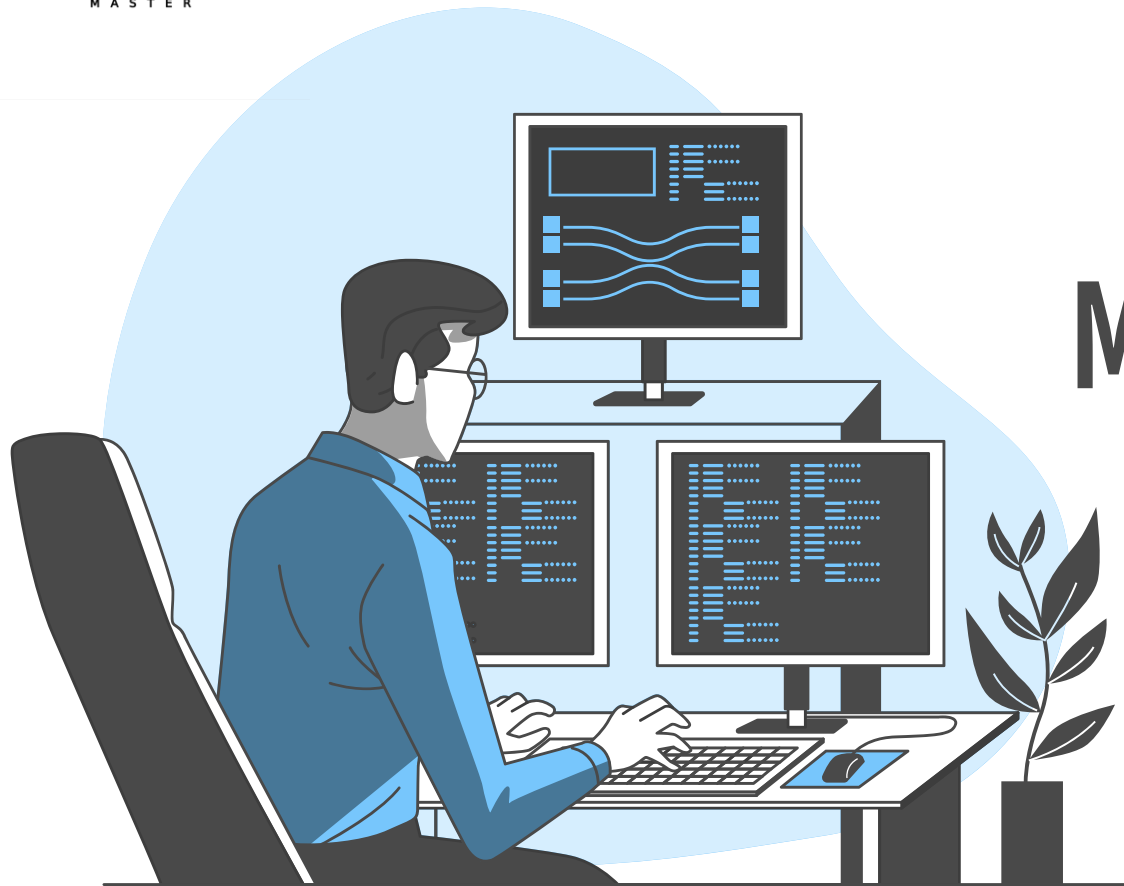


# Monolith to Microservices Examples

Juan Carlos Blázquez Muñoz  
Miguel Ángel Huerta Rodríguez



# Patrones de descomposición

01

**Strangler Fig**

02

**Branch By Abstraction**

03

**Decorating Collaborator**

04

**Parallel Run**

05

**Change Data Capture**



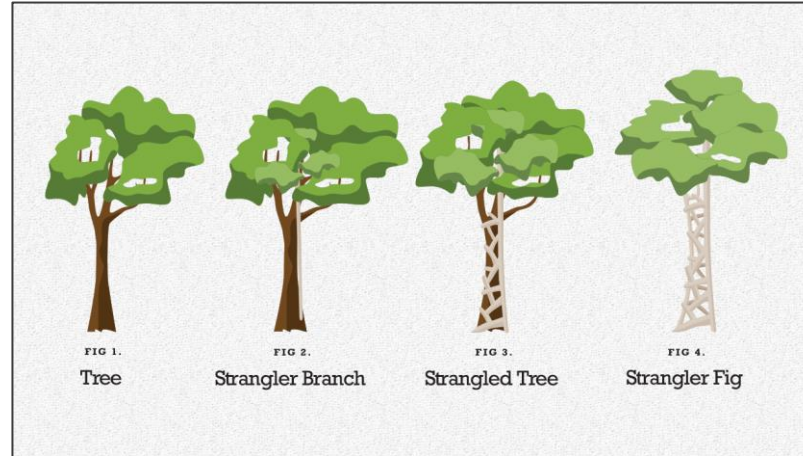


01

Strangler Fig

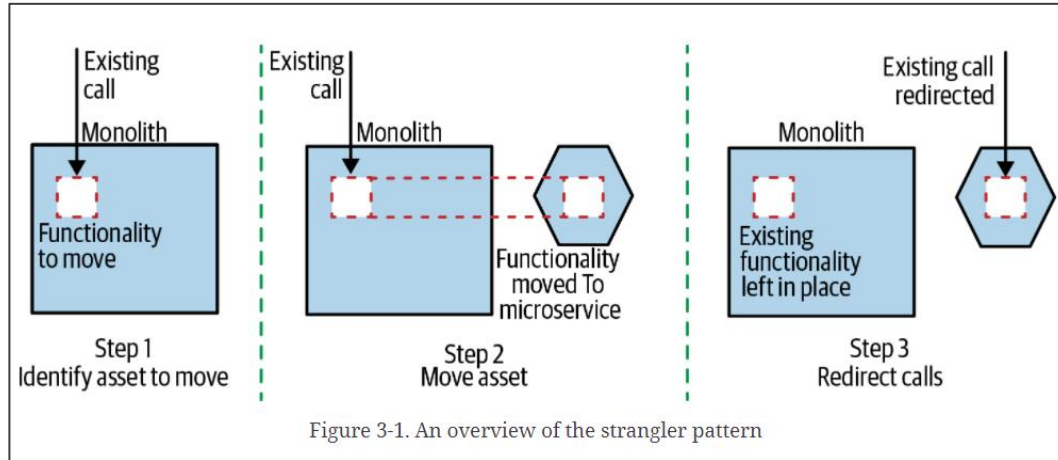


# Strangler Fig



- Mencionado por primera vez por *Martin Fowler*, inspirado en una planta que se apoya sobre un árbol.
- La idea es que lo viejo y lo nuevo puedan coexistir, dando tiempo al nuevo sistema para crecer y potencialmente reemplazar por completo al antiguo.

# Strangler Fig



Se basa en 3 pasos:

1. Las peticiones y funcionalidades se responden dentro del monolito.
2. Implementación de la funcionalidad en un nuevo microservicio.
3. Con su nueva implementación lista, migramos las peticiones del monolito al microservicio.



Da mucha importancia a que cada paso sea fácilmente reversible.

