



PONTIFICIA
UNIVERSIDAD
CATÓLICA
DE CHILE

IIC3103

Taller de Integración

Profesores

Arturo Tagle / Daniel Darritchon



Systems Design

**Principios de diseño de
sistemas escalables**

TEMARIO

1. Introducción y Fundamentos

1. Objetivos y Conceptos Básicos del Diseño de Sistemas
2. Importancia del Diseño de Sistemas en el Desarrollo de Software
3. Conceptos Clave en el diseño de sistemas

2. Domain-Driven Design (DDD)

1. Fundamentos, Modelado del Dominio, Entidades y Agregados

3. Arquitectura Hexagonal (Ports and Adapters)

1. Principios de la Arquitectura Hexagonal
2. DDD y arquitectura Hexagonal en la práctica.

4. Arquitectura de Microservicios

1. Introducción, Principios de diseño y beneficios de los Microservicios

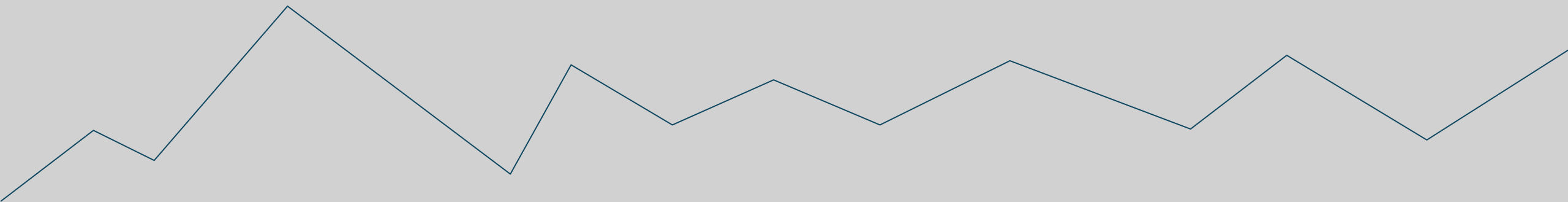
5. Principios SOLID

1. Principios y Aplicación en el Diseño de Sistemas

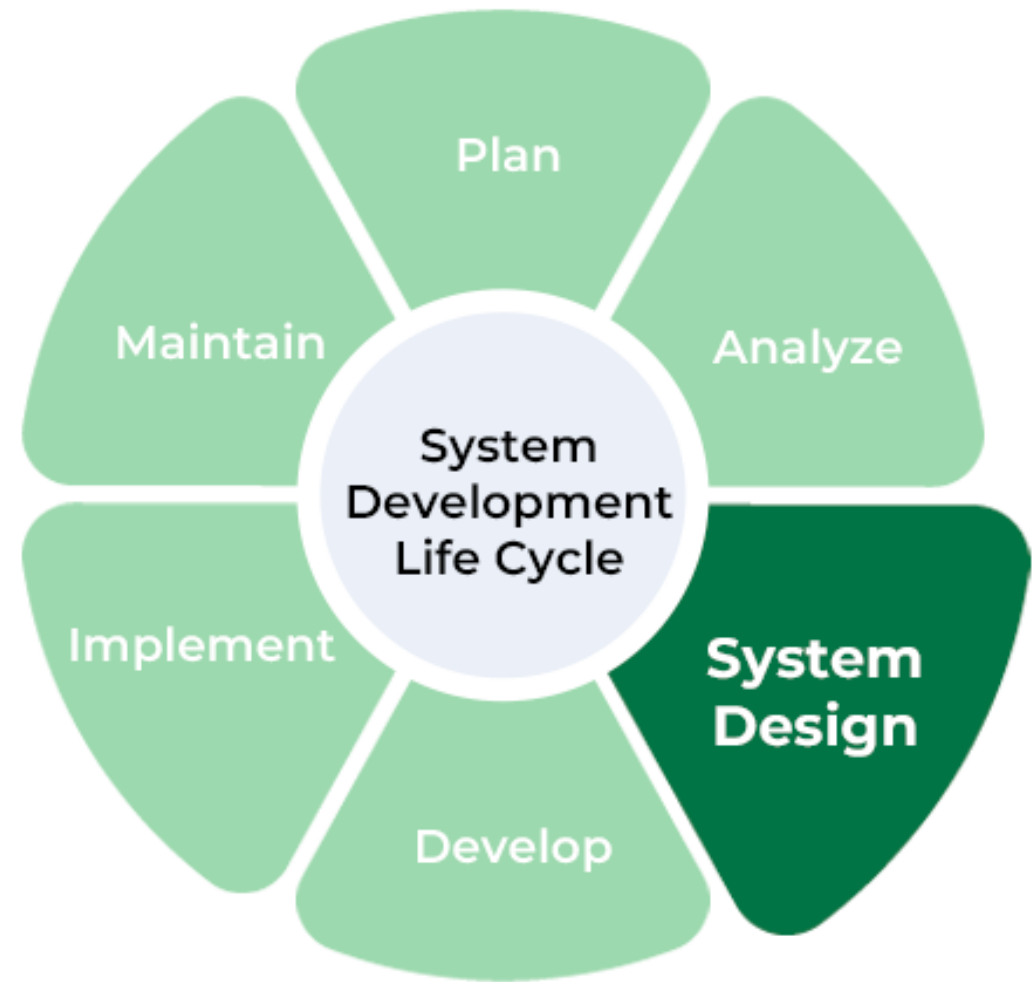


Introducción al Diseño de Sistemas

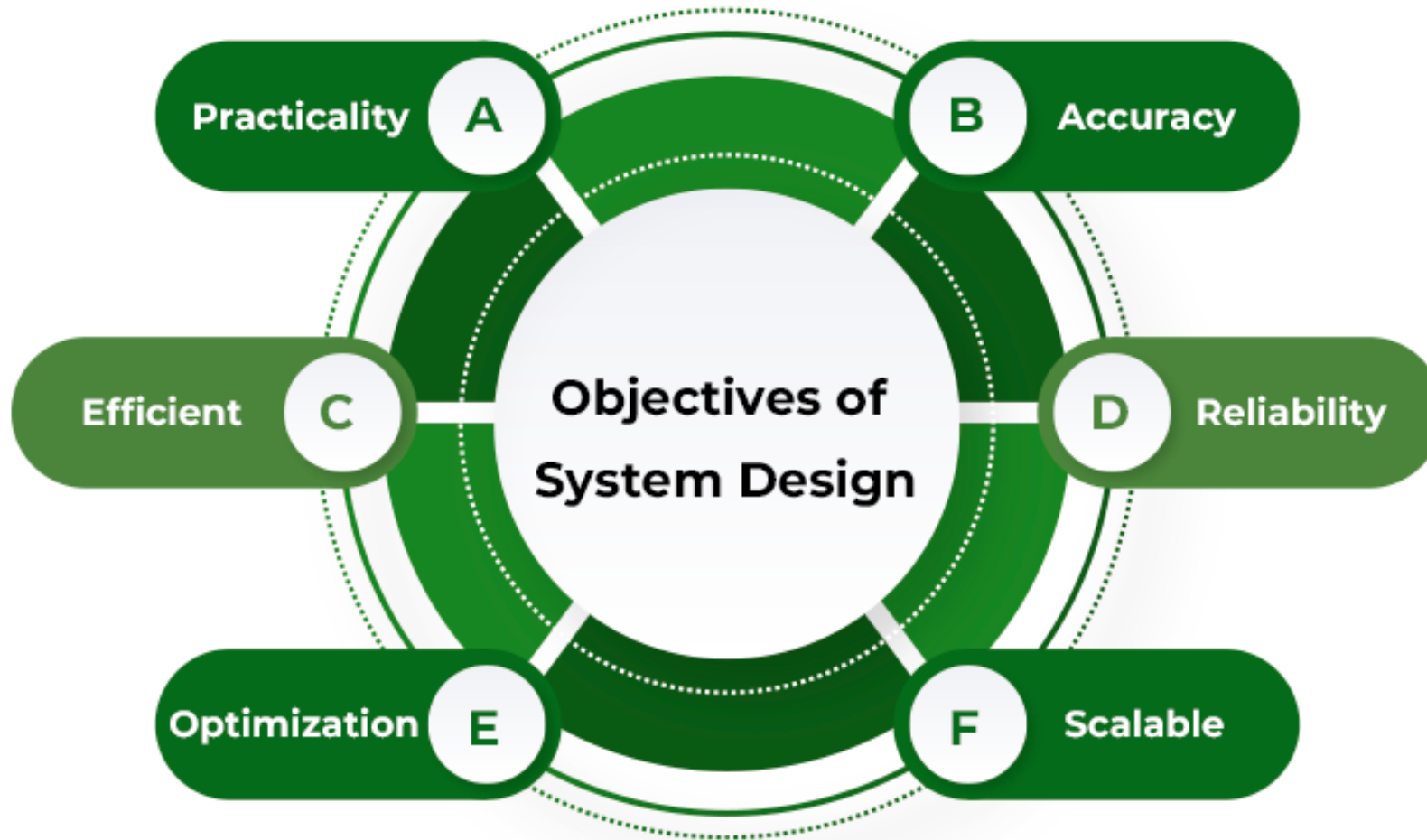
Conceptos Clave y la Relevancia del Diseño de Sistemas



Objetivos y Conceptos Básicos de Diseño de Sistemas



Objetivos



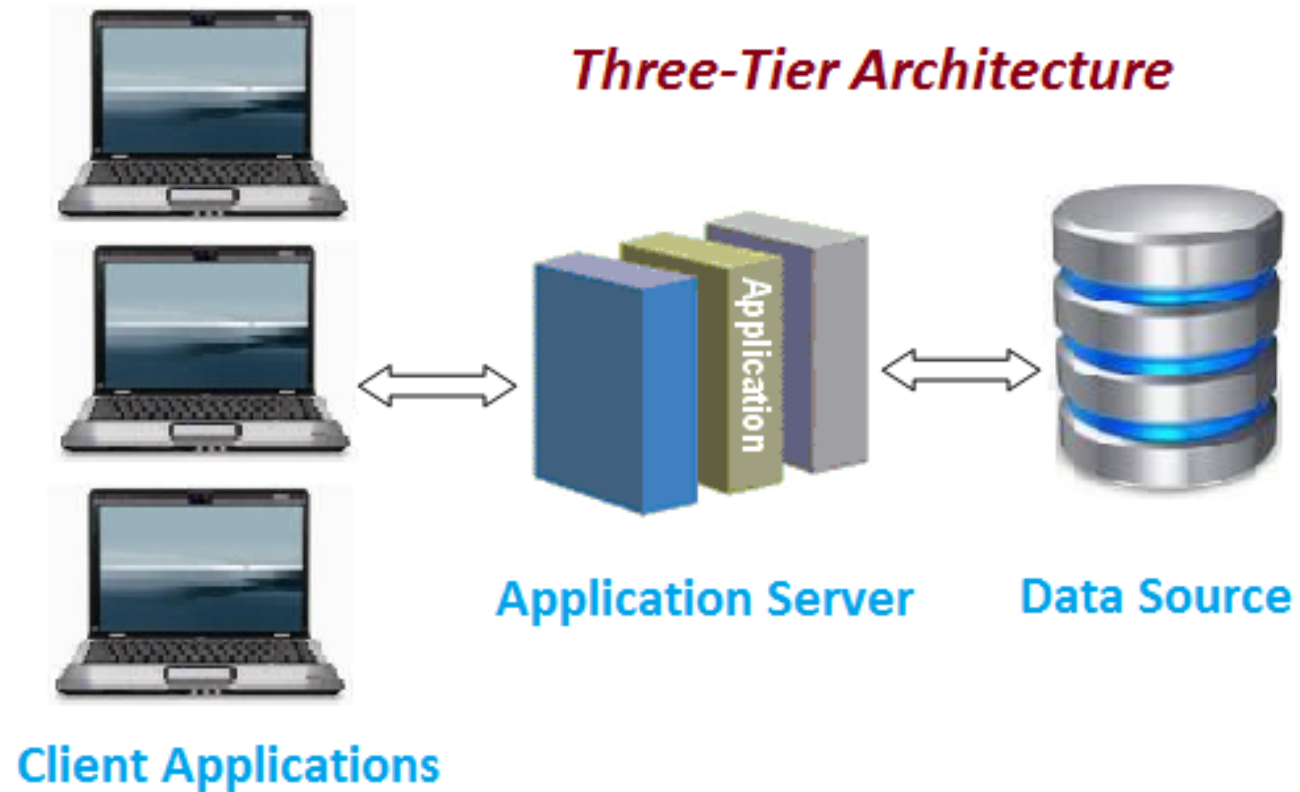
Objetivos

- **Practicidad:** Necesitamos un sistema que esté dirigido al conjunto de audiencias (usuarios) para los cuales se está diseñando.
- **Exactitud:** Debe realizarse de tal manera que cumpla con casi todos los requisitos en torno a los cuales se diseña, ya sean requisitos funcionales o no funcionales.
- **Eficiencia:** No sobreutilizar ni Subutilizar Recursos
- **Fiabilidad:** Debe estar disponible la mayor parte del tiempo posible.
- **Optimización:** El tiempo y el espacio son recursos escasos.
- **Escalabilidad (flexibilidad):** Debe ser adaptable con el tiempo según necesidades cambiantes.

