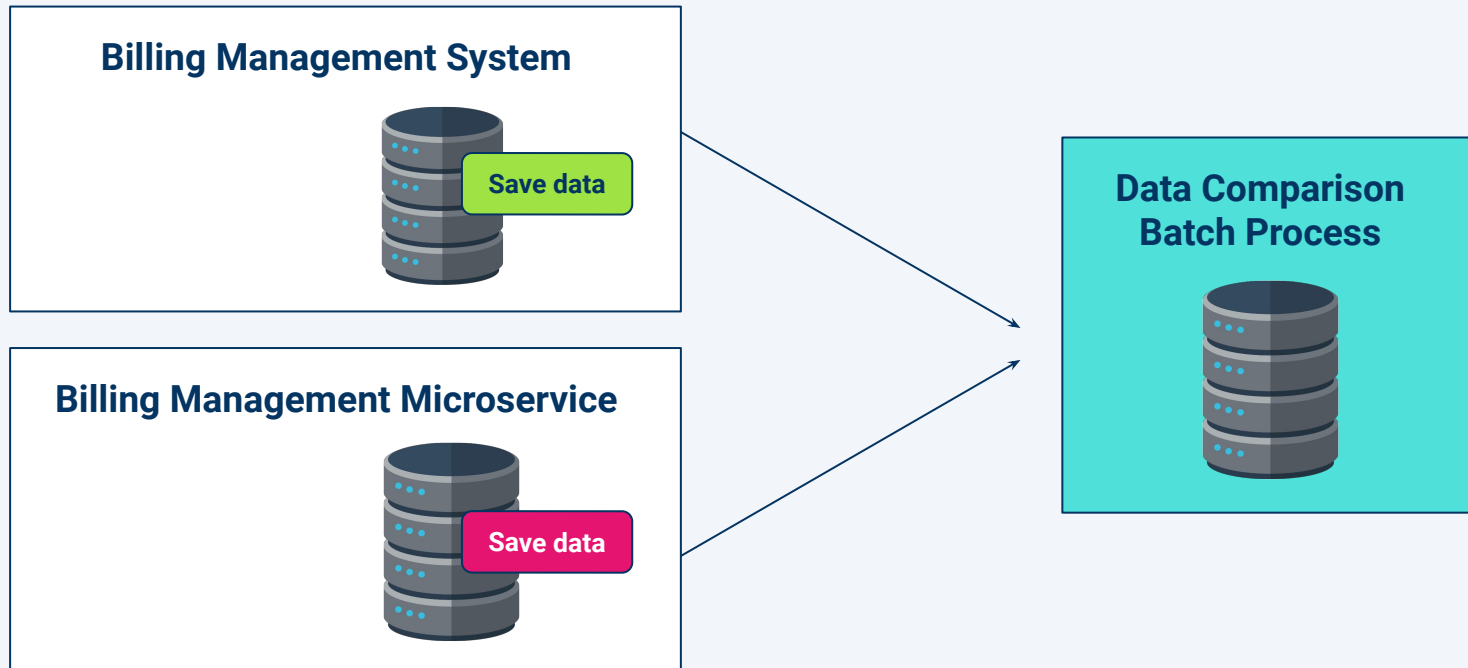
05

Remove the old implementation.