# Strangler Fig – Ejemplos



Extracción de funcionalidad independiente



Extracción de funcionalidad interna

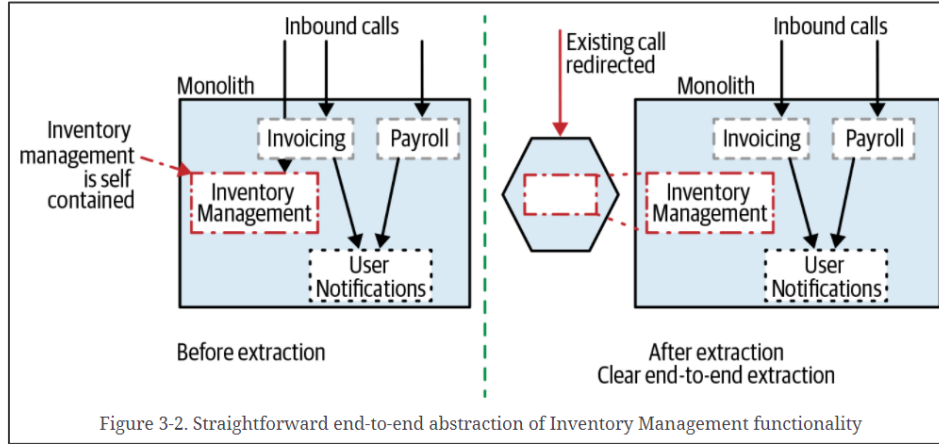


Interceptación de mensajes

1. Podemos cambiar el código del monolito
2. No podemos cambiar el código del monolito
3. No podemos cambiar la fuente de datos



# Extracción de funcionalidad independiente

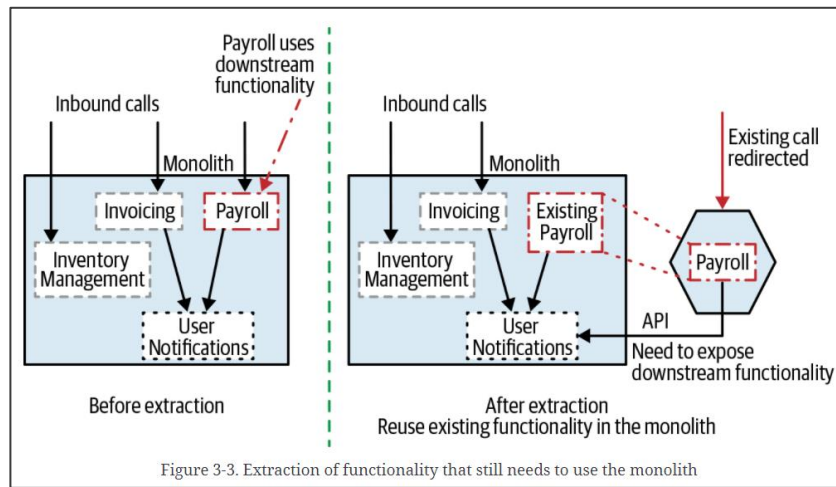


1. Nuestro monolito.
2. Creamos nuestro microservicio, copiamos el código de *Inventory Management*.
3. Redirigimos las peticiones de *Inventory Management* para que vayan a nuestro microservicio.



En caso de error, migramos las peticiones a la configuración inicial.

# Extracción de funcionalidad interna



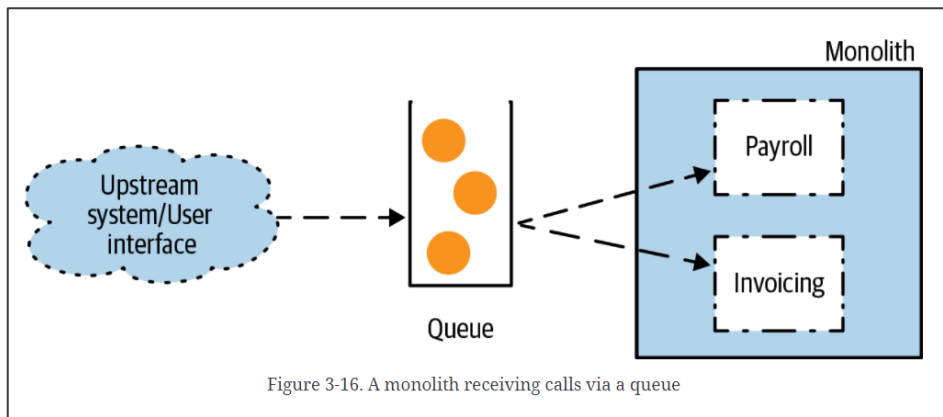
1. Nuestro monolito.
2. Creamos una v2 del monolito en el que expongamos *User Notifications*. Desarrollamos nuestro microservicio, basándonos en el código de *Payroll* y realizando una petición al monolito cuando debamos enviar una notificación al usuario.
3. Redirigimos las peticiones de *Payroll* al microservicio y el resto de las notificaciones a la versión 2 del monolito.



En caso de error, migramos las peticiones a la configuración inicial.

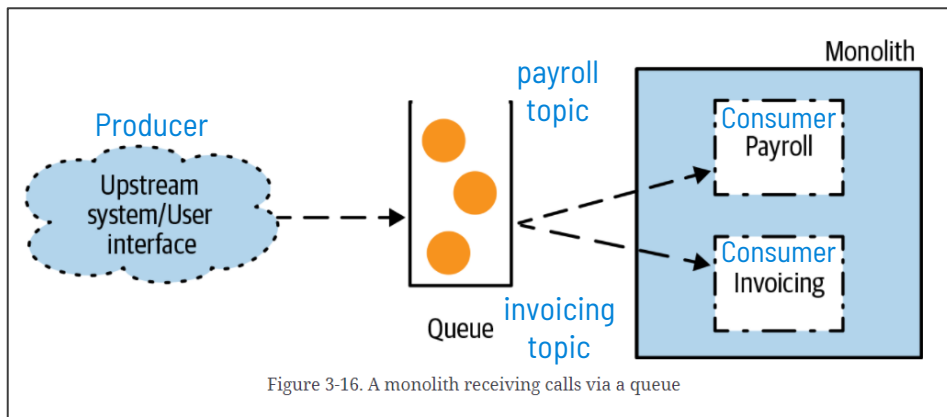


# Interceptación de mensajes



- Queremos interceptar y redirigir mensajes de la cola de mensajería. En ejemplos previos redirigíamos peticiones HTTP a través de un proxy.
- Como cola de mensajería elegimos utilizar *Kafka*.

# Interceptación de mensajes



- Queremos interceptar y redirigir mensajes de la cola de mensajería. En ejemplos previos redirigíamos peticiones HTTP a través de un proxy.
- Como cola de mensajería elegimos utilizar *Kafka*.
- 3 ejemplos:
  - Podemos cambiar el código del monolito
  - No podemos cambiar el código del monolito
  - No podemos cambiar la fuente de datos

# Interceptación de mensajes

## 1. Podemos cambiar el código del monolito

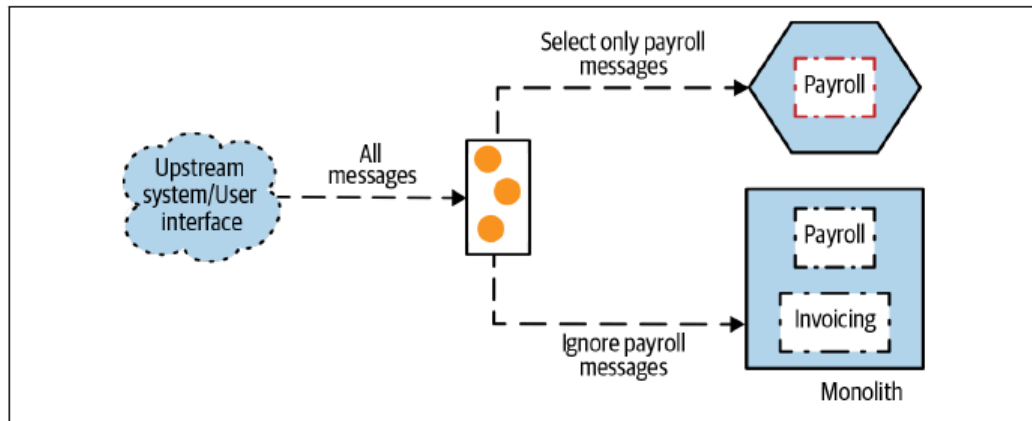


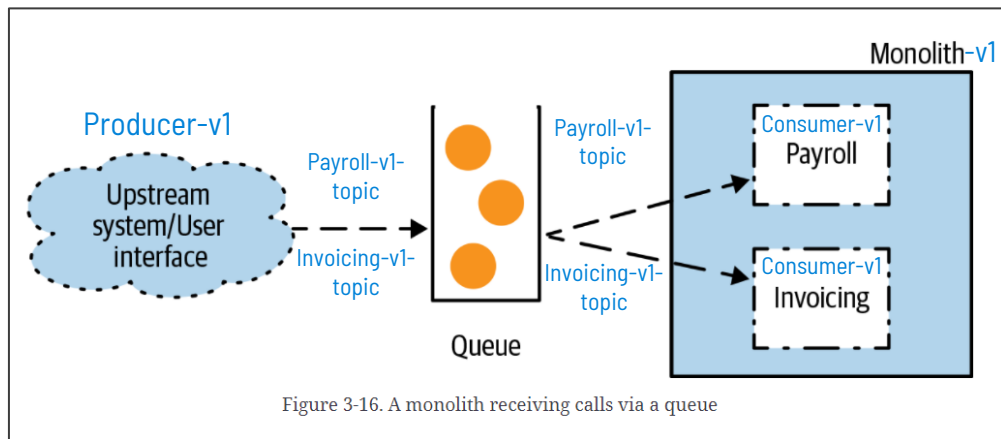
Figure 3-18. Using a content-based router to intercept messaging calls

Buscamos conseguir esta estructura, llegando sólo los mensajes de *Payroll* al microservicio mientras que en el monolito, se ignoran.