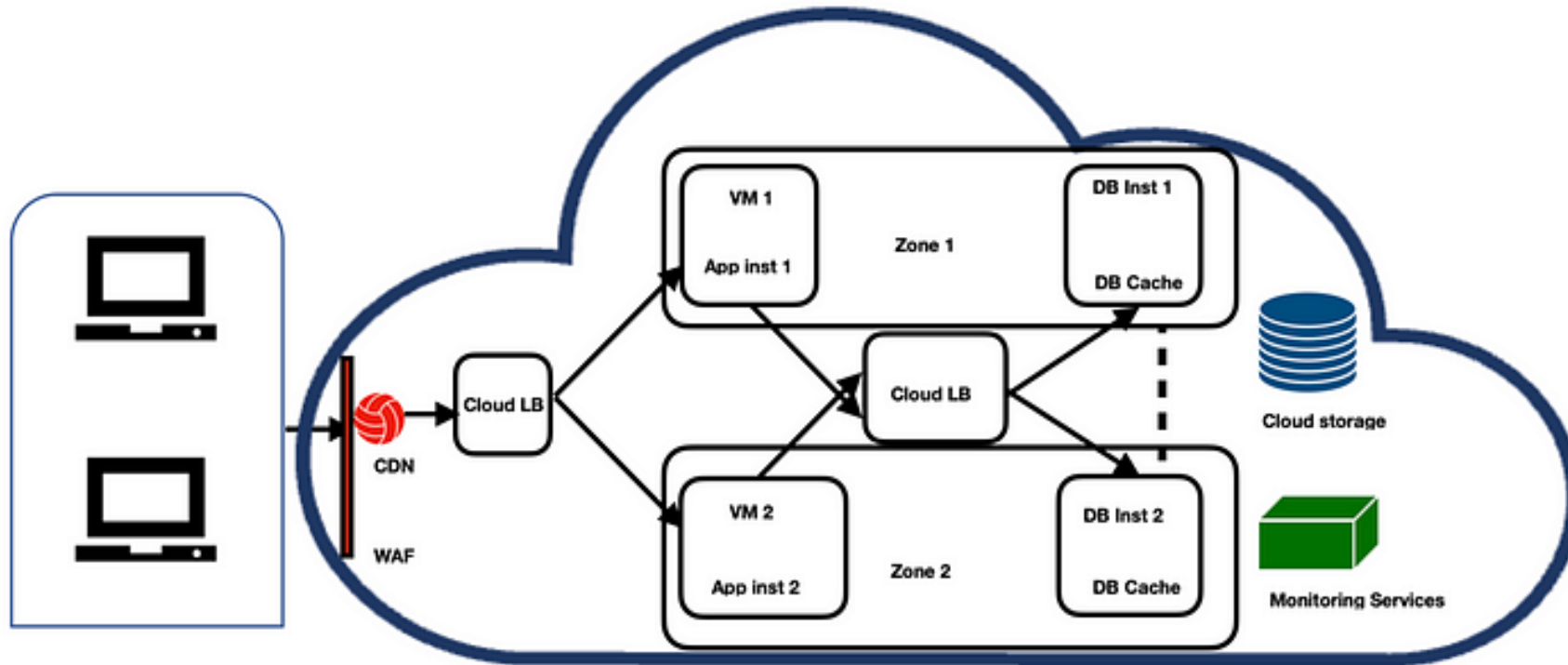
Importancia del Diseño de Sistemas



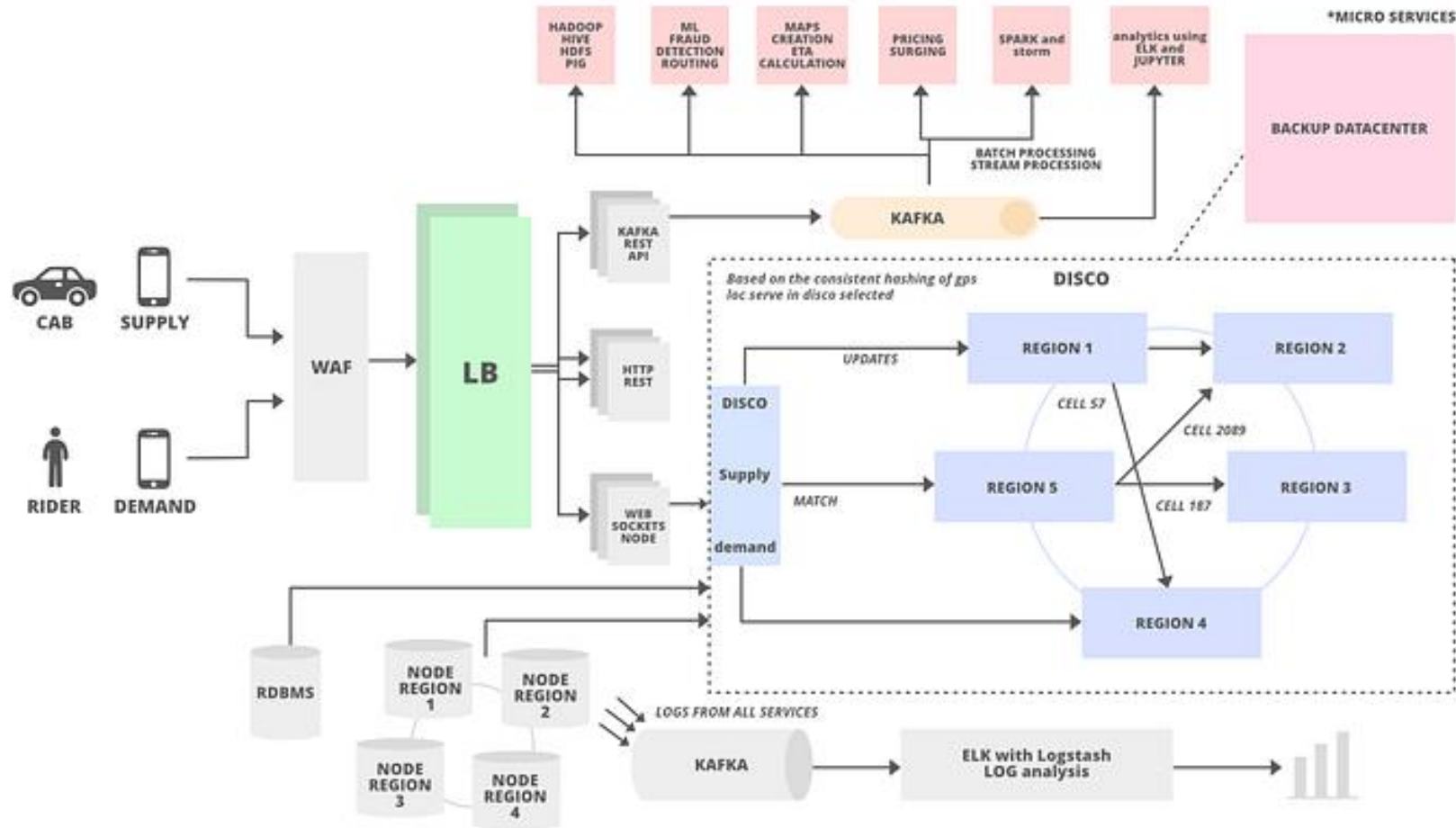
Importancia del diseño de sistemas



Importancia del diseño de sistemas



Importancia del diseño de sistemas



2020

Importancia del diseño de sistemas



2030

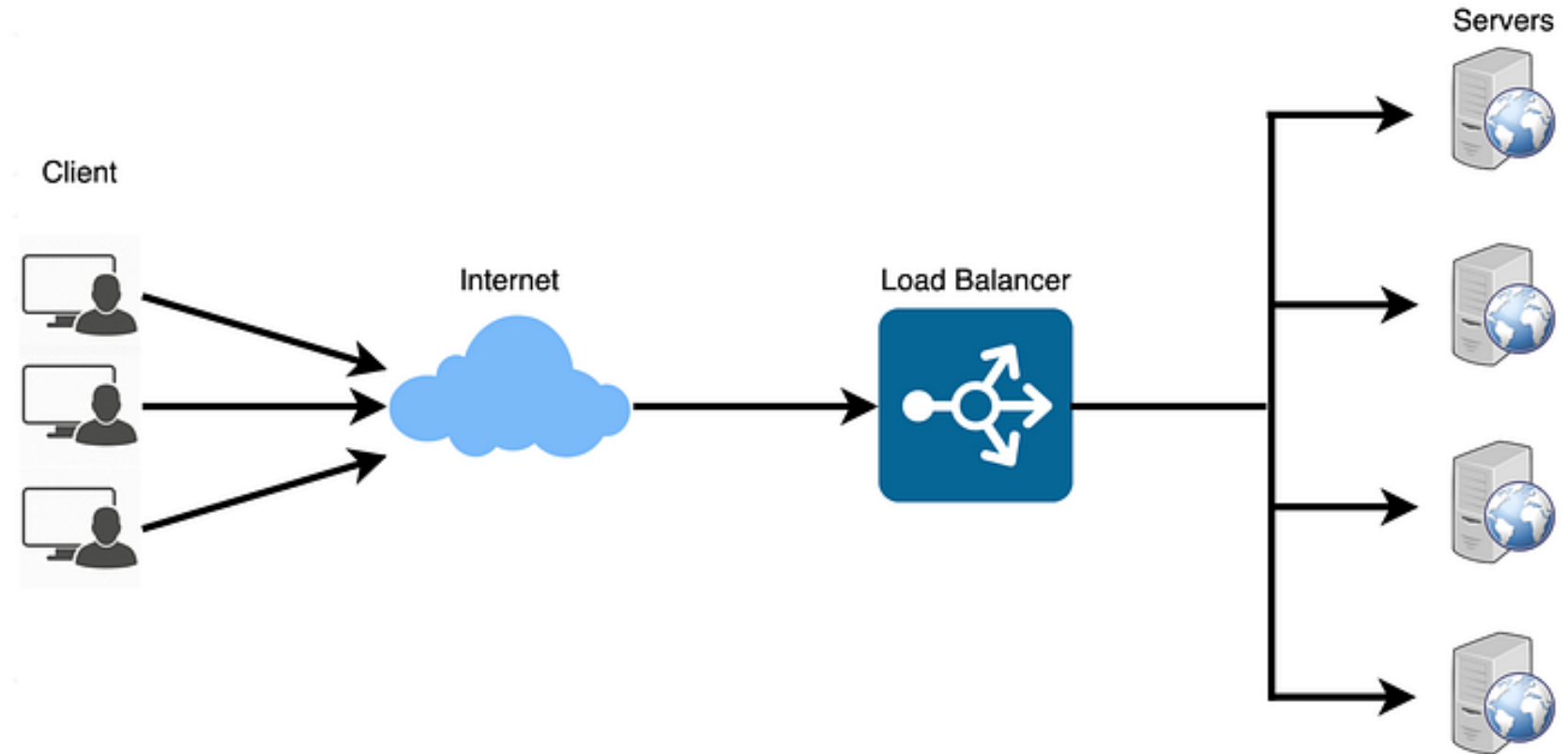
Importancia del diseño de sistemas

"El diseño de sistemas es esencial para alinear la tecnología con los objetivos de negocio, proporcionando una base sólida para la innovación y el éxito a largo plazo."

Conceptos Clave en el Diseño de Sistemas

Load Balancer

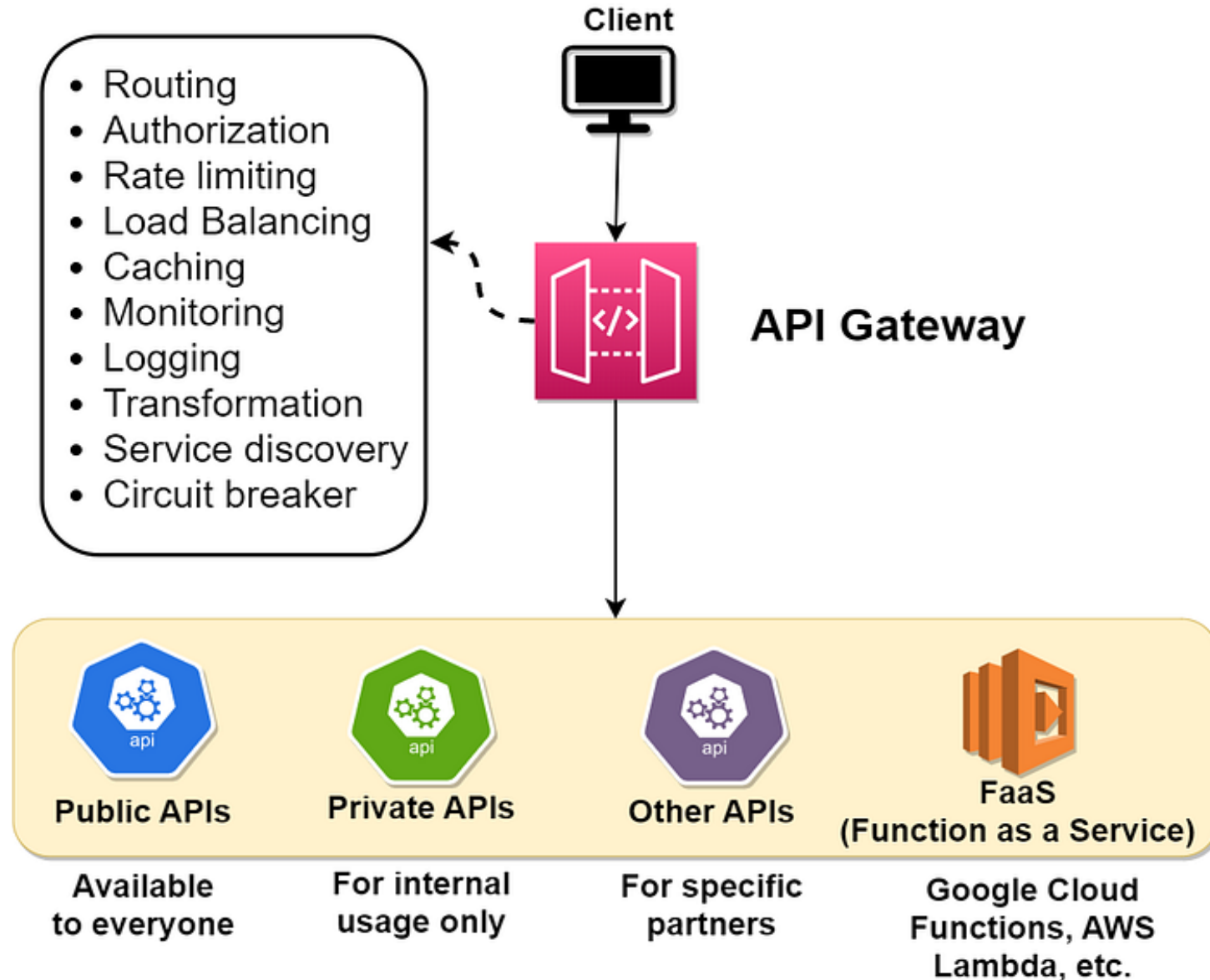
Distribuir el tráfico de red de manera equilibrada entre varios servidores.