Parallel Run

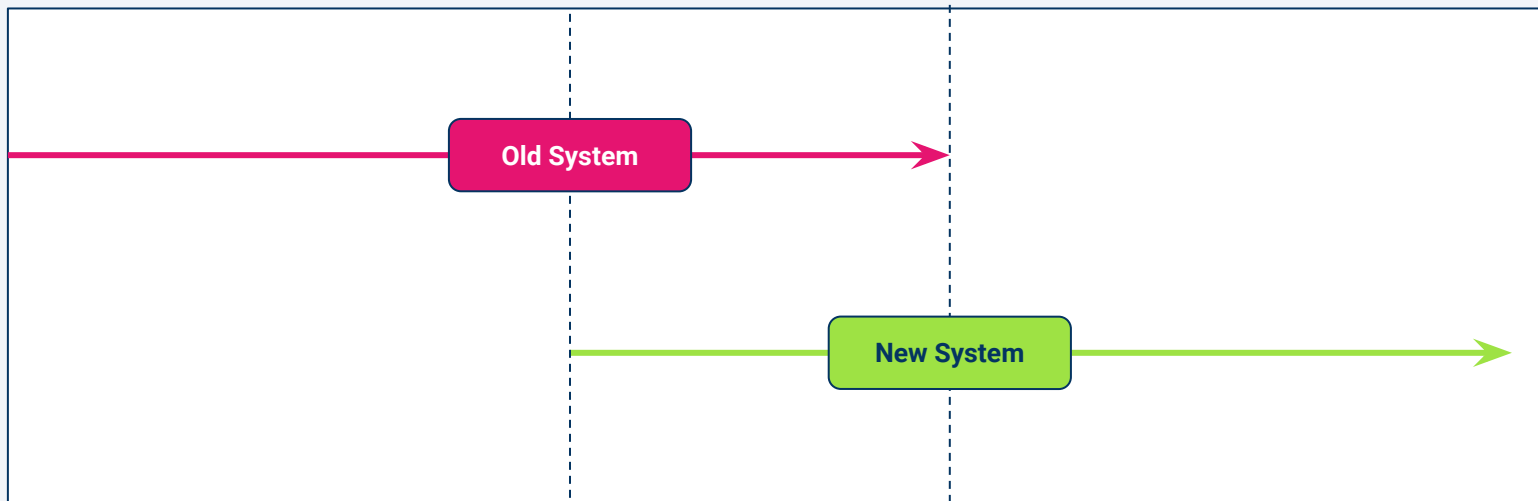
With the **parallel run** pattern, there is no feature toggle.
We are actually running BOTH implementations at the same time.



Parallel Run

Only one of the implementations is considered the source of truth.

Usually the old implementation is the source of truth, and we compare the results of the new implementation against the old implementation to verify that the new implementation is working correctly.



UI Composition



With the **UI composition** pattern, the user interface of an application is made by assembling many small parts. This is different from the other patterns we are learning because we are changing the UI and the back end that supports that UI.



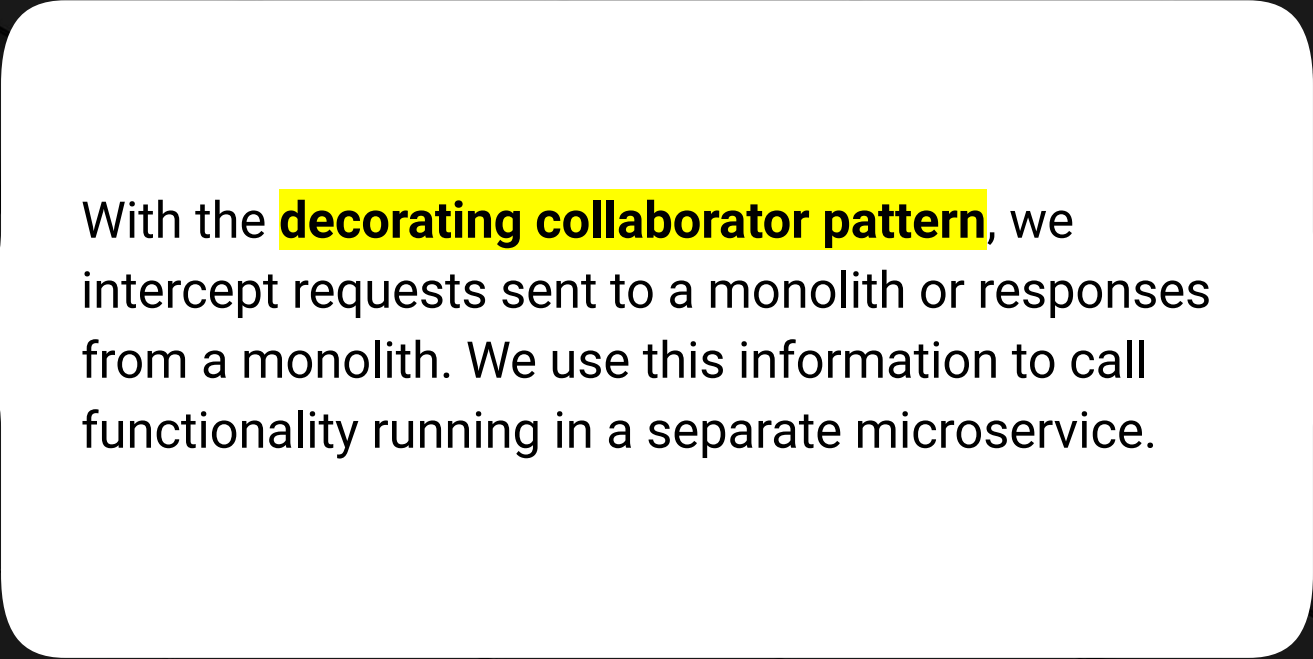
Splitting the UI into isolated components allows us to change or swap out specific components without breaking the entire UI. This pattern is similar to the branch by abstraction and parallel run patterns, but we're using components in the UI as the boundary for making changes.