💀 *Kafka* no permite filtrar los mensajes para los consumidores. Si un consumidor está suscrito a un *topic*, leerá todos los mensajes.

# Interceptación de mensajes

1. Podemos cambiar el código del monolito



1. El *Producer* escribe mensajes en *invoicing-v1-topic* y *payroll-v1-topic* y el monolito consume esos mensajes.

# Interceptación de mensajes

## 1. Podemos cambiar el código del monolito

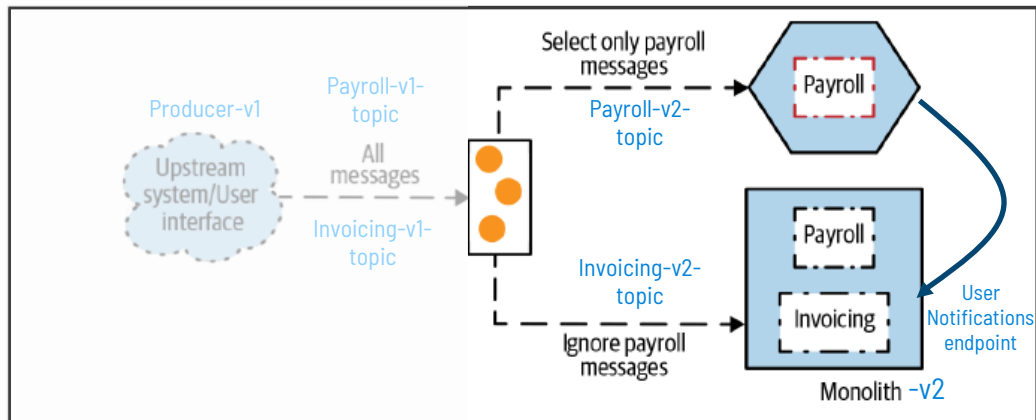


Figure 3-18. Using a content-based router to intercept messaging calls

2. Creamos una v2 del monolito en el que expongamos *User Notifications*. Desarrollamos nuestro microservicio, basándonos en el código de *Payroll* y realizando una petición al monolito cuando debemos enviar una notificación al usuario. El monolito v2 consume datos escritos en *invoicing-v2-topic* y el microservicio consume datos de *payroll-v2-topic*.

# Interceptación de mensajes

## 1. Podemos cambiar el código del monolito

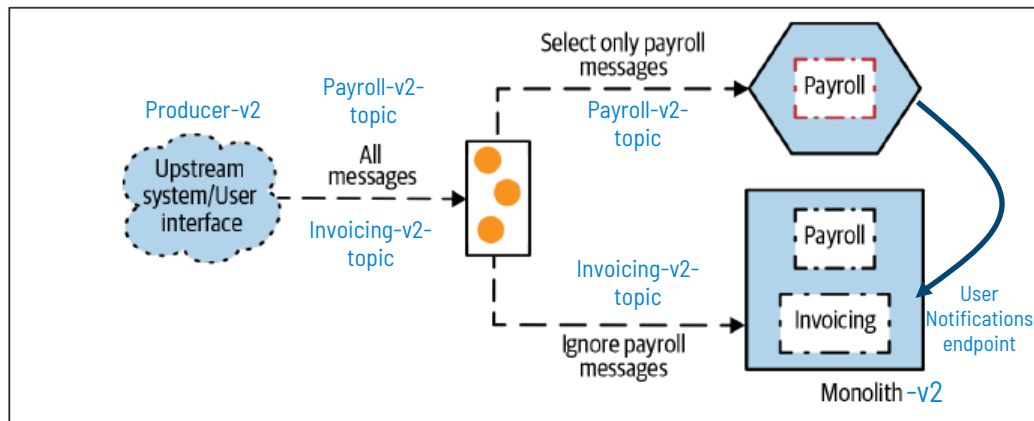


Figure 3-18. Using a content-based router to intercept messaging calls

3. En este punto, la redirección de mensajes la realizamos lanzando una nueva versión de nuestro microservicio generador de datos *Producer*, el cuál ahora escribe mensajes en *invoicing-v2-topic* y *payroll-v2-topic*.

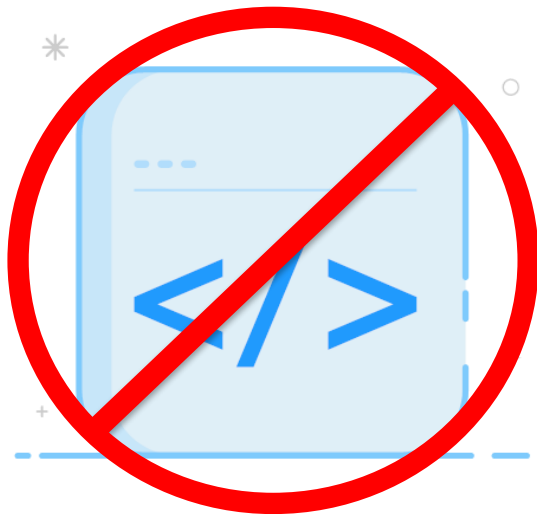


En caso de error, podemos lanzar la versión inicial del *Producer*, escribiendo en *invoicing-v1-topic* y *payroll-v1-topic*.

# Interceptación de mensajes

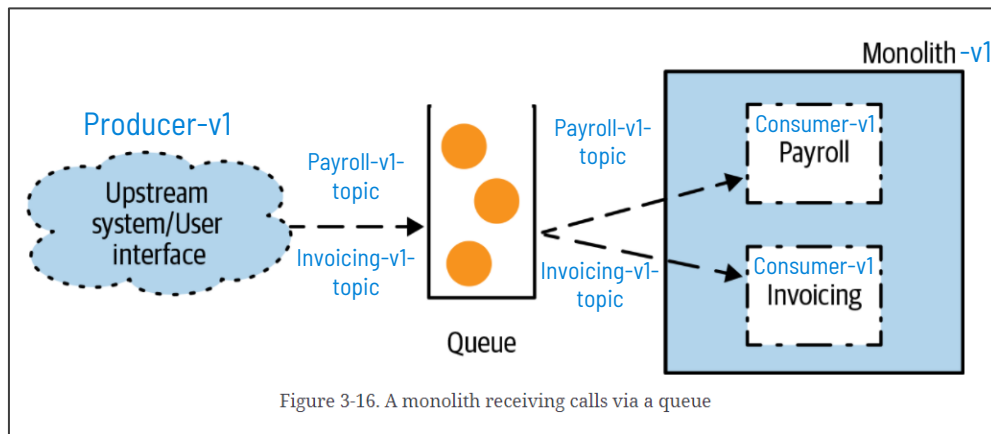
## 2. No podemos cambiar el código del monolito

- En este ejemplo, hemos partido de la premisa de que no podemos cambiar ni el código ni la configuración del monolito.
- No podemos cambiar los *topics* a los que se suscribe ni lanzar una versión 2 como hemos hecho en el anterior ejemplo.