Conceptos Clave en el Diseño de Sistemas

API Gateway

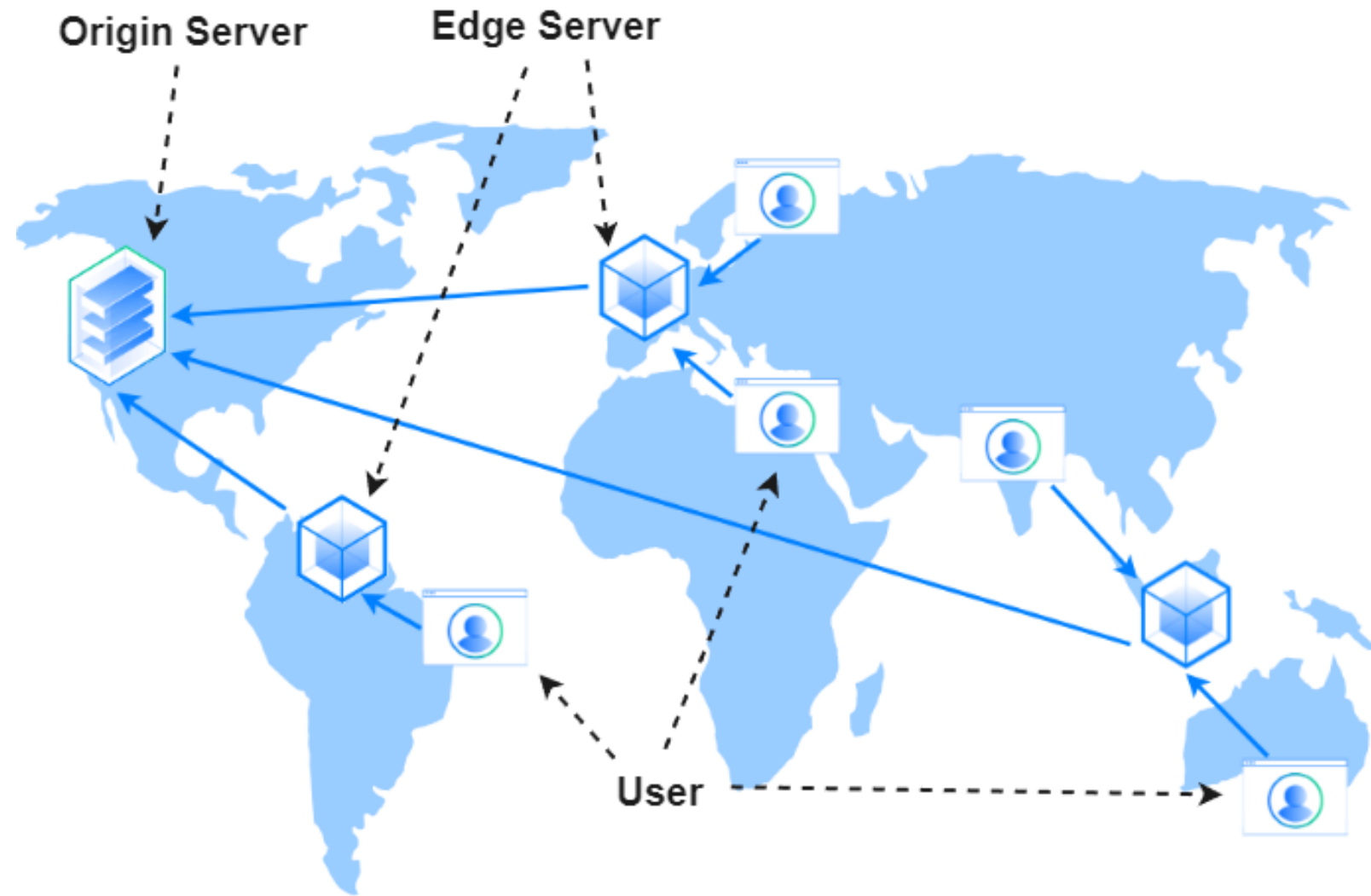
Gestionar y coordinar las solicitudes de API entre clientes y servicios backend.



Conceptos Clave en el Diseño de Sistemas

Content Delivery Network (CDN)

Acelerar la entrega de contenido a los usuarios distribuidos geográficamente.



Content Delivery Network (CDN)

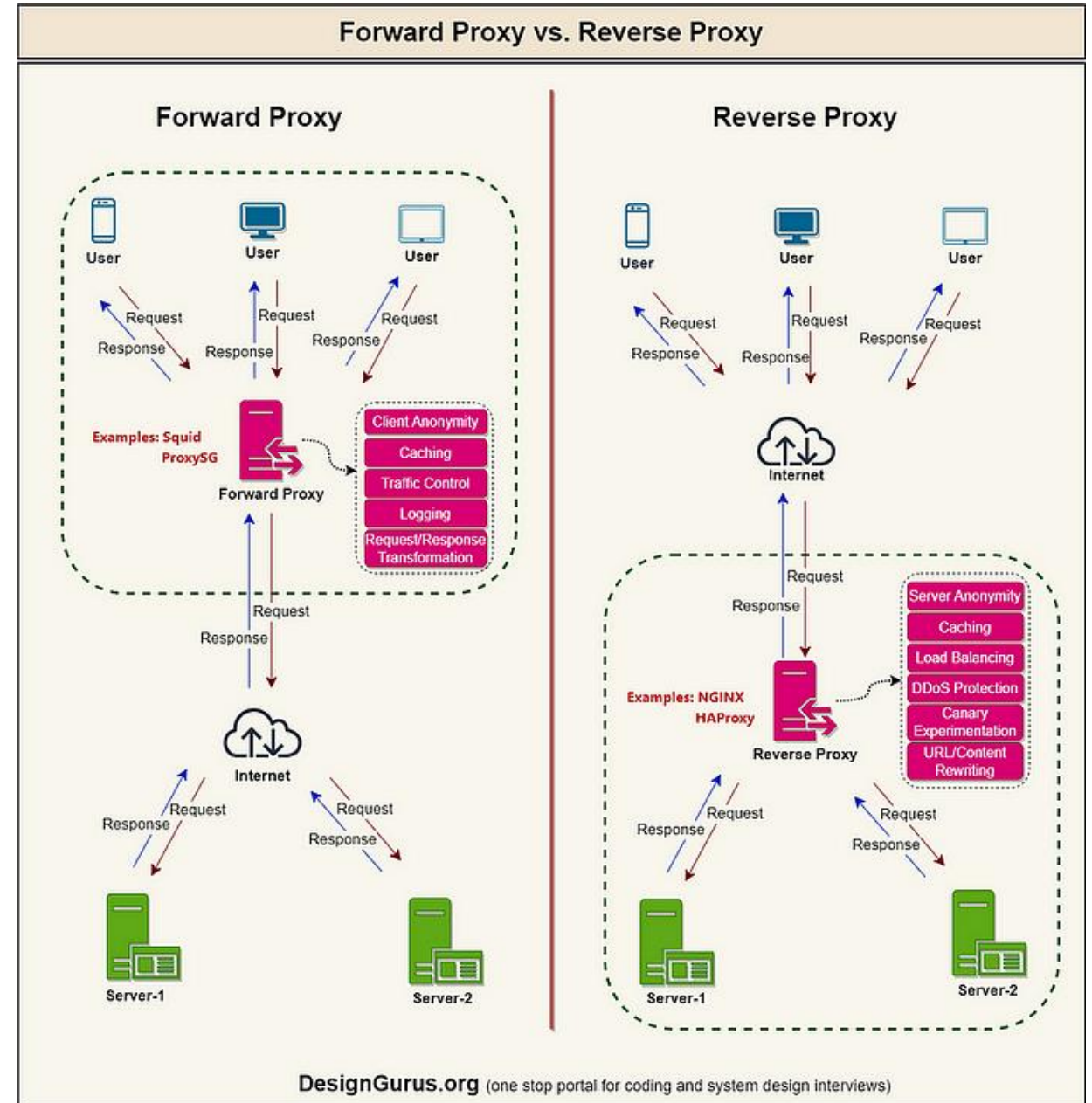
Conceptos Clave en el Diseño de Sistemas

Forward Proxy

Actuar en nombre de los clientes para acceder a recursos en internet. Anonimiza al cliente.

Reverse Proxy

Filtrar el tráfico entrante y distribuirlo a servidores backend. Anonimiza al servidor.



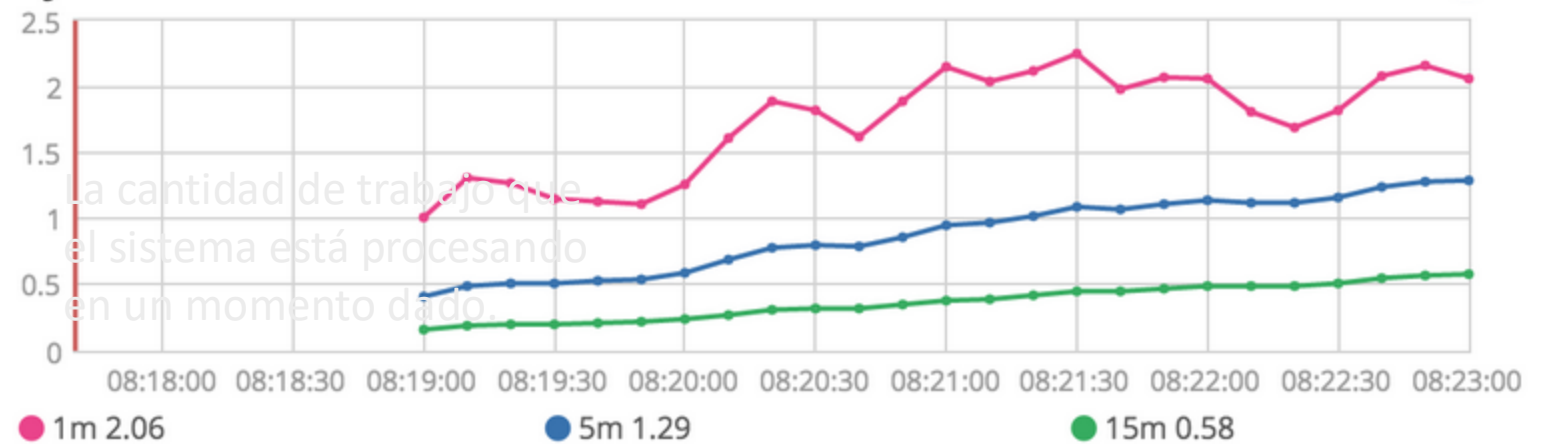
Conceptos Clave en el Diseño de Sistemas

Carga del sistema

La cantidad de trabajo que el sistema está procesando en un momento dado.

La carga de trabajo también determina las tecnologías a utilizar y otros detalles de implementación.

System Load

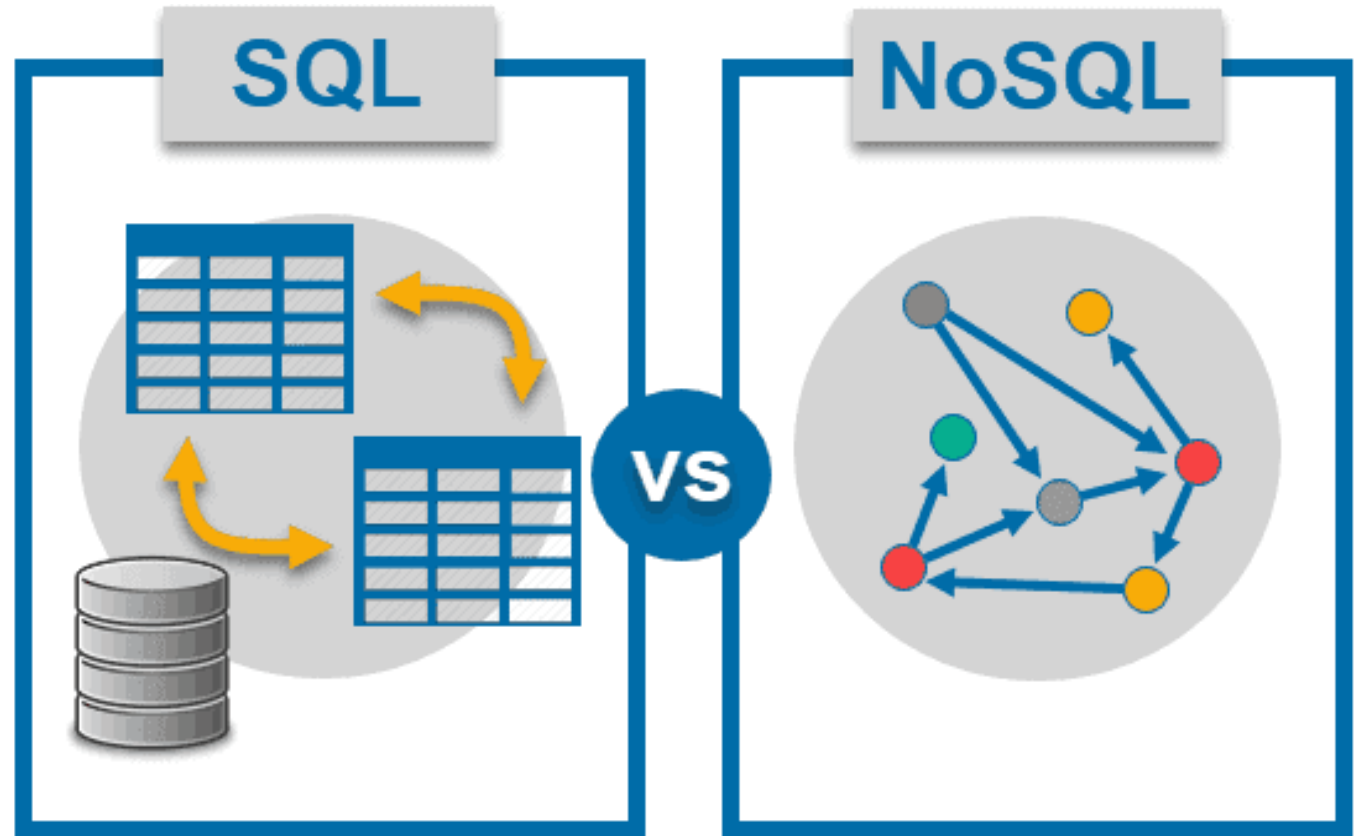


Conceptos Clave en el Diseño de Sistemas

Bases de Datos Apropriadas

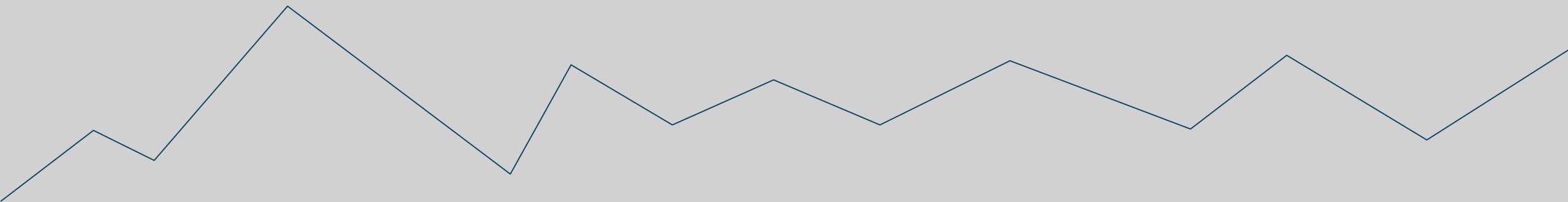
SQL: Requieren transacciones complejas y relaciones entre datos.