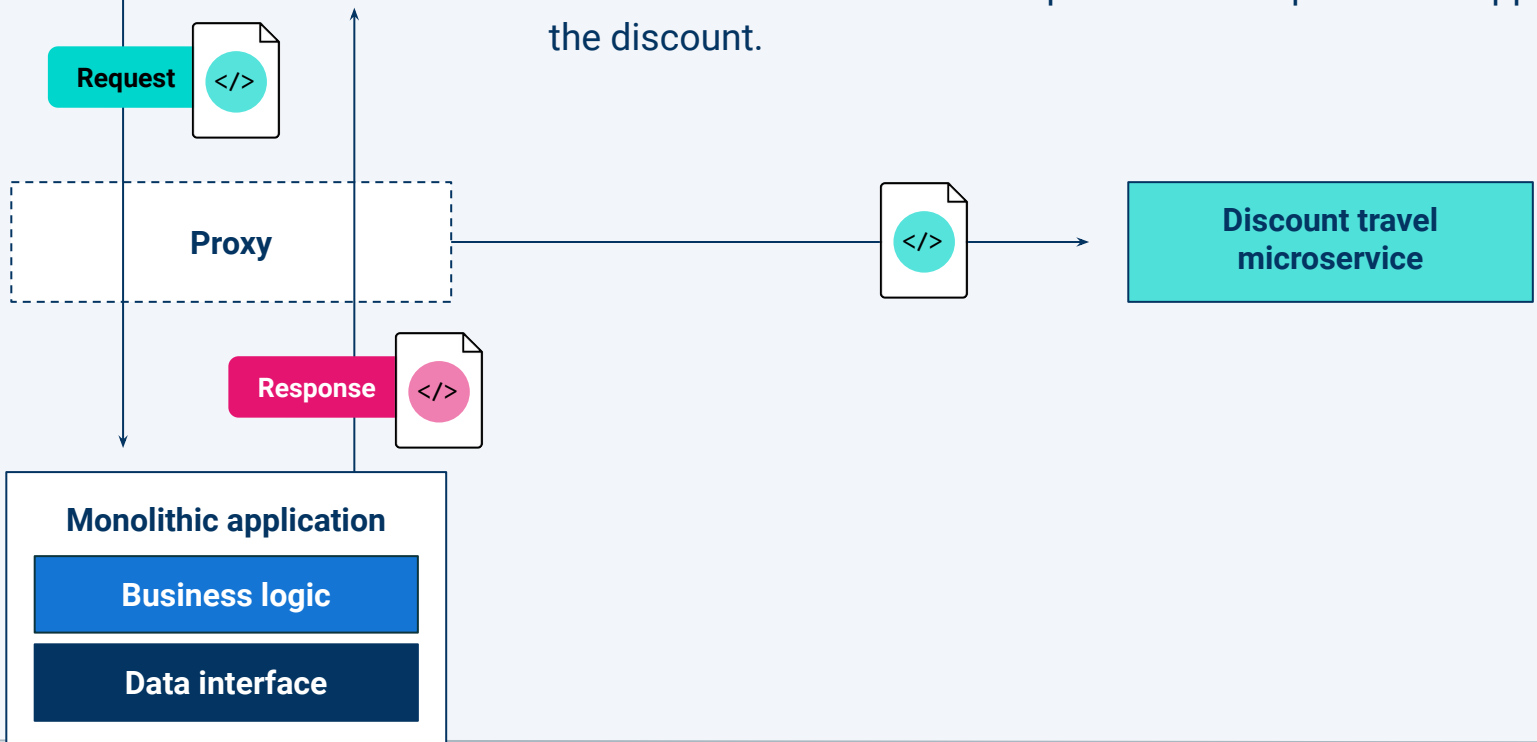
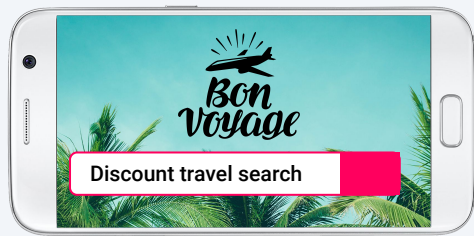
If we make a UI component that handles displaying that information, we can separate that component from the rest of the UI and reuse it where needed. This also allows us to change that component without affecting the rest of the UI.