# Interceptación de mensajes

## 2. No podemos cambiar el código del monolito

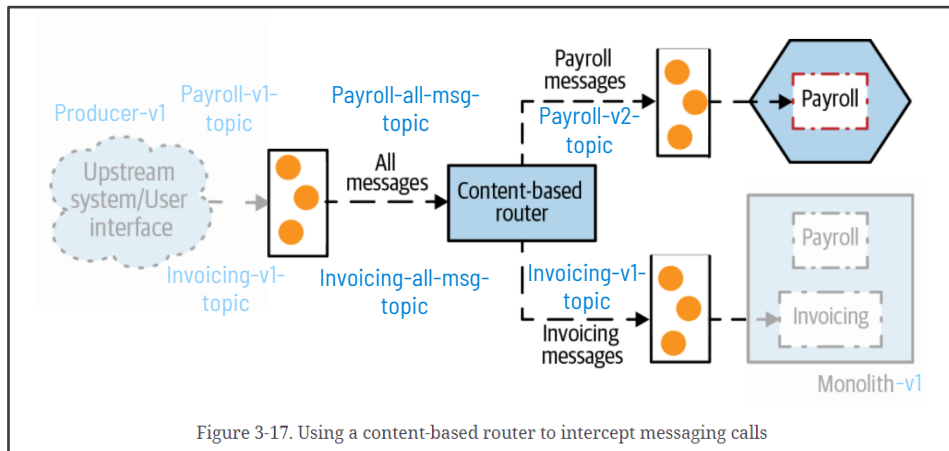


1. El *Producer* escribe mensajes en *invoicing-v1-topic* y *payroll-v1-topic*.

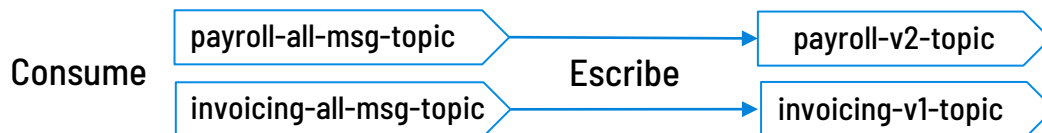


# Interceptación de mensajes

## 2. No podemos cambiar el código del monolito

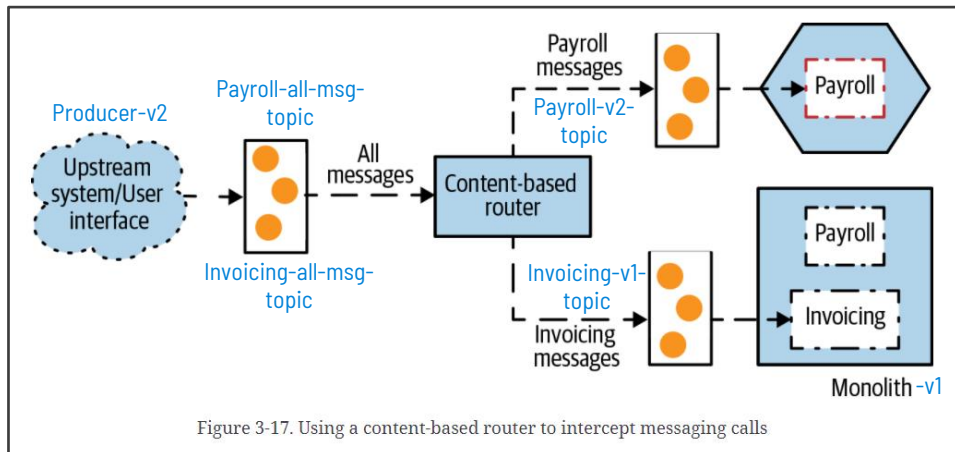


2. No podemos sacar una versión 2 del monolito. Al crear nuestro microservicio *Payroll* implementamos la funcionalidad de notificaciones al usuario *User Notifications*. Creamos el microservicio enrutador cbr – *Content Based Router*:



# Intercepción de mensajes

## 2. No podemos cambiar el código del monolito



- Realizamos la redirección de mensajes lanzando una nueva versión de nuestro microservicio generador de datos *Producer*, el cuál ahora escribe mensajes en *invoicing-all-msg-topic* y *payroll-all-msg-topic*.



En caso de error, podemos lanzar la versión inicial del *Producer*, escribiendo en *invoicing-v1-topic* y *payroll-v1-topic*.

# Interceptación de mensajes

## 3. No podemos cambiar la fuente de datos

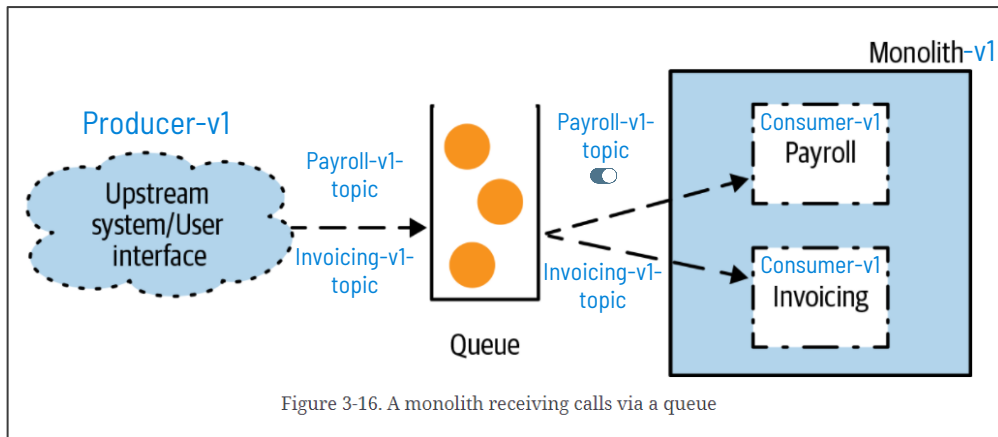

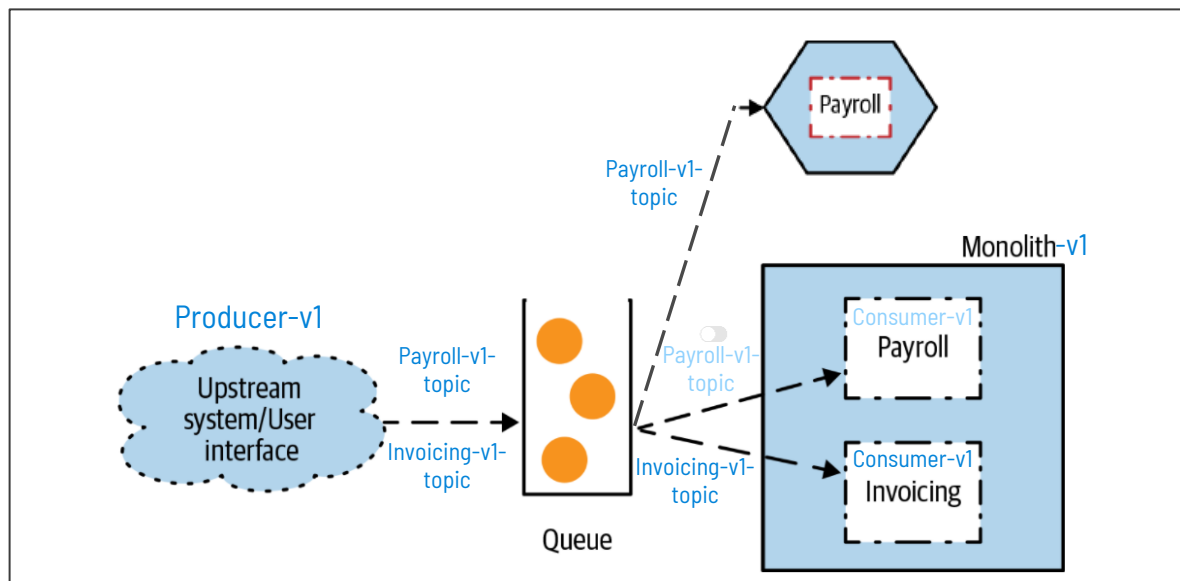


Figure 3-16. A monolith receiving calls via a queue

- En este ejemplo, hemos partido de la premisa de que no podemos cambiar la fuente de datos, nuestro *Produder*.
-  Podría hacerse de forma similar al anterior apartado, teniendo cuidado para no duplicar información. Al levantar el CBR, debemos detener la versión anterior, no pueden convivir
1. Partimos de una versión ampliada del monolito, en el que añadimos un *flag* gracias a *FF4J* que nos permite cambiar en caliente si deseamos consumir datos desde el *topic* o no.