NoSQL: Necesitan flexibilidad en la estructura de datos y escalabilidad horizontal.

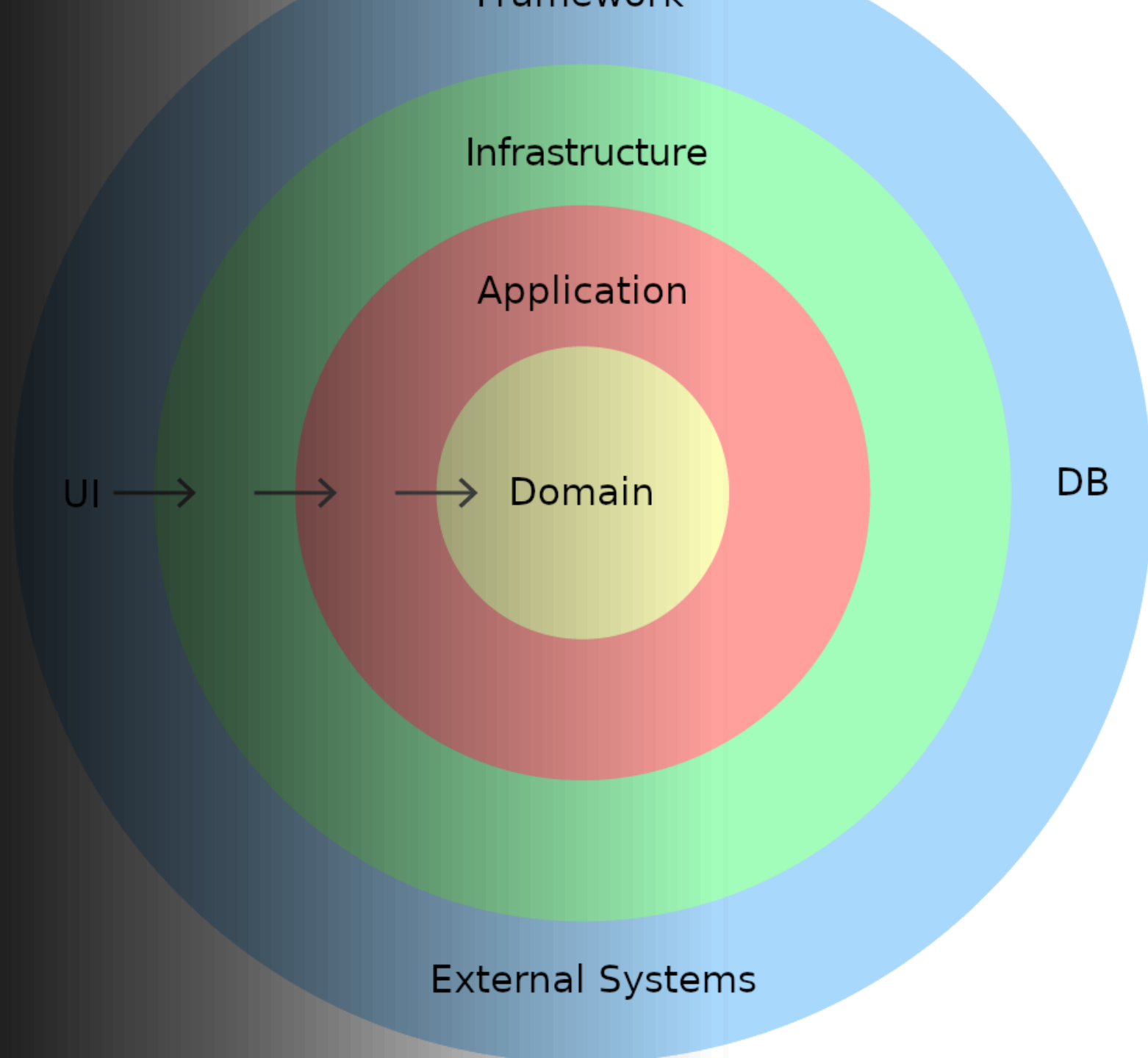


Domain-Driven Design (DDD)

Enfocando el Diseño en el Dominio del Problema



Fundamentos de Domain- Driven Design



Fundamentos de Domain-Driven Design

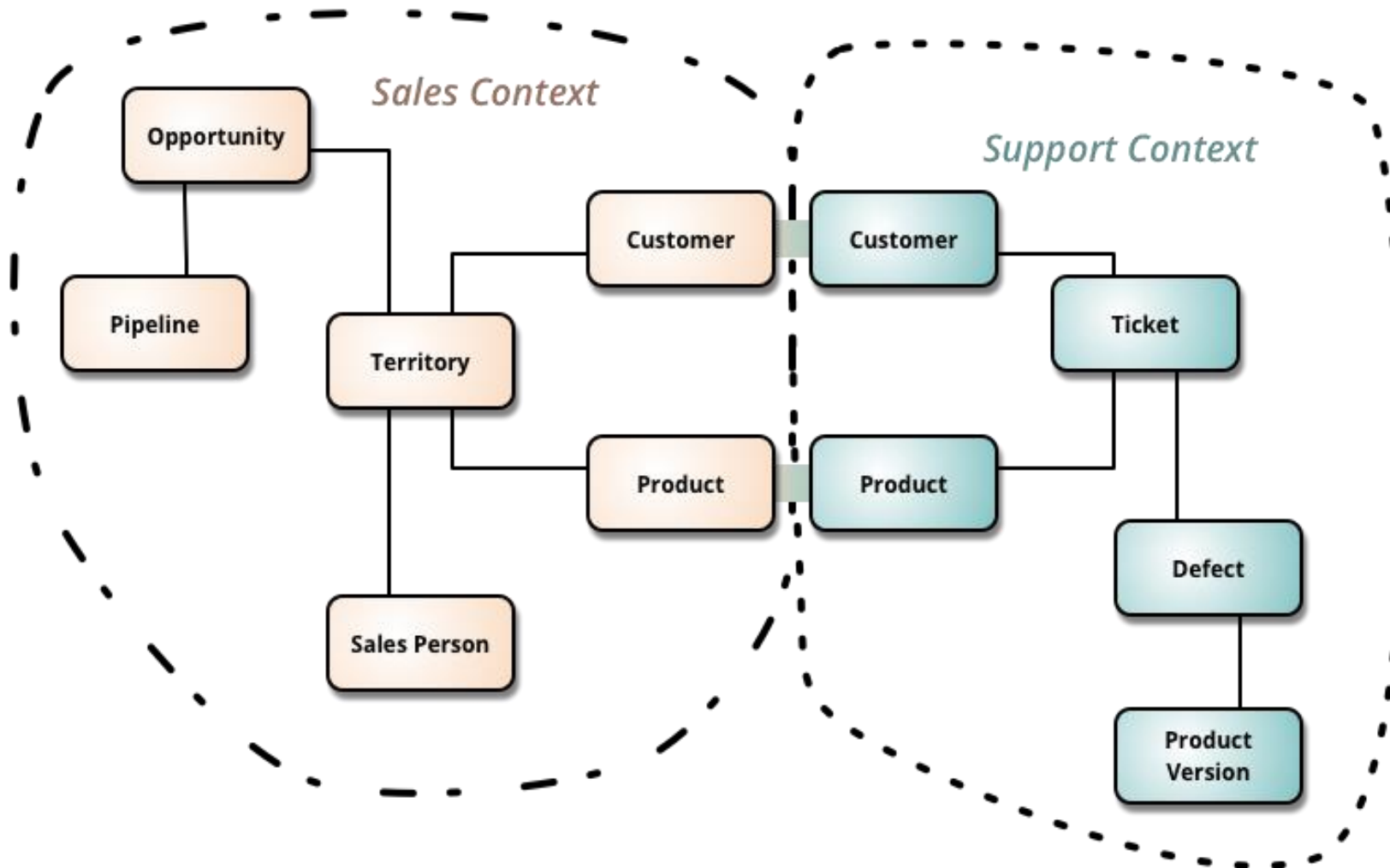
Definición: Enfoque de diseño de software que enfatiza la importancia del dominio de negocio.

Objetivo: Alinear la estructura y el lenguaje del software con el negocio.

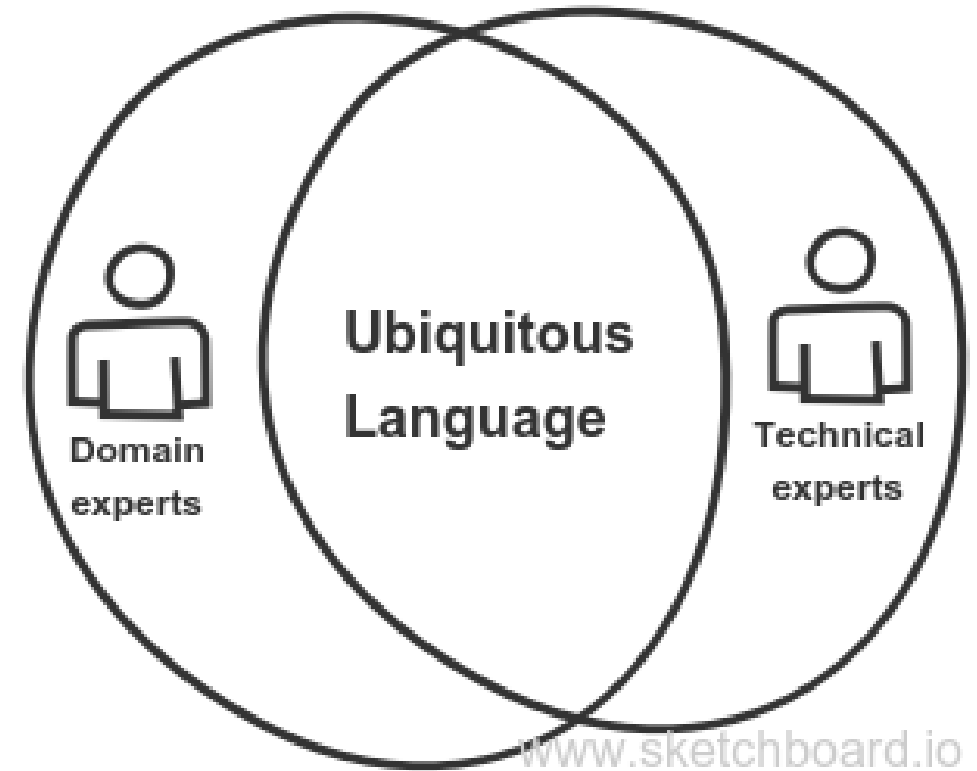
Componentes Clave:

- **Ubiquitous Language (Lenguaje Ubicuo):** Un lenguaje común compartido por desarrolladores y expertos en dominio.
- **Bounded Contexts (Contextos Delimitados):** Dividir el sistema en partes manejables y cohesivas.