**How can we migrate to
microservices if we can't change
the monolith code directly?**



With the **decorating collaborator pattern**, we intercept requests sent to a monolith or responses from a monolith. We use this information to call functionality running in a separate microservice.



Imagine we have an application that lets users place orders. We create a new **Discount microservice** to handle giving our users discounts. With the Decorating Collaborator pattern, the microservice could intercept the order requests and apply the discount.

Decorating Collaborator Pattern

Key points:



The client continues to send order requests and get responses.



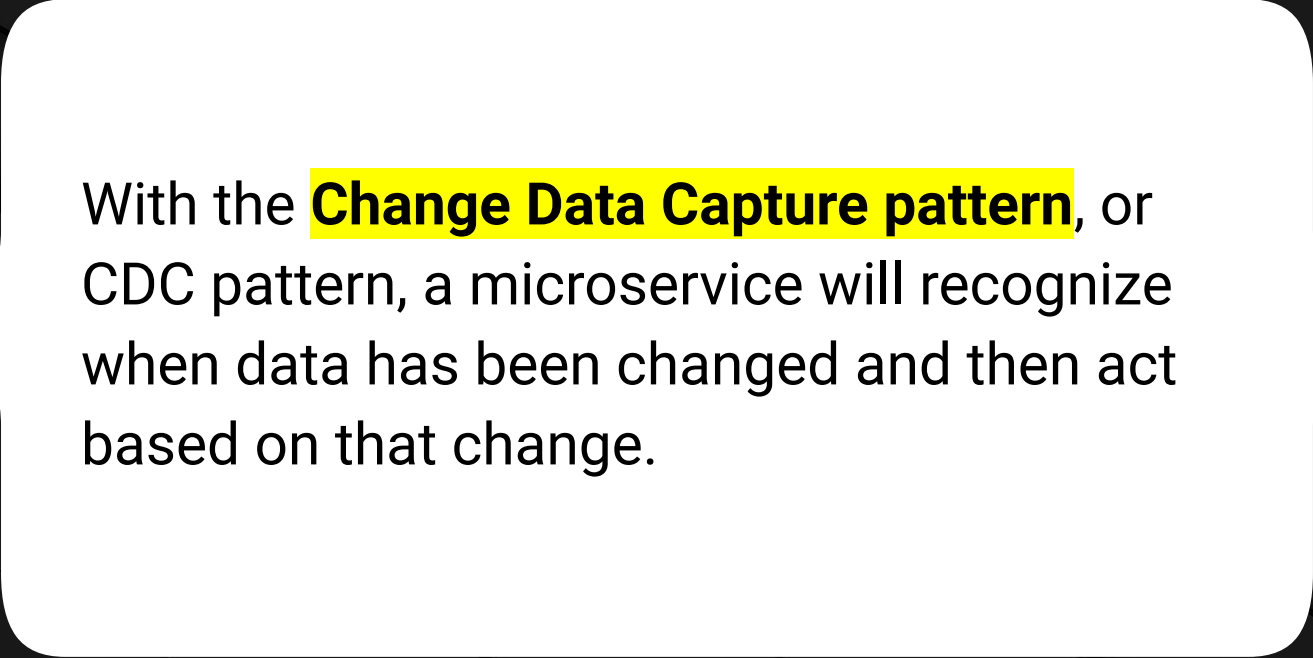
The monolith application continues to receive requests and send responses.