# Interceptación de mensajes

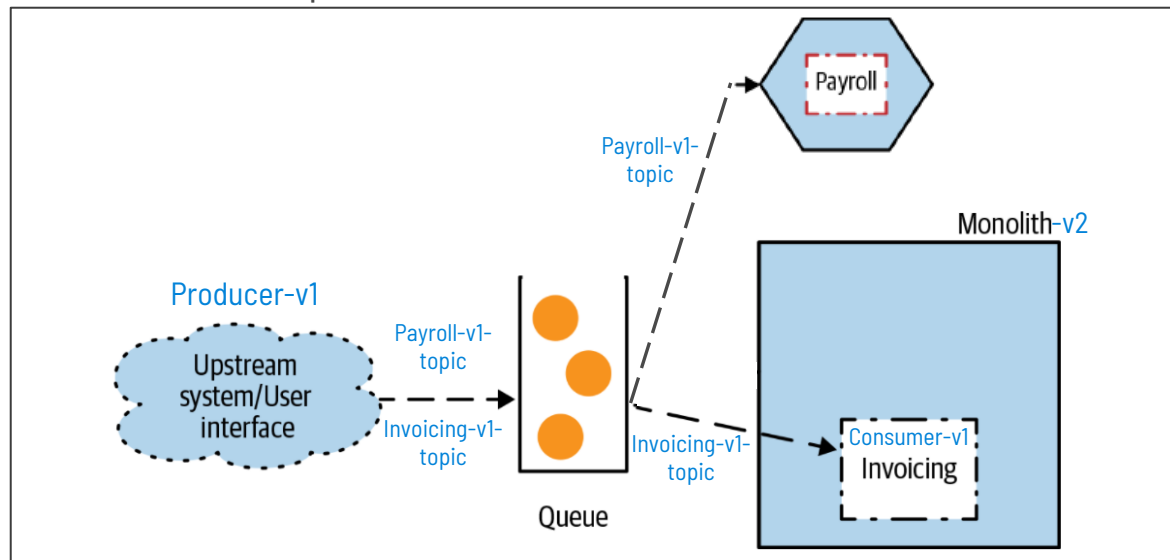
## 3. No podemos cambiar la fuente de datos



2. Desplegamos de nuestro microservicio de *Payroll* y desactivamos del consumo de datos de *payroll-v1-topic* del monolito.

# Interceptación de mensajes

## 3. No podemos cambiar la fuente de datos



3. Viendo que funciona correctamente, eliminamos del uso del *flag* y actualizamos el monolito eliminando el código no utilizado de *Payroll*.



En caso de error, podemos lanzar la versión inicial del *Producer*, escribiendo en *invoicing-v1-topic* y *payroll-v1-topic*.



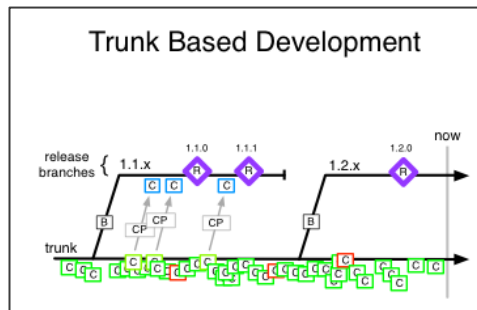
# 02

## Branch By Abstraction



# Branch By Abstraction

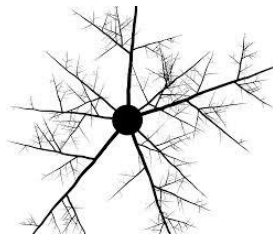
- Hemos tratado en el patrón *Strangler Fig* la extracción de la funcionalidad de *Inventory* y de *Payroll / Invoicing*. ¿Qué ocurre si deseamos extraer la funcionalidad de *User Notification*?
- Nos situamos en el caso de que necesitamos migrar un código interior del monolito el cuál recibe peticiones internas de otros servicios.
- Es habitual afrontar estos cambios en otra rama separada para evitar afectar a otros desarrolladores. Problemas al integrar.
- Este patrón permite que dos implementaciones del mismo código coexistan en la misma versión, sin afectar la funcionalidad.
- Muy utilizada en *Trunk-based development*.



# Branch By Abstraction

Se basa en 6 pasos:

1. Crear una abstracción para reemplazar la funcionalidad.
2. Cambiar los clientes de la funcionalidad existente para utilizar la nueva abstracción.
3. Crear una nueva implementación de la abstracción que realice la petición a nuestro nuevo microservicio.
4. Cambiar la abstracción para usar nuestra nueva implementación.
5. Limpiar la abstracción y eliminar la implementación anterior.
6. (Opcional): Borrar la interfaz.





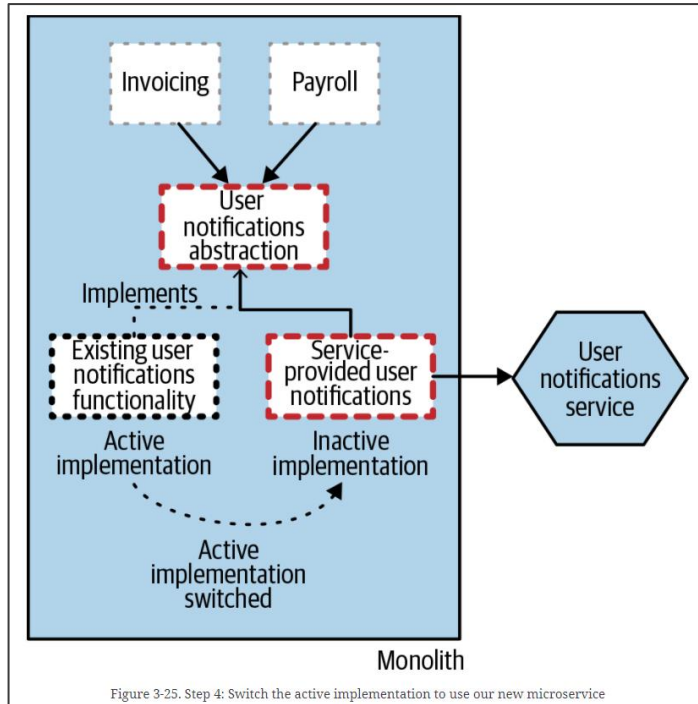
# Branch By Abstraction – Ejemplos



Extracción de funcionalidad dependiente



# Extracción de funcionalidad dependiente



1. Creamos la interfaz *UserNotificationService*.
2. Adaptamos la implementación de *UserNotificationService* (que pasa a llamarse *UserNotificationServiceImpl*) existente para utilizar la interfaz.
3. Creamos una nueva implementación de la interfaz, *UserNotificationServiceMSImpl* que realizará una petición a nuestro nuevo microservicio (antes de pasar al siguiente paso, debe estar desarrollado).
4. Modificamos a través de *FF4J* el uso de una u otra implementación en tiempo de ejecución.
5. Eliminamos el *flag* y la implementación antigua.

# Extracción de funcionalidad dependiente

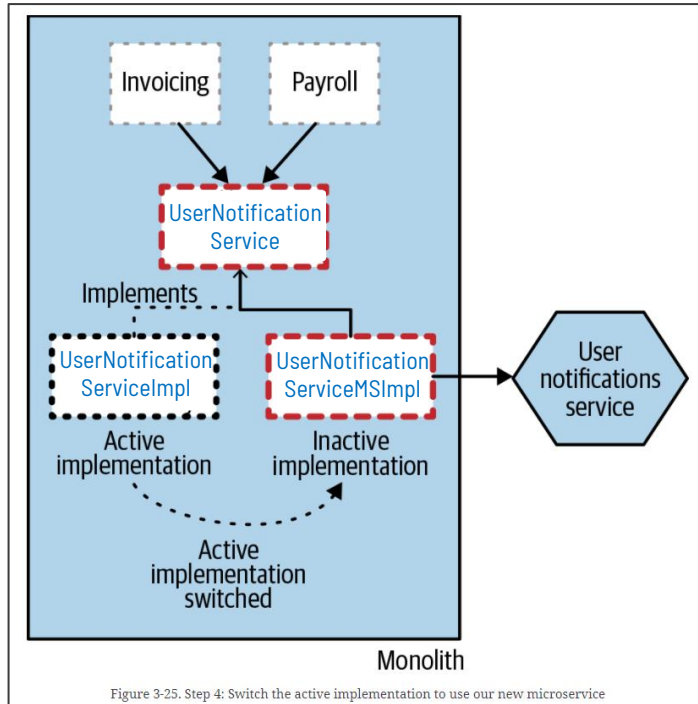
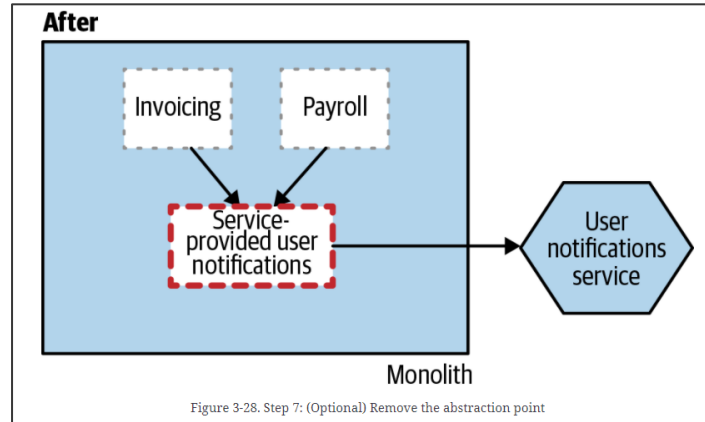