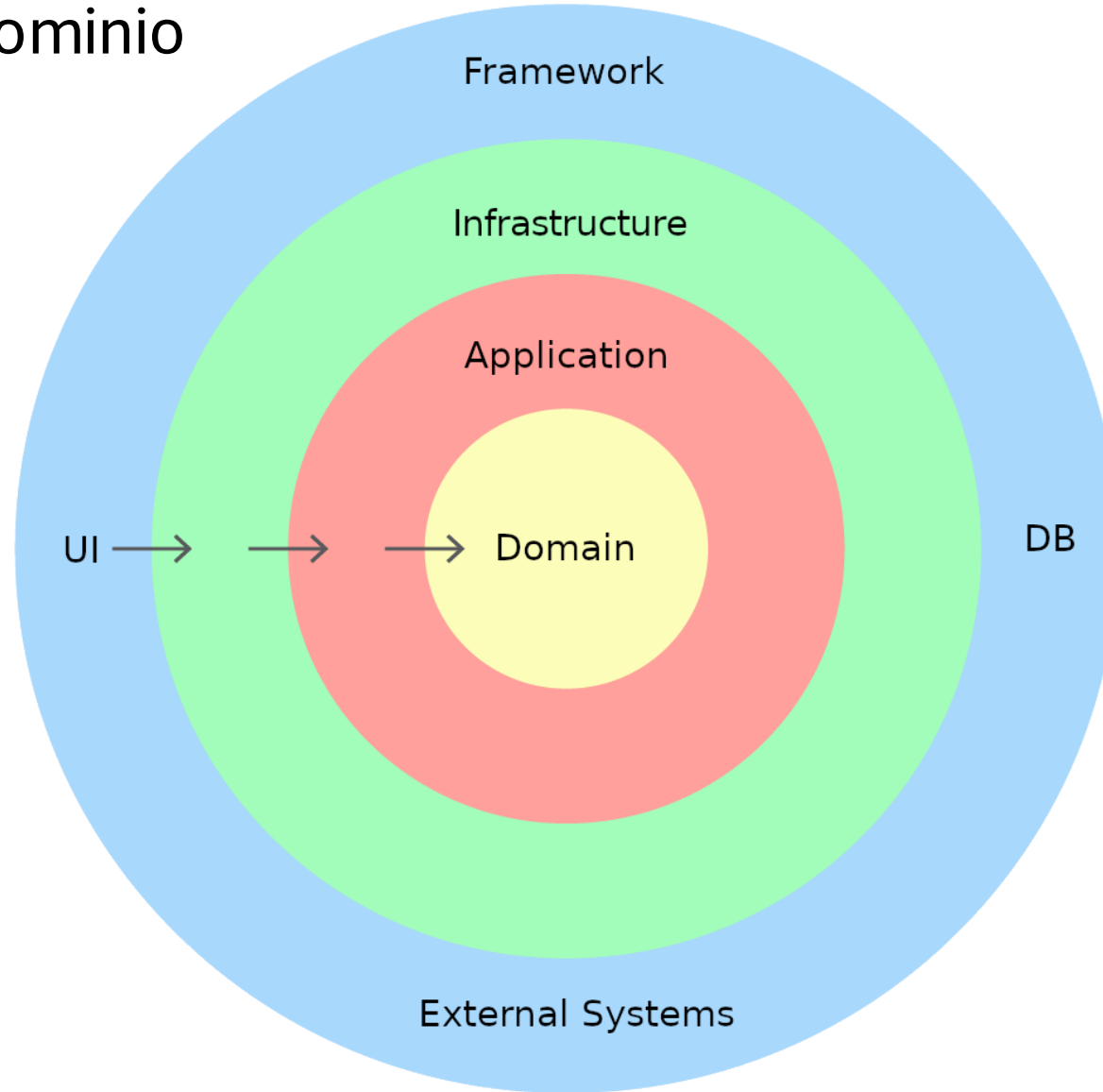
Bounded Contexts



Ubiquitous Language

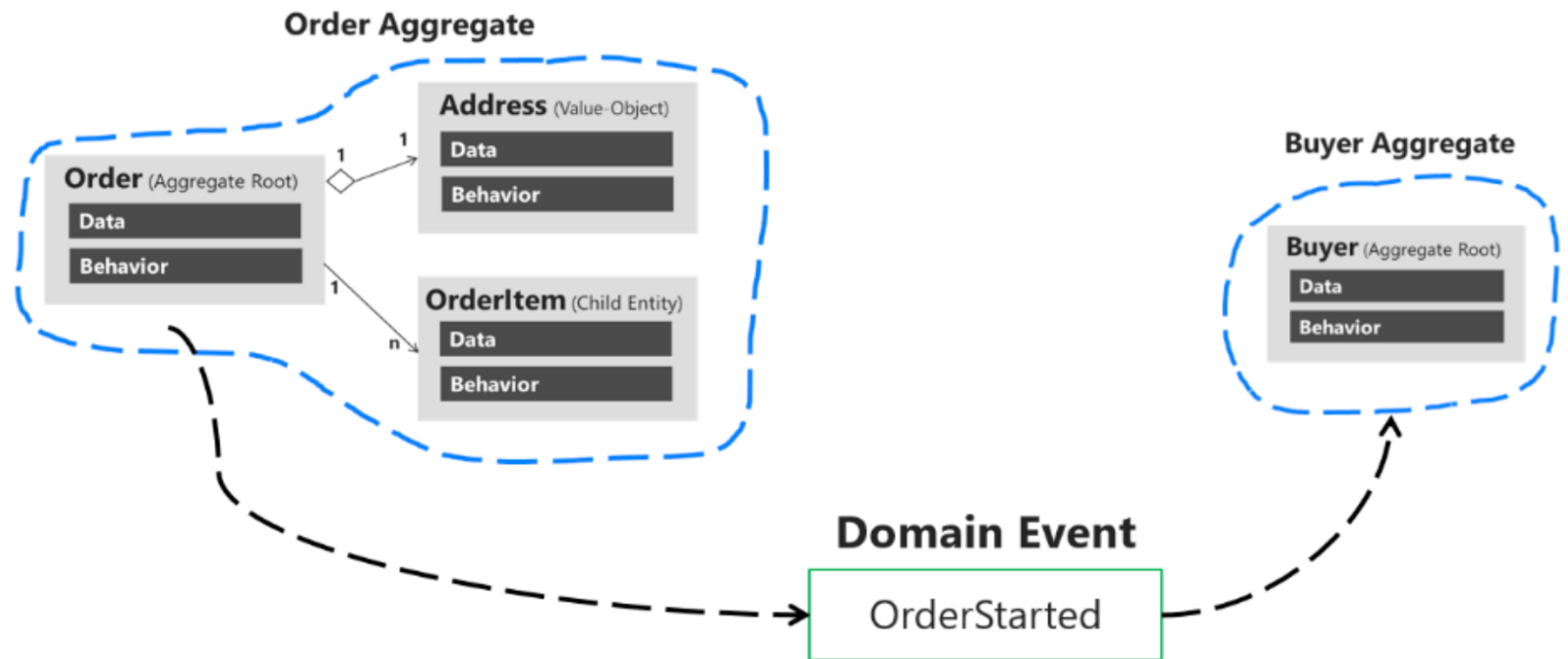


Modelado del Dominio



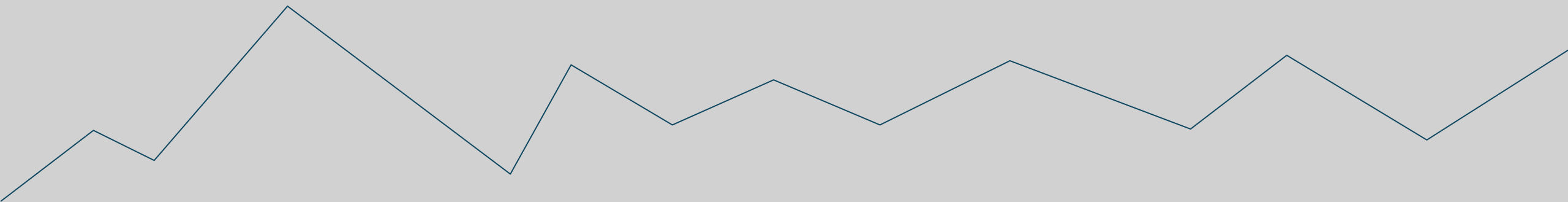
Entidades y Agregados


- **Entidades:** Objetos con identidad única.
 - **Ejemplo:** Cliente, Pedido.
- **Agregados:** Conjunto de objetos tratados como una unidad.
 - **Ejemplo:** Un Pedido y sus Items
- **Domain Event:** Eventos dentro de un mismo dominio.
 - **Ejemplo:** Orden Started



Arquitectura Hexagonal (Ports and Adapters)

Separación de Concerns a través de Puertos y Adaptadores





Principios de la Arquitectura Hexagonal



Principios de la Arquitectura Hexagonal

Definición: Enfoque arquitectónico que facilita la separación de concerns y la flexibilidad del sistema.

Objetivo: Crear sistemas que sean fácilmente testables, mantenibles y adaptables a cambios.

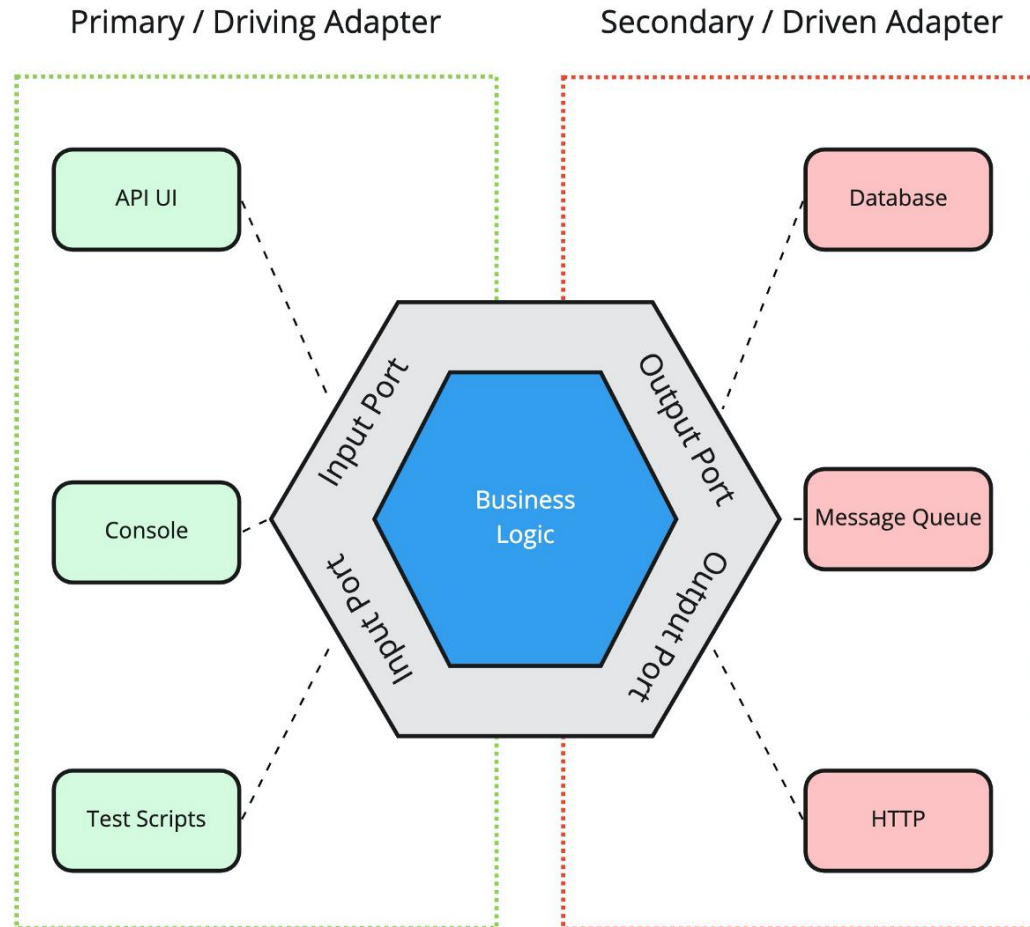
Componentes Clave:

⑩ **Núcleo (Core):** Contiene la lógica de negocio.

⑩ **Puertos (Ports):** Interfaces que definen puntos de entrada y salida del sistema.

⑩ **Adaptadores (Adapters):** Implementaciones concretas que interactúan con el mundo exterior.

Componentes: Núcleo, Puertos y Adaptadores

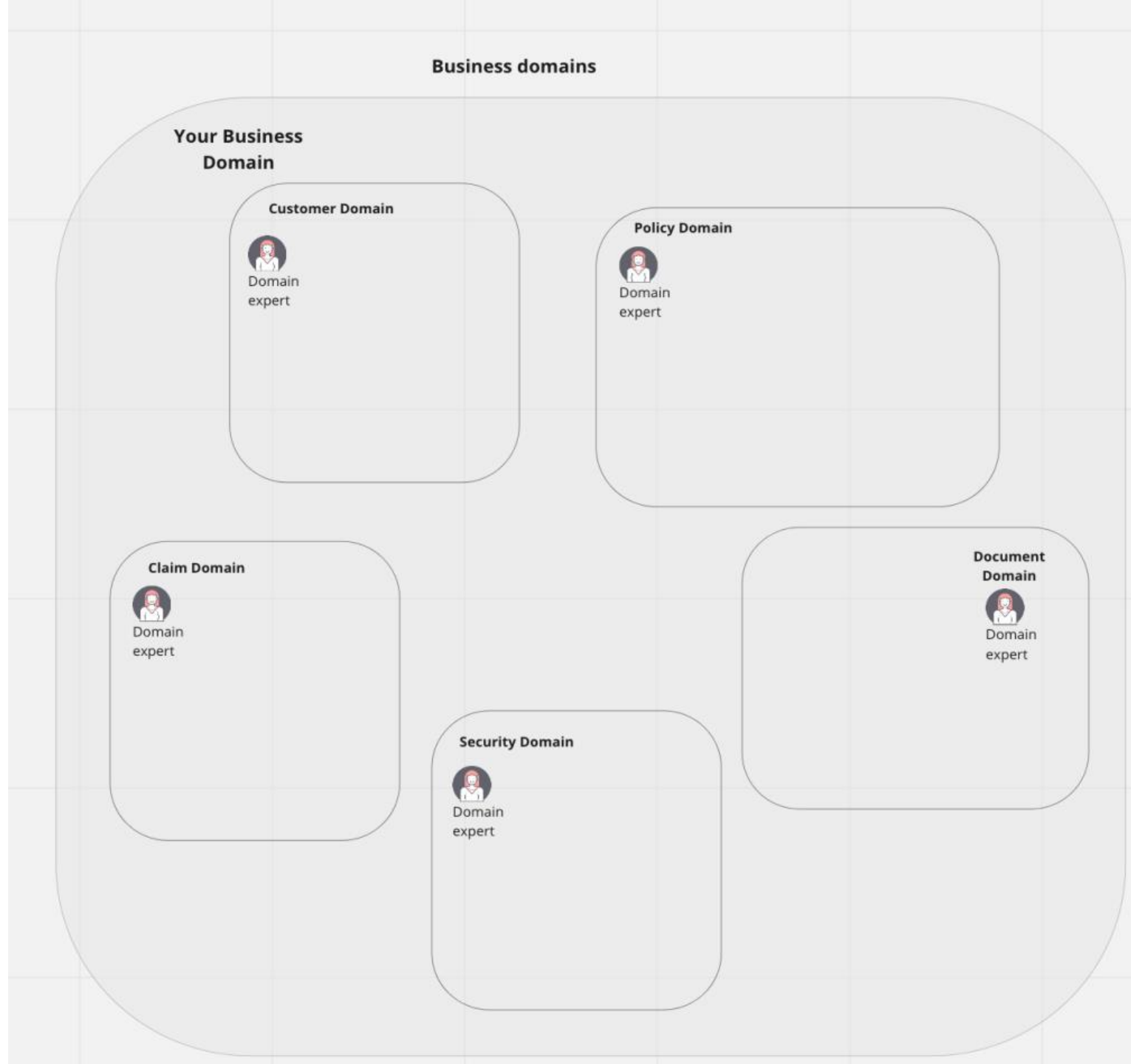


Mejora la modularidad y la mantenibilidad del código.