The Discount microservice handles all of the logic and work of applying discounts.



The client and monolith know nothing about the Discount microservice.

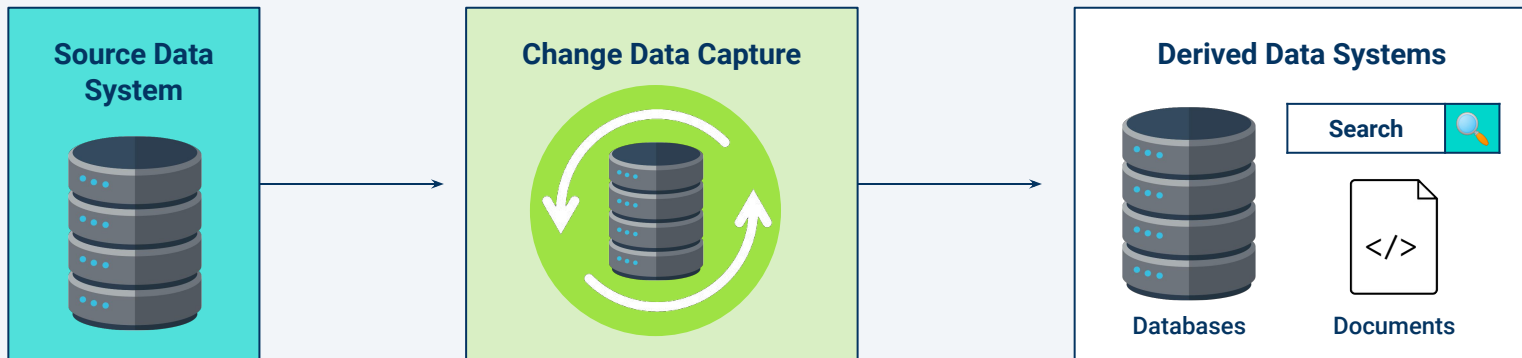


With the **Change Data Capture pattern**, or CDC pattern, a microservice will recognize when data has been changed and then act based on that change.

Change Data Capture Pattern

Similar to the decorating collaborator, instead of intercepting requests, in the **change data capture** pattern the microservice is examining the data store and detecting changes to the data.

We are not changing code or data in the monolith. We are monitoring the database for the monolith and detecting changes to the data. Based on those changes, the microservice will perform its functionality.





What are some challenges
with this pattern?



Detecting that a change happened is not difficult; we have tools and techniques that can help. The difficulty is understanding what a data change means.

Change Data Capture Pattern

For example, if the quantity of a product in the database decreases, what does it mean?



Did we sell the product?



Has a product been damaged or destroyed?



Was the quantity entered incorrectly, and we are updating it?