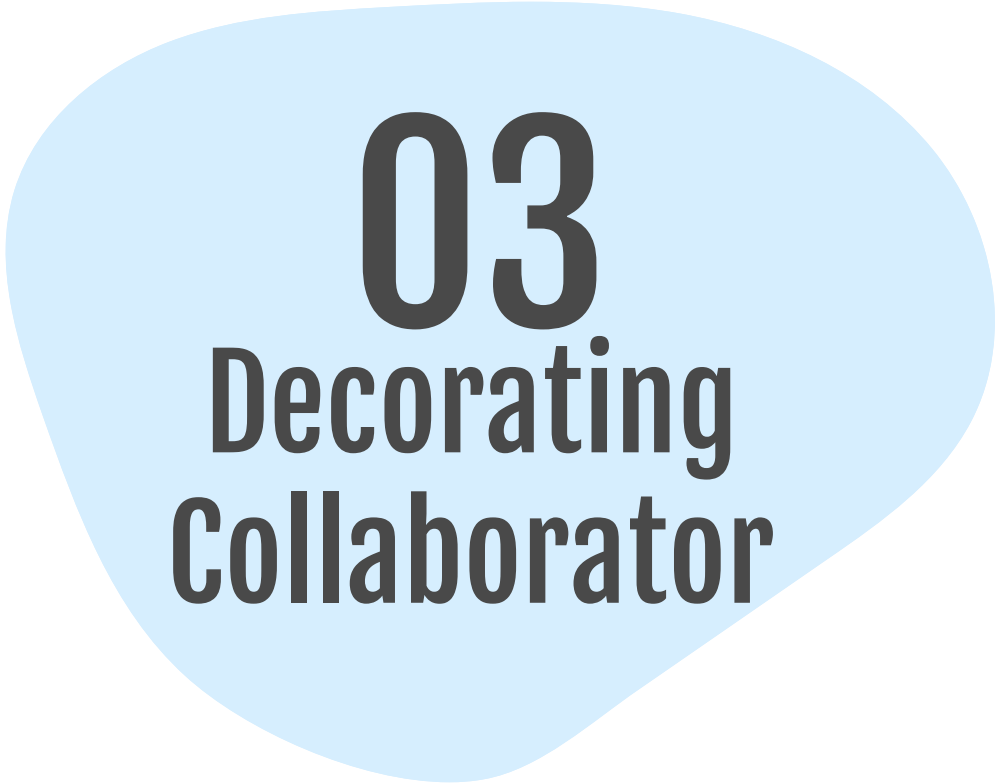

Figure 3-25. Step 4: Switch the active implementation to use our new microservice

1. Creamos la interfaz *UserNotificationService*.
2. Adaptamos la implementación de *UserNotificationService* (que pasa a llamarse *UserNotificationServiceImpl*) existente para utilizar la interfaz.
3. Creamos una nueva implementación de la interfaz, *UserNotificationServiceMSImpl* que realizará una petición a nuestro nuevo microservicio (antes de pasar al siguiente paso, debe estar desarrollado).
4. Modificamos a través de *FF4J* el uso de una u otra implementación en tiempo de ejecución.
5. Eliminamos el *flag* y la implementación antigua.

# Extracción de funcionalidad dependiente


- (Opcional): Borrar la interfaz. Podríamos también renombrar *UserNotificationServiceMSImpl* por *UserNotificationService*.





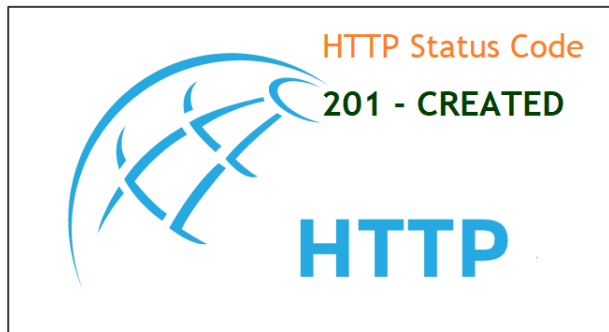
# 03

## Decorating Collaborator



# Decorating Collaborator

- Este patrón permite añadir una nueva funcionalidad a un sistema sin que la aplicación subyacente sepa nada al respecto.
- Es un patrón ideal para añadir funcionalidades cuando no podemos cambiar el código del sistema principal.
- En lugar de interceptar estas llamadas antes de que lleguen al monolito, permitimos que la llamada siga adelante con normalidad. Luego, cuando la petición es respondida, según su resultado, podemos llamar a nuestros microservicios externos a través de un proxy.



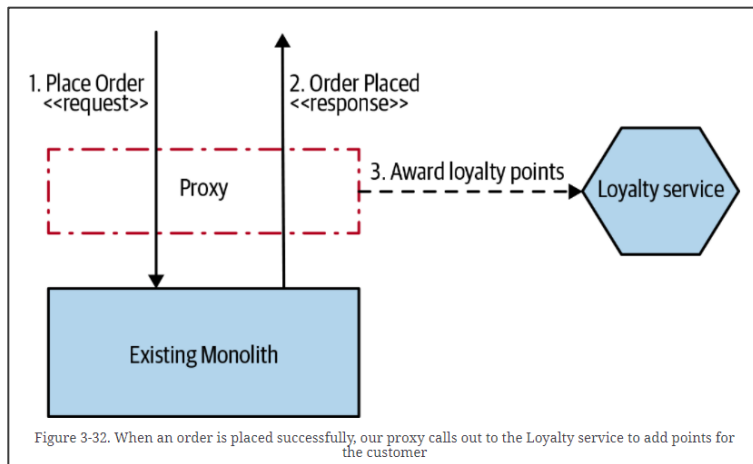
# Decorating Collaborator – Ejemplos



Nueva funcionalidad de fidelización



# Nueva funcionalidad de fidelización



1. Nuestro monolito. Tenemos un proxy de *NGINX* que dirige las peticiones al monolito.
  2. Desplegamos nuestro microservicio de *Loyalty* y nuestro *Spring Cloud Gateway*.
  3. Redirigimos las peticiones del proxy de *NGINX* a nuestro *Spring Cloud Gateway* el cuál se encargará de direccionarlas al monolito o al microservicio.
- En el caso de realizar una creación de una *Order*, si es correcta, se añaden puntos al usuario en nuestro servicio de fidelización *Loyalty*.



En caso de error, podemos migrar las peticiones con la configuración inicial del proxy.



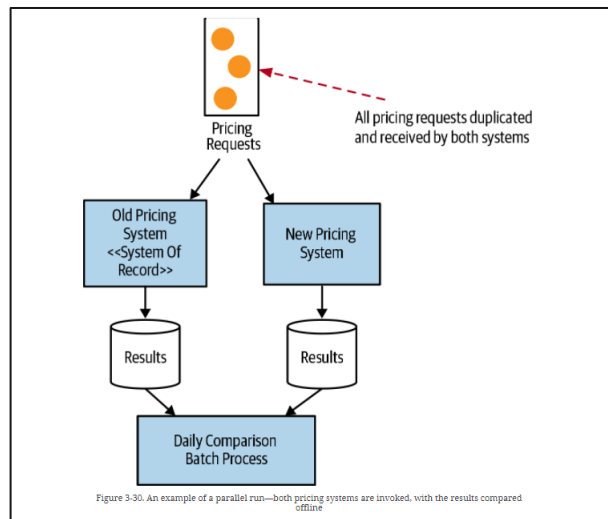


04

Parallel Run



# Parallel Run



- Se basa en llamar a la implementación antigua y a la nueva de forma simultánea.
- Nos permite comparar los resultados para asegurarnos de que sean equivalentes.
- Es un patrón invisible para el usuario.