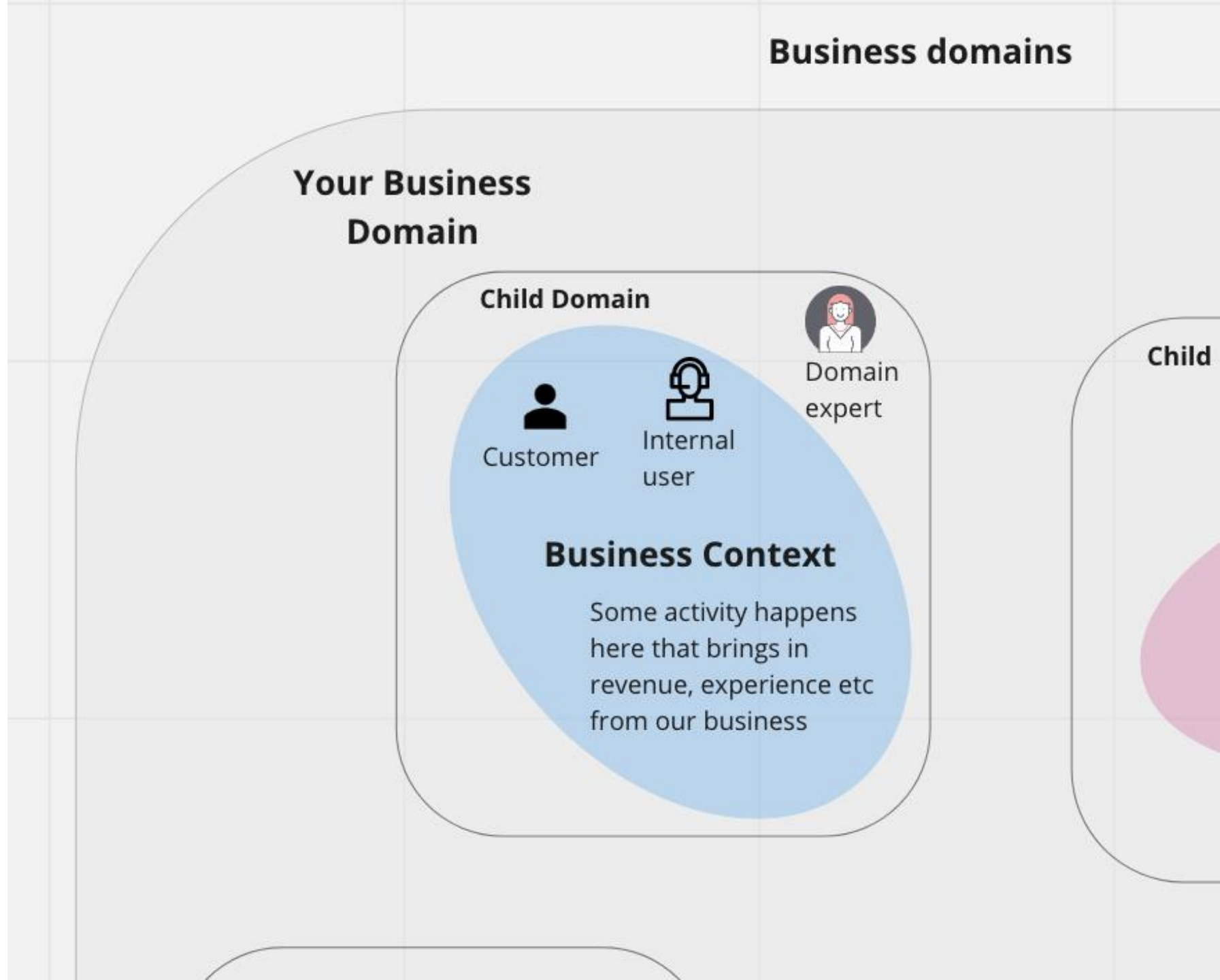
Flexibilidad Tecnológica: Permite la sustitución de adaptadores sin afectar la lógica de negocio en el núcleo.

Facilidad para Testing: Permitir la sustitución de adaptadores reales por adaptadores simulados (mock) durante las pruebas.

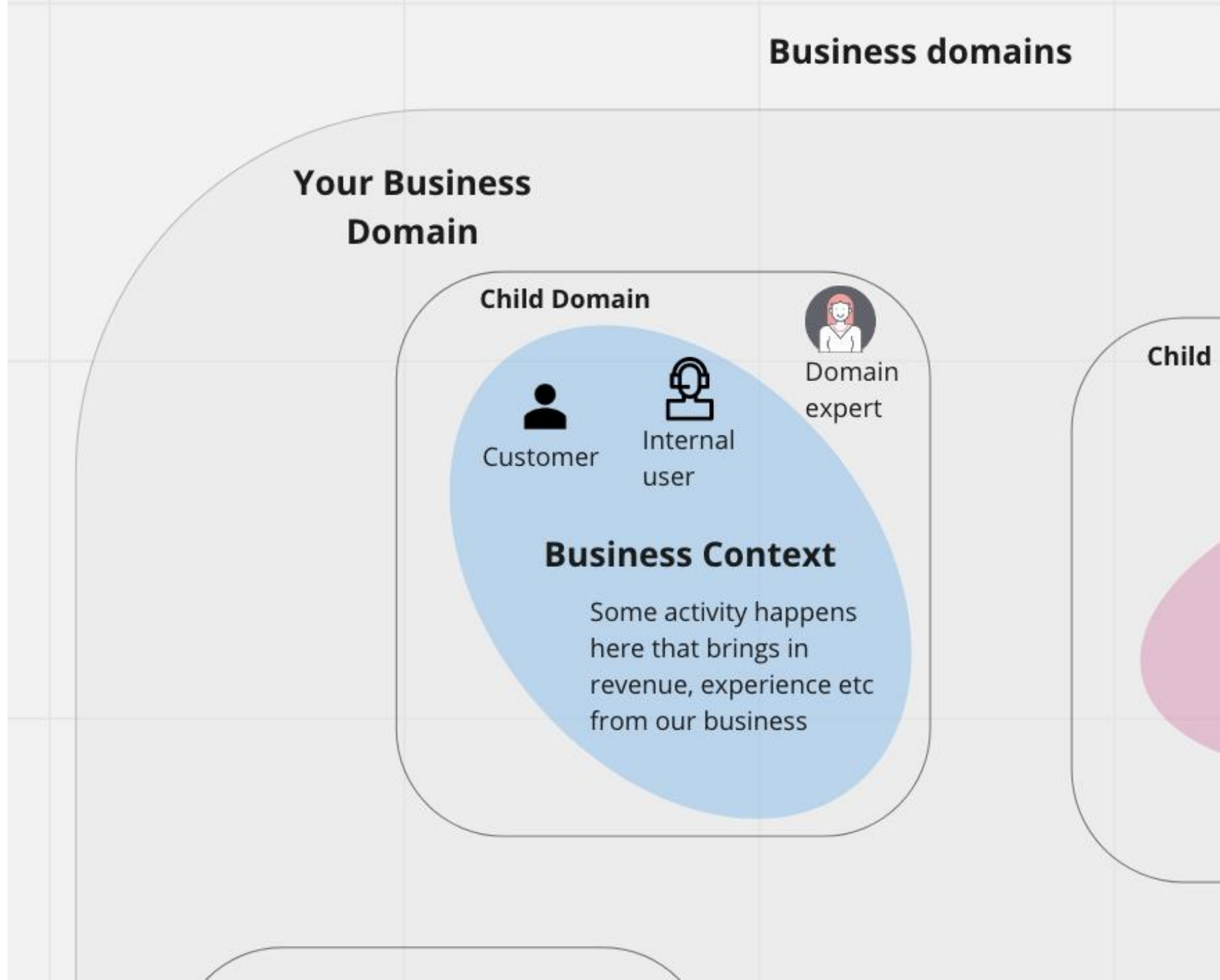
DDD y Arquitectura Hexagonal en la práctica



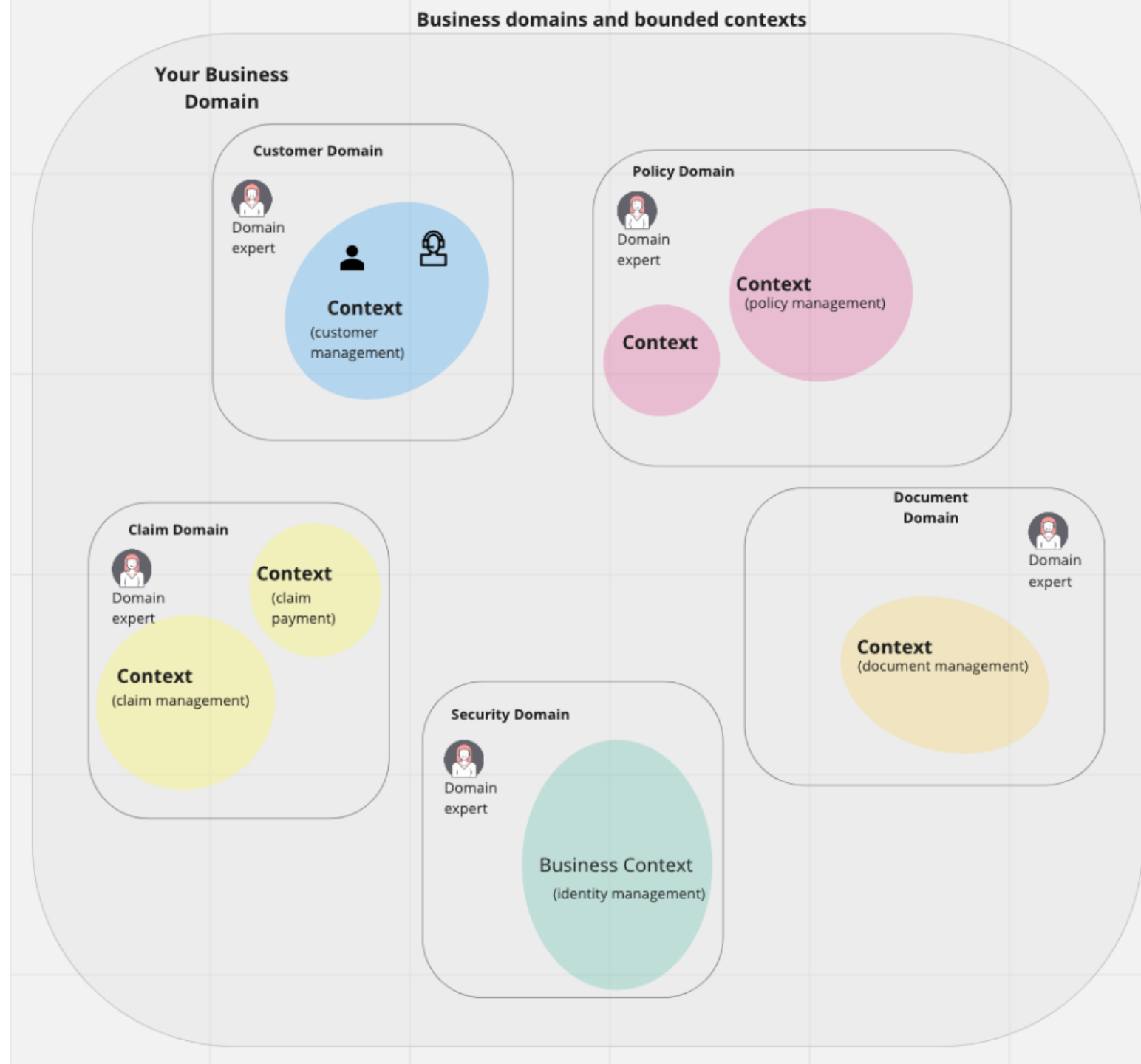
DDD y Arquitectura Hexagonal en la práctica



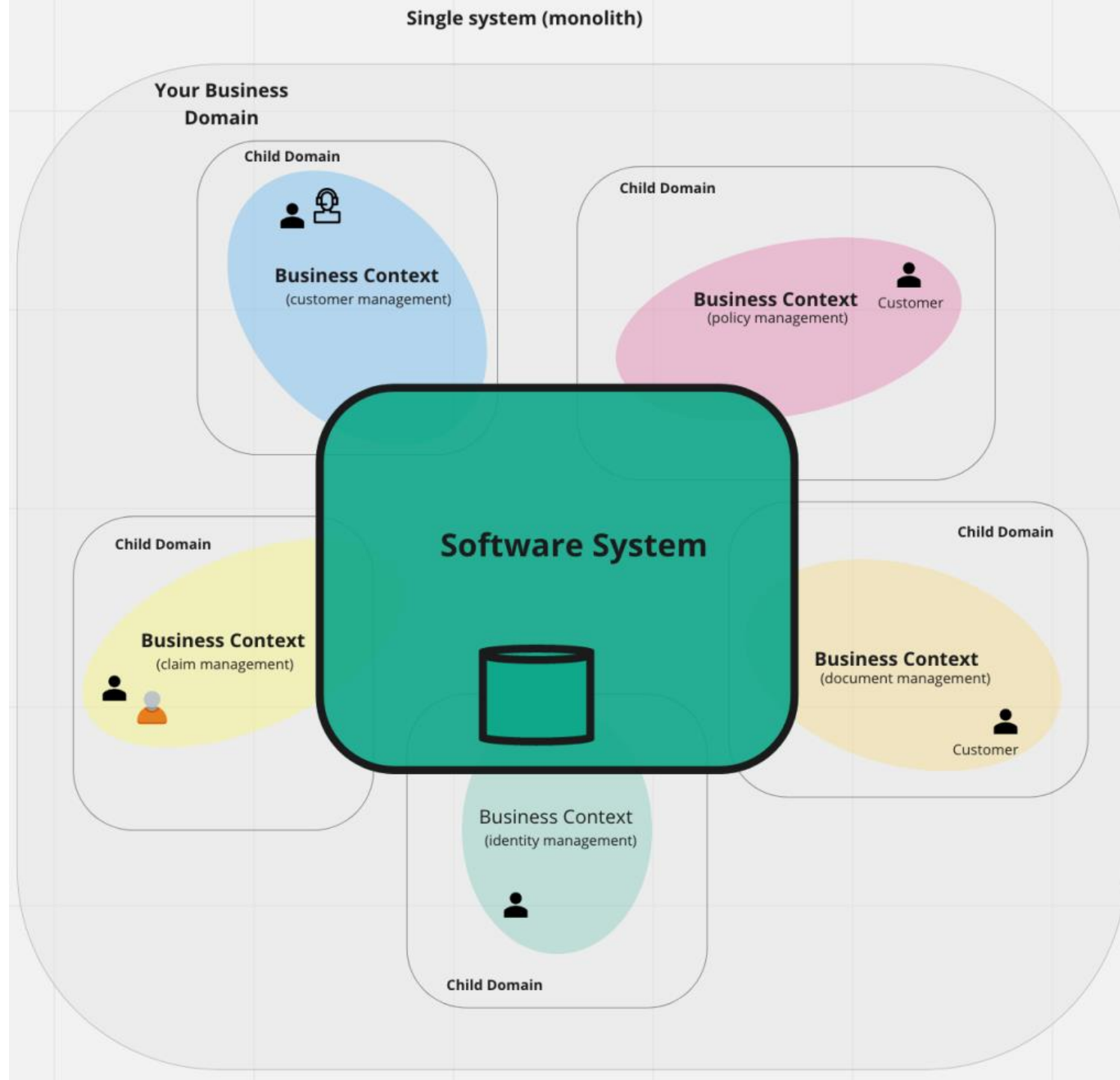
DDD y Arquitectura Hexagonal en la práctica



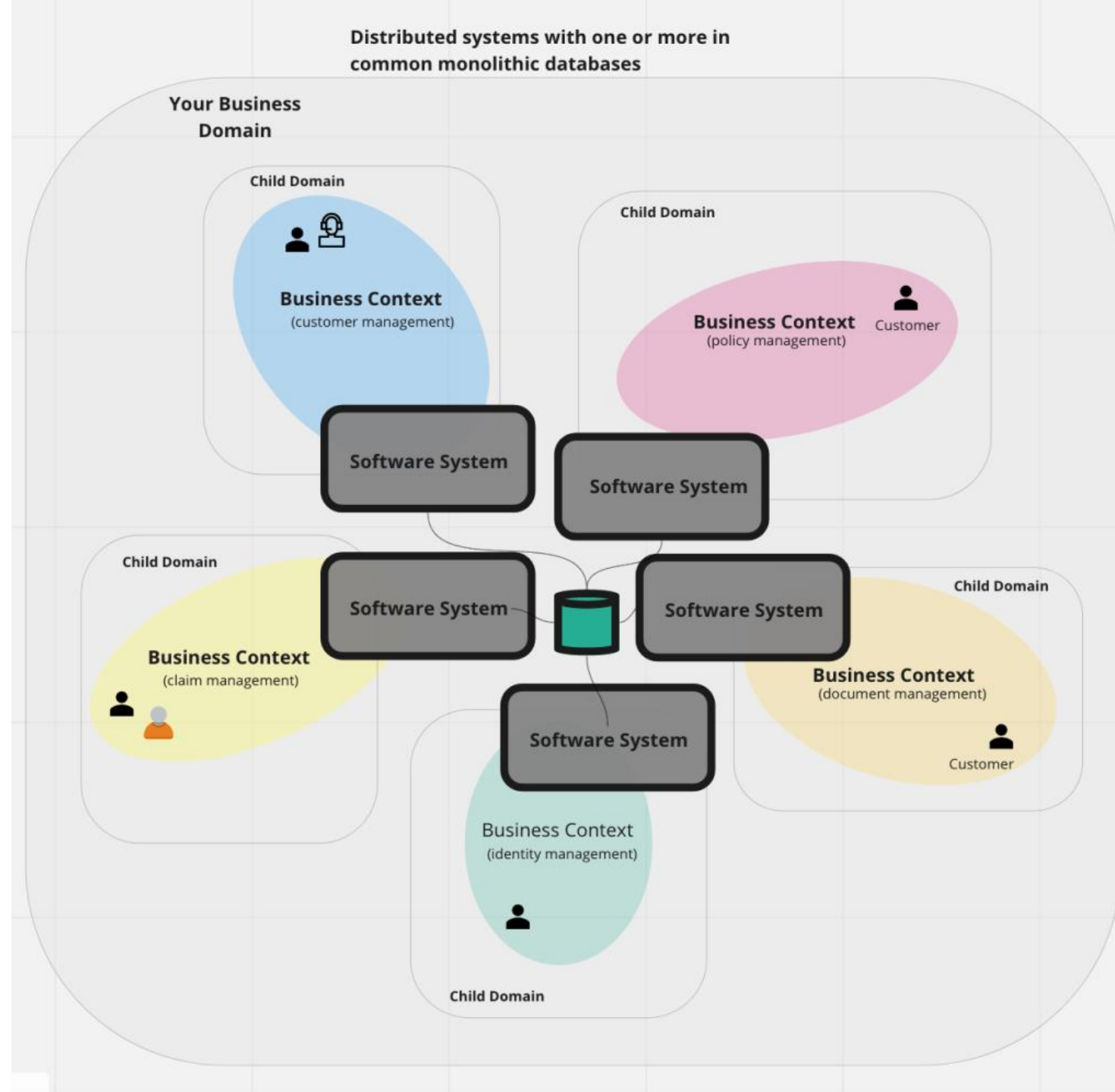
DDD y Arquitectura Hexagonal en la práctica



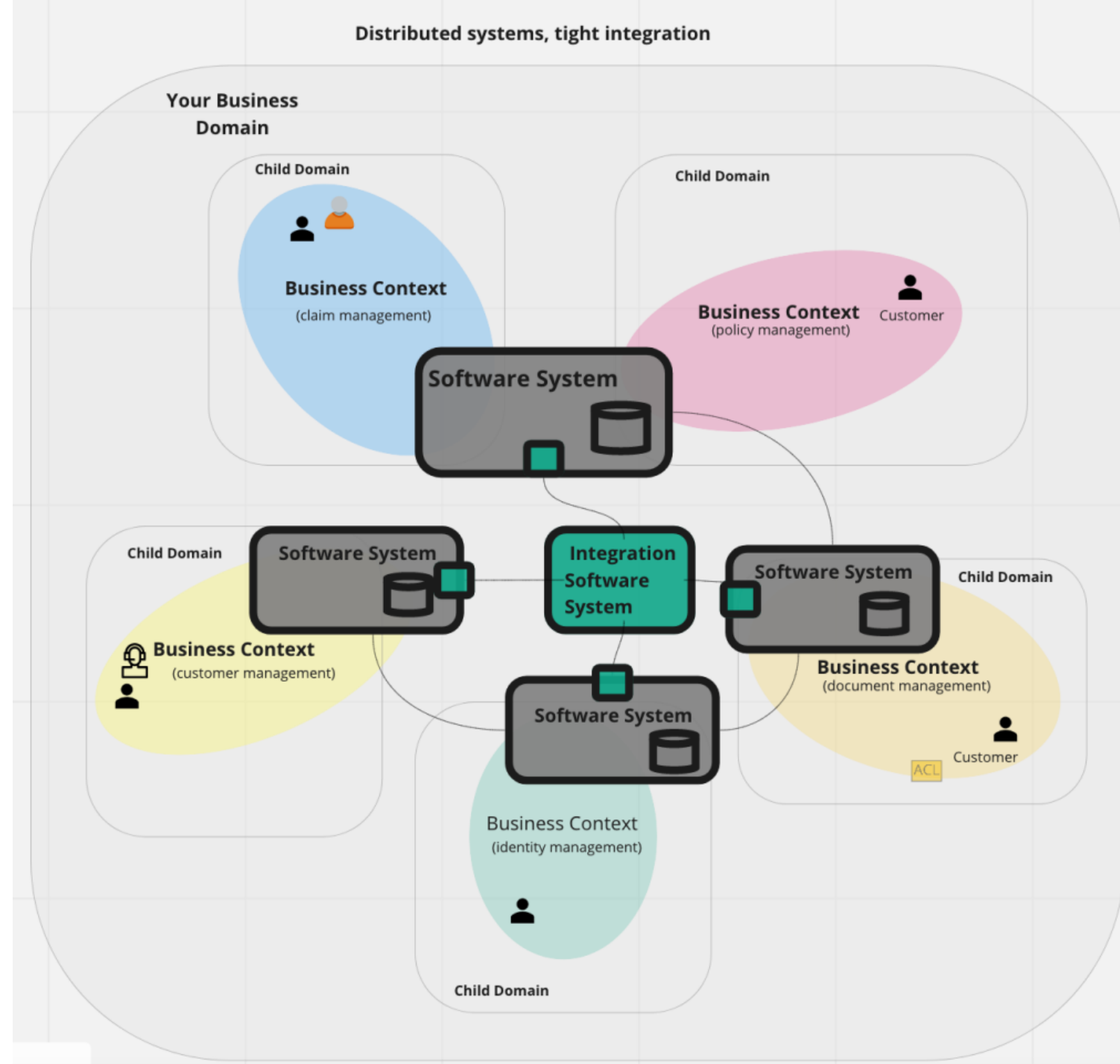
DDD y Arquitectura Hexagonal en la práctica



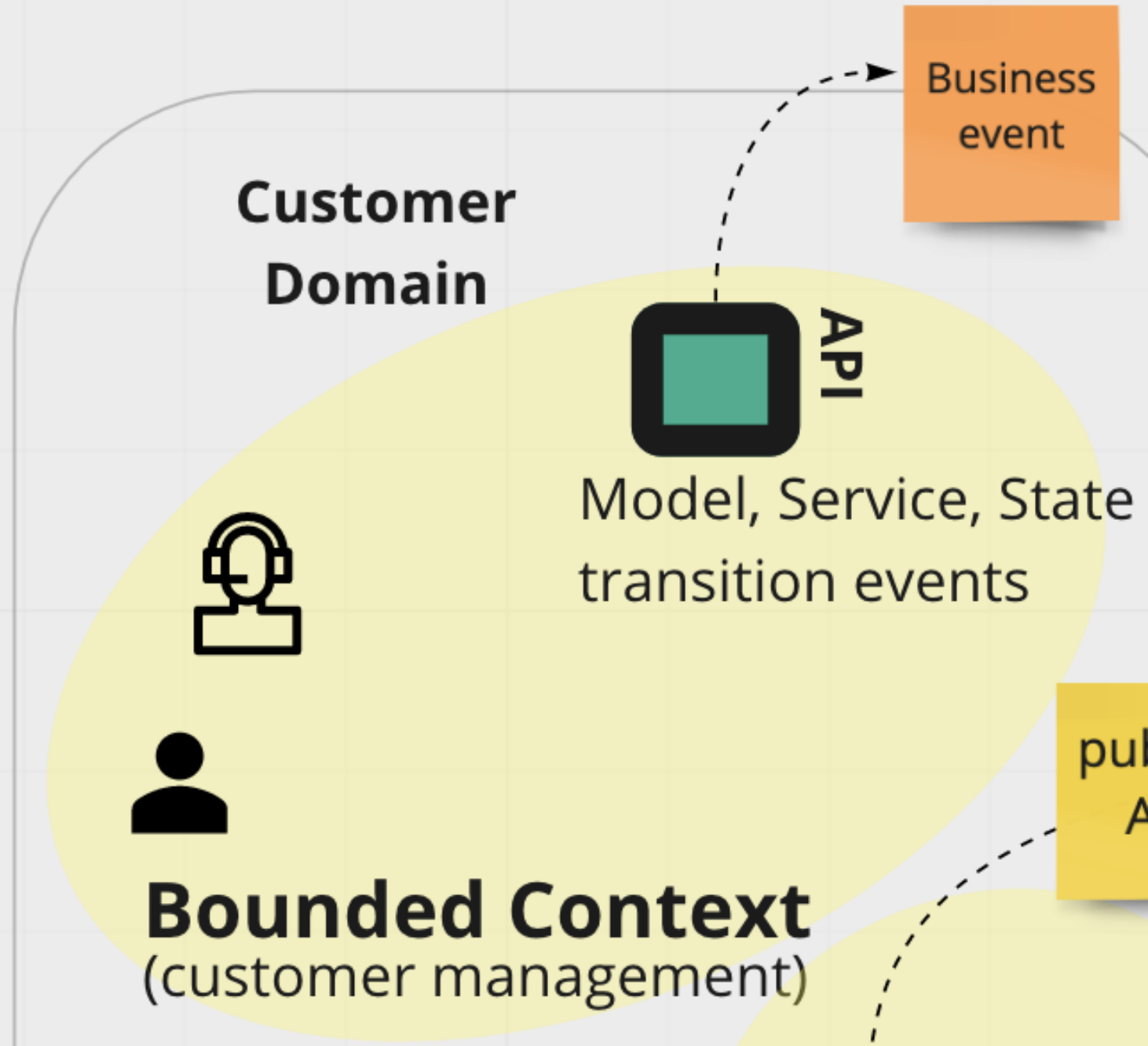
DDD y Arquitectura Hexagonal en la práctica



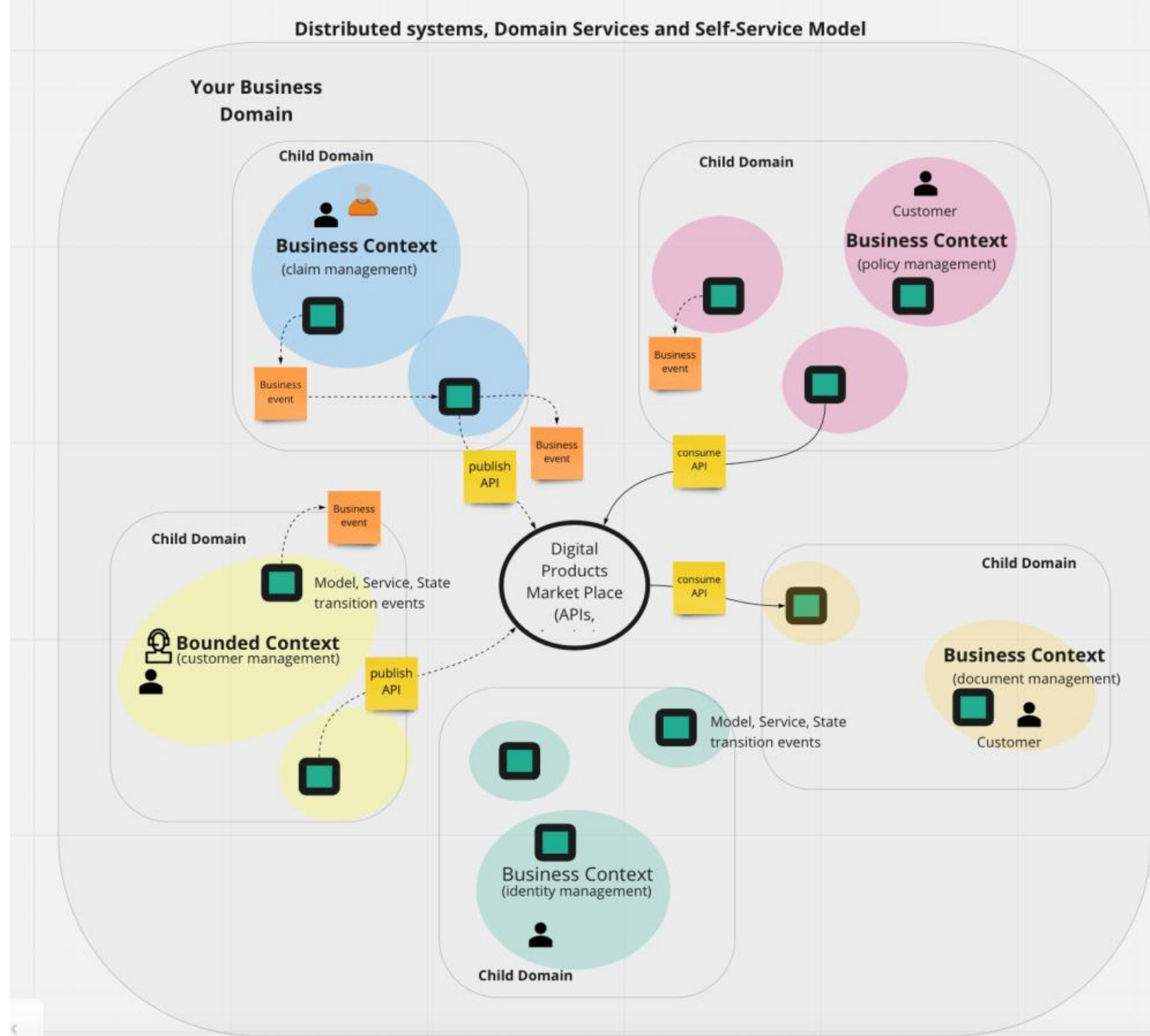
DDD y Arquitectura Hexagonal en la práctica