En caso de error, migramos las peticiones a la configuración inicial.

# Parallel Run – Ejemplos



Usando Spies



Github Scientist

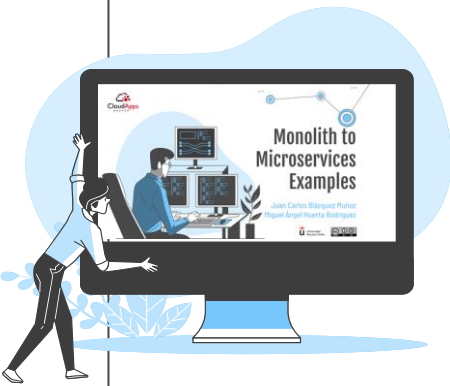


Diffy



Canary Releasing





# Usando Spies

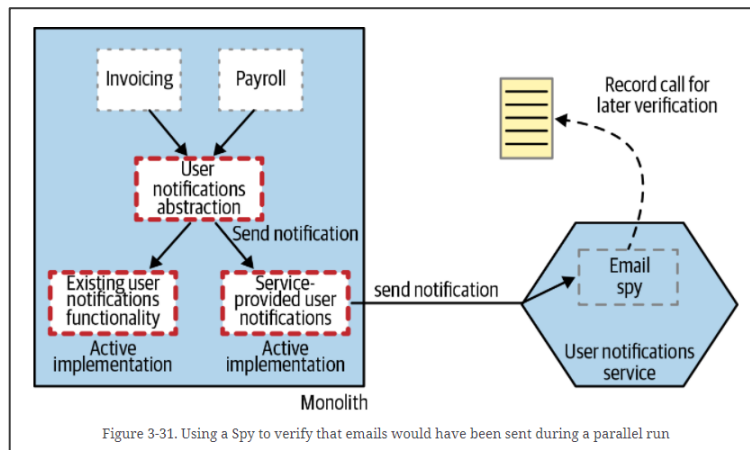


Figure 3-31. Using a Spy to verify that emails would have been sent during a parallel run

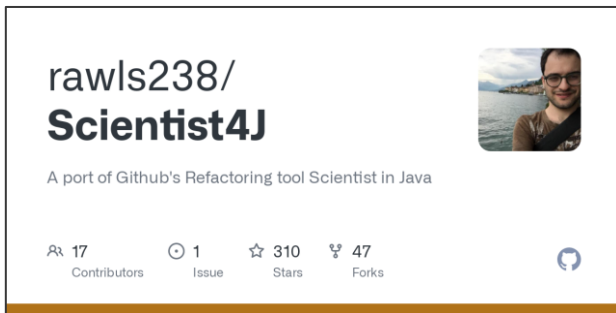
1. Nuestro monolito. Tenemos un proxy que dirige las peticiones al monolito.
2. Debemos desarrollar nuestro microservicio, con una implementación modificada, que no envíe realmente la notificación pero la registre como si la hubiera enviado (Spy). Ambas implementaciones van a convivir y no queremos que se dupliquen las notificaciones. Usaríamos en nuestro caso, un MS con un *batch* que compara los resultados registrados.
3. Una vez se compruebe que todas las peticiones son equitativas, antigua y nueva implementación, podríamos sacar una versión final con las peticiones migradas al MS.

# Github Scientist

- Existen librerías que permiten comparar resultados de forma muy sencilla. En nuestro caso, vamos a proceder a realizar un ejemplo con *Scientist4J*.
- Debemos hacer una pequeña modificación al código del monolito y del microservicio, puesto que nuestras operaciones son *void*.
- Vamos a retornar un valor booleano para poder comparar las llamadas de la antigua implementación y de la nueva.



El ejemplo anterior, la comparación se realizaba en *background* con nuestro *batch*, sin embargo, con esta librería lo ejecutaríamos en tiempo real. Por contra, esta solución añade latencias y puede no ser adecuada en ciertos contextos.



# Github Scientist

```
public Boolean scientistExperiment(Long id) {
    DropwizardMetricsProvider metricRegistry = new DropwizardMetricsProvider();
    Experiment<String> experiment = new Experiment<>("notify", metricRegistry);

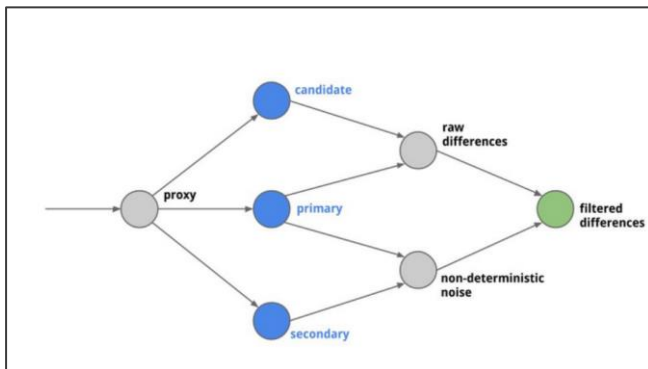
    Callable<String> oldCodePath = () -> userNotificationService.getNotify(id);
    Callable<String> newCodePath = () -> userNotificationServiceMS.getNotify(id);
    try {
        experiment.run(oldCodePath, newCodePath);
    } catch (Exception e) {
        System.out.println(e);
    }

    MetricName metricName = MetricName.build("scientist.notify.mismatch");
    Counter result = metricRegistry.getRegistry().getCounters().get(metricName);

    log.info("The ScientistExperiment result with compare oldCode/newCode is " + result.getCount() + " mismatch");
    return result.getCount() == 0;
}
```

# Diffy

- Diffy actúa como un proxy externo, en nuestro caso como un comparador externo.
- Sirve para encontrar posibles errores en nuestro servicio utilizando las instancias en ejecución de la nueva funcionalidad y de la antigua.
- Multiplica las peticiones que recibe a cada una de las instancias en ejecución. A continuación, compara las respuestas e informa de cualquier diferencia que pueda surgir en esas comparaciones.
- Si dos implementaciones del servicio devuelven respuestas "similares", para un conjunto suficientemente grande y diverso de peticiones, entonces las dos implementaciones son equivalentes.



# Diffy

**diffy**

## Diffy Testing Service

unknown

Last reset a few seconds ago

Exclude Noise ☐

fail-api	100.0%	FAILING
1 Requests		
success-api	0.0%	FAILING
1 Requests		

success-api

0 DIFFS 1 REQUESTS 0.00 % FAILING

No differences

### Primary Response

```
200
chunked: false
content
  type: json
  value
    id: 1
    shipTo: user 2
    total: 6549.95
headers
  content-length
    0: 42
  content-type
    0: application/json
  date
    0: Thu, 18 Nov 2021 00:49:35 GMT
```

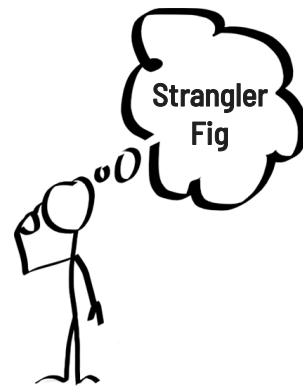
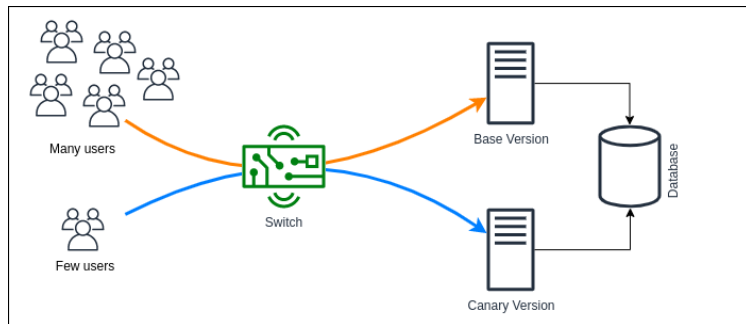
### Candidate Response

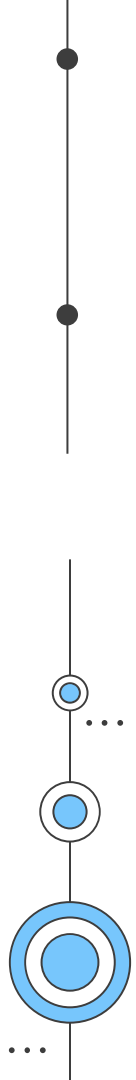

```
404
chunked: false
content
  type: json
  value
    error: Not Found
    path: /payroll/1
    status: 404
    timestamp: 2021-11-18T00:49:35.786+00:00
headers
  content-length
    0: 98
  content-type
    0: application/json
  date
    0: Thu, 18 Nov 2021 00:49:35 GMT
vary
  0: Access-Control-Request-Headers
  1: Access-Control-Request-Method
  2: Origin
```



# Canary Releasing

1. Nuestro monolito.
  2. Creamos una v2 del monolito en el que exponemos *User Notifications*. Desarrollamos nuestro microservicio, basándonos en el código de *Payroll* y realizando una petición al monolito cuando debamos enviar una notificación al usuario.
  3. Nuestro proxy actúa también como **balanceador de carga**. Dirigimos parte de las peticiones de *Payroll* al microservicio, vamos aumentando progresivamente.
- ⚠ En caso de error, migramos las peticiones a la configuración inicial.





# 05

## Change Data Capture

# Change Data Capture

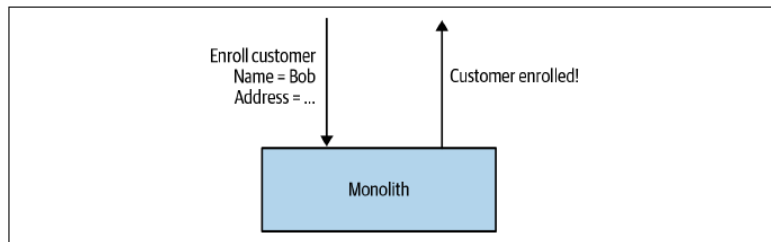


Figure 3-34. Our monolith doesn't share much information when a customer is enrolled

- Este patrón se basa en reaccionar a los cambios realizados en la base de datos.
- Se aplica cuando en nuestro monolito, la respuesta a las peticiones no tiene suficiente información provocando que no podamos aplicar el patrón *Decorating Collaborator* (usábamos el id del usuario para añadirle puntos de fidelización).
- En este caso, podríamos aplicar *Change Data Capture*, detectando la inserción en las tablas de la BBDD y realizando una llamada a nuestro MS.

# Change Data Capture – Ejemplos



Transaction log pollers -  
Debezium

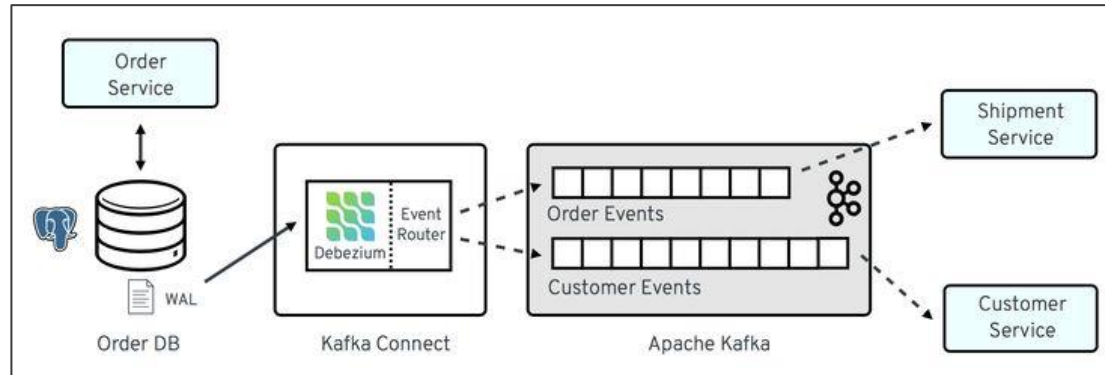


Batch delta copier



# Transaction log pollers - Debezium

- En las diferentes Bases de Datos existe un archivo en el que se registran las diferentes operaciones realizadas. El log de Transacciones.
- *Debezium* está formado por un conjunto de servicios distribuidos de código abierto que permiten detectar y transmitir, como flujos de eventos, los cambios que ocurren en una base de datos.



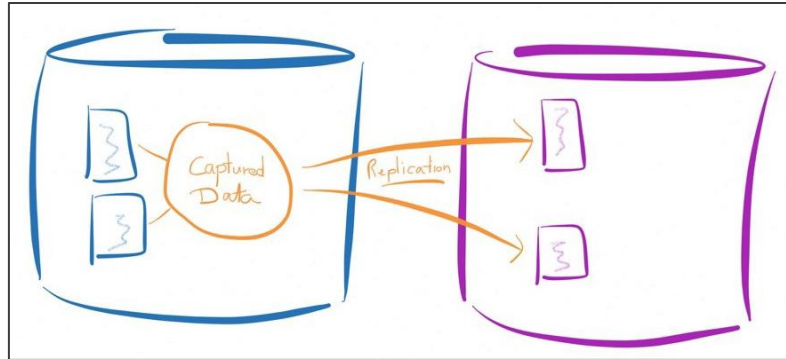
# Transaction log pollers – Debezium

- *Schema* contiene toda la información del cambio. La fila afectada antes y después del cambio, la operación de realizada, el conector o la fila modificada.
- El objeto *payload* contiene toda la información sobre los valores del cambio. Este objeto es el que más se suele explotar en los casos de uso que consumen esta información, ya que contiene toda la información de los valores de la fila anteriores al cambio, los posteriores al cambio, la operación que se ha realizado y la información complementaria tanto del conector como del propio cambio.
- Hemos utilizado *Kafdrop* como UI para ver los eventos que maneja *Debezium* dentro de *Kafka*.

```
{
  "schema": {...},
  "payload": {
    "before": { ①
      "id": 1004,
      "first_name": "Anne",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "after": { ②
      "id": 1004,
      "first_name": "Anne Marie",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "source": { ③
      "name": "1.7.1.Final",
      "name": "dbserver1",
      "server_id": 223344,
      "ts_sec": 1486501486,
      "gtid": null,
      "file": "mysql-bin.000003",
      "pos": 364,
      "row": 0,
      "snapshot": null,
      "thread": 3,
      "db": "inventory",
      "table": "customers"
    },
    "op": "u", ④
    "ts_ms": 1486501486308 ⑤
  }
}
```

# Batch delta copier

- Probablemente este sea el enfoque más sencillo. Consiste en escribir un programa que de forma periódica escanee la base de datos para ver qué datos han cambiado y se copien en el destino, un *batch*.
- El principal problema es averiguar qué datos han cambiado realmente. El diseño del esquema puede complicar mucho esta tarea.
- En nuestro caso, hemos agregado marcas de tiempo en nuestras entidades, de creación y modificación.
- Supone un esfuerzo bastante grande en comparación con el ejemplo planteado anteriormente, en el que utilizamos un sistema de captura de datos de cambios.



# Conclusiones

01

## Proceso complicado

Debe realizarse de forma incremental midiendo el impacto de los errores y su dificultad para resolverlos.

02

## Dependiente de la tecnología

Podemos encontrar limitaciones y características especiales que facilitan o complican la aplicación de los patrones.

03

## Importancia del contexto

Es importante adaptarse a la situación inicial para elegir el patrón adecuado. No existe un patrón ideal para todos los casos.

04

## Migración en caliente

Es uno de los puntos más complicados. Al final, estamos tratando una migración.





# ¡Gracias!

## ¿Preguntas?

jc-blazquez@hotmail.com  
miguel5692@gmail.com



**CREDITS:** This presentation template was created by [Slidesgo](#), including icons by [Flaticon](#), infographics & images by [Freepik](#) and illustrations by [Stories](#)