DDD y Arquitectura Hexagonal en la práctica

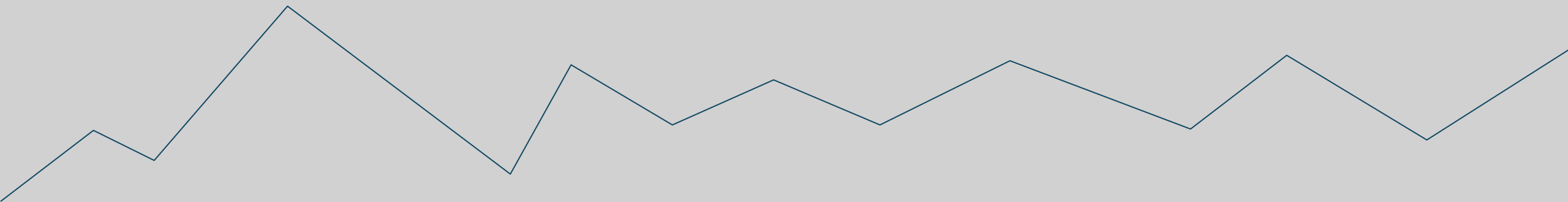


DDD y Arquitectura Hexagonal en la práctica



Arquitectura de Microservicios

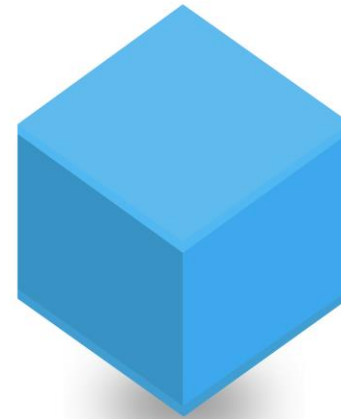
Descomponiendo Sistemas en Servicios Independientes



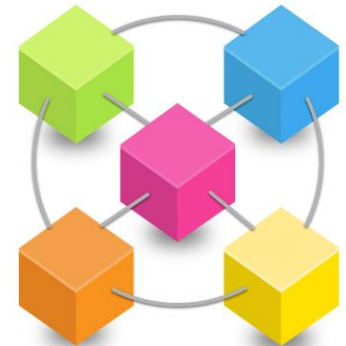
Introducción a los Microservicios

- **Definición:**
 - Estilo arquitectónico donde una aplicación se construye como un conjunto de pequeños servicios, cada uno ejecutando su propio proceso y comunicándose a través de interfaces bien definidas (normalmente HTTP/REST).
- **Objetivo:**
 - Crear aplicaciones más modulares y flexibles.

Monolithic



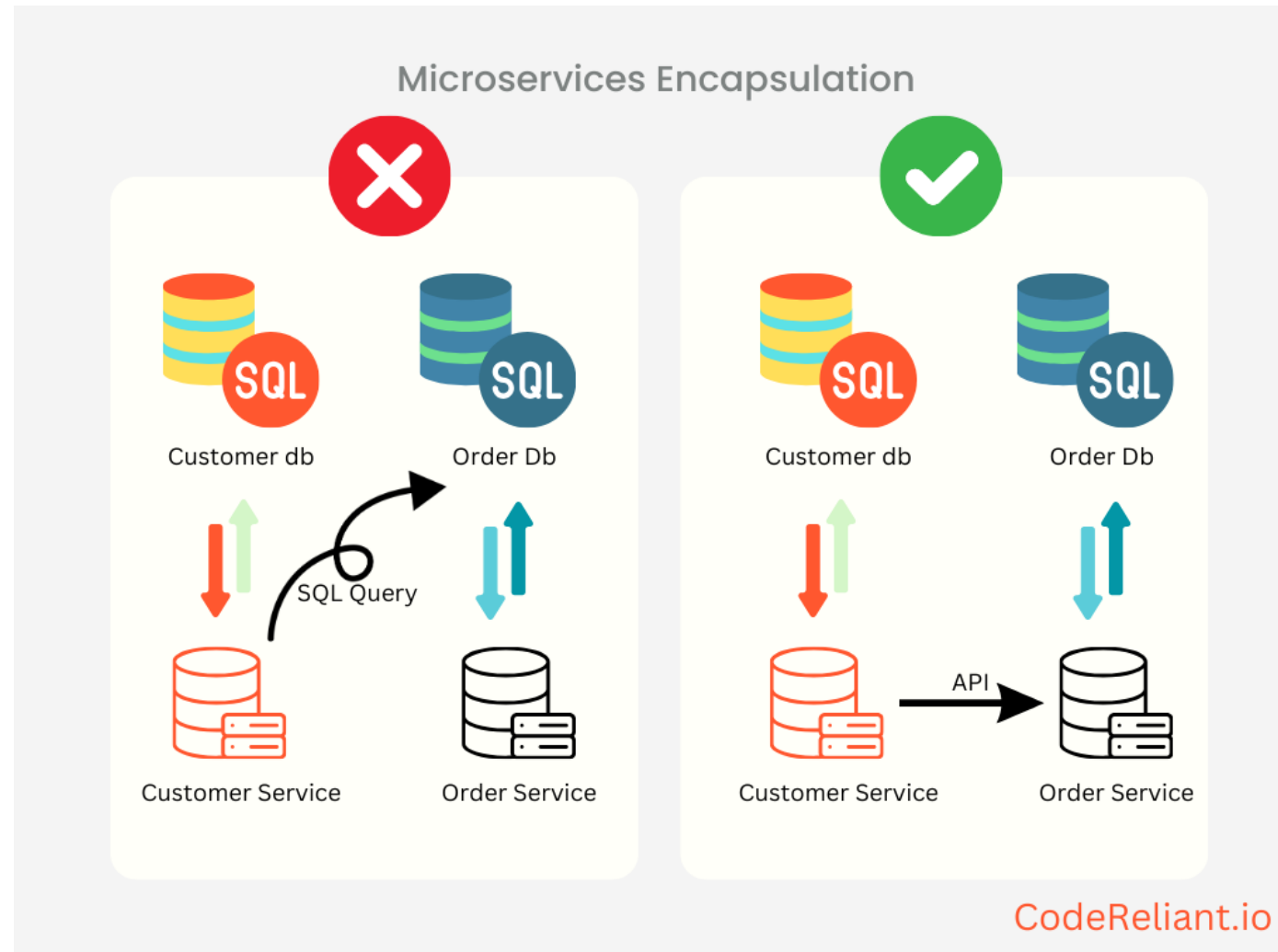
Microservices



Principios de diseño de Microservicios

- **Descomposición:**
 - Dividir aplicaciones monolíticas en servicios pequeños y autónomos.
- **Independencia:**
 - Cada microservicio puede ser desarrollado, desplegado y escalado de manera independiente.
- **Encapsulamiento:**
 - Cada servicio oculta su implementación interna y expone solo interfaces públicas.

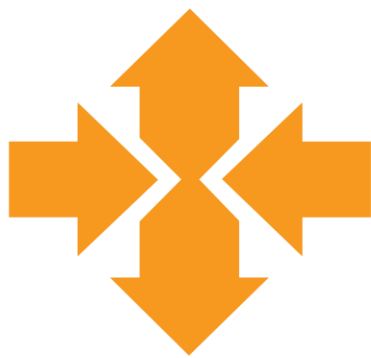
Principios de diseño de Microservicios



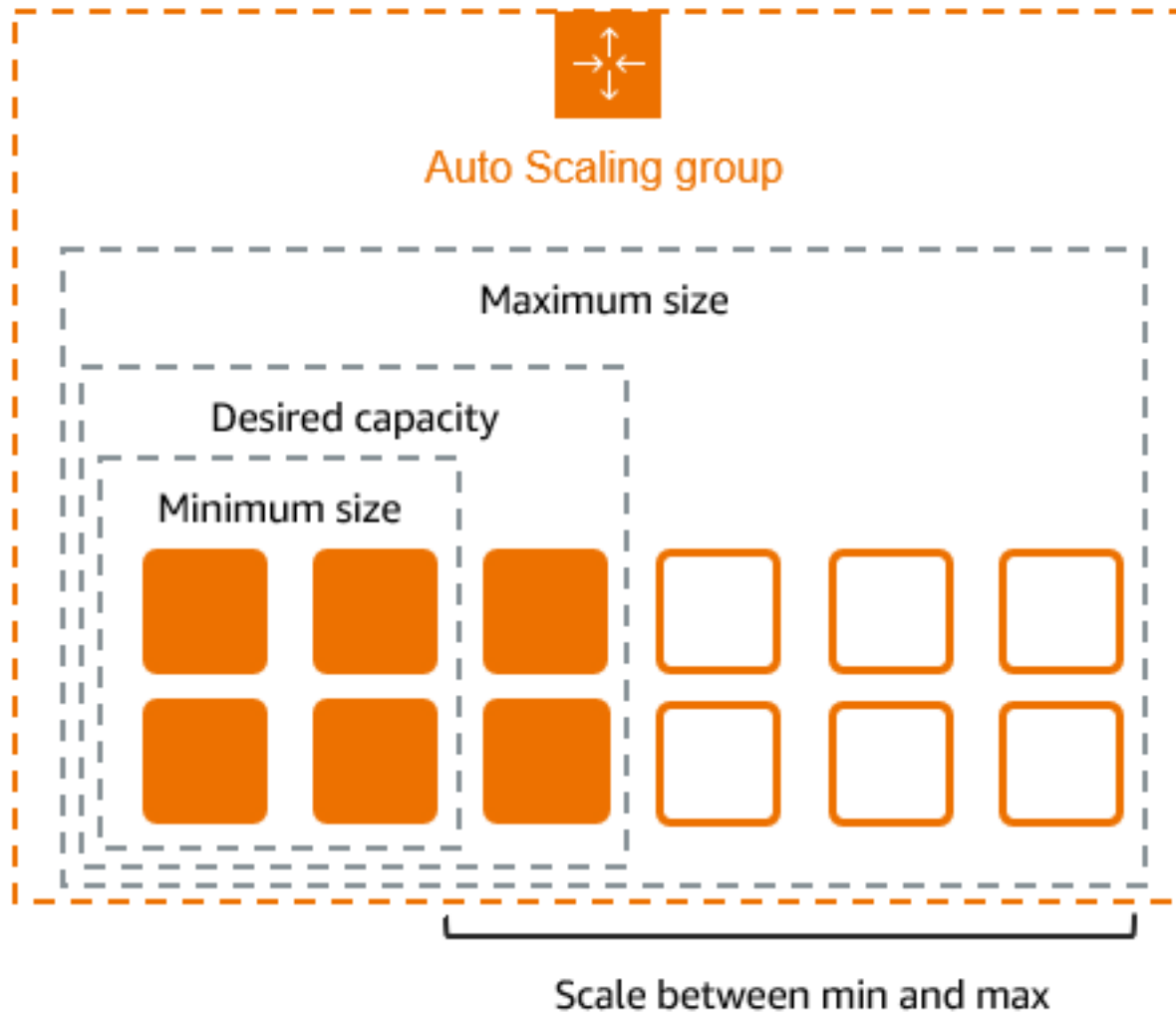
Beneficios de la arquitectura de Microservicios

- **Escalabilidad:**
 - Permite escalar servicios individualmente según las necesidades.
- **Despliegue Continuo:**
 - Facilita la implementación de nuevas características y correcciones de errores.
- **Resiliencia:**
 - Aisla fallos, evitando que un problema en un servicio afecte a toda la aplicación.
- **Flexibilidad Tecnológica:**
 - Permite utilizar diferentes tecnologías y lenguajes de programación para diferentes servicios.

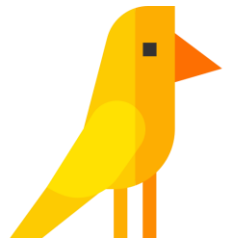
Beneficios de la arquitectura de Microservicios



Autoscaling

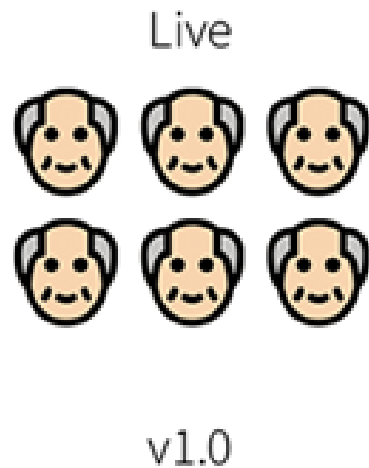


Beneficios de la arquitectura de Microservicios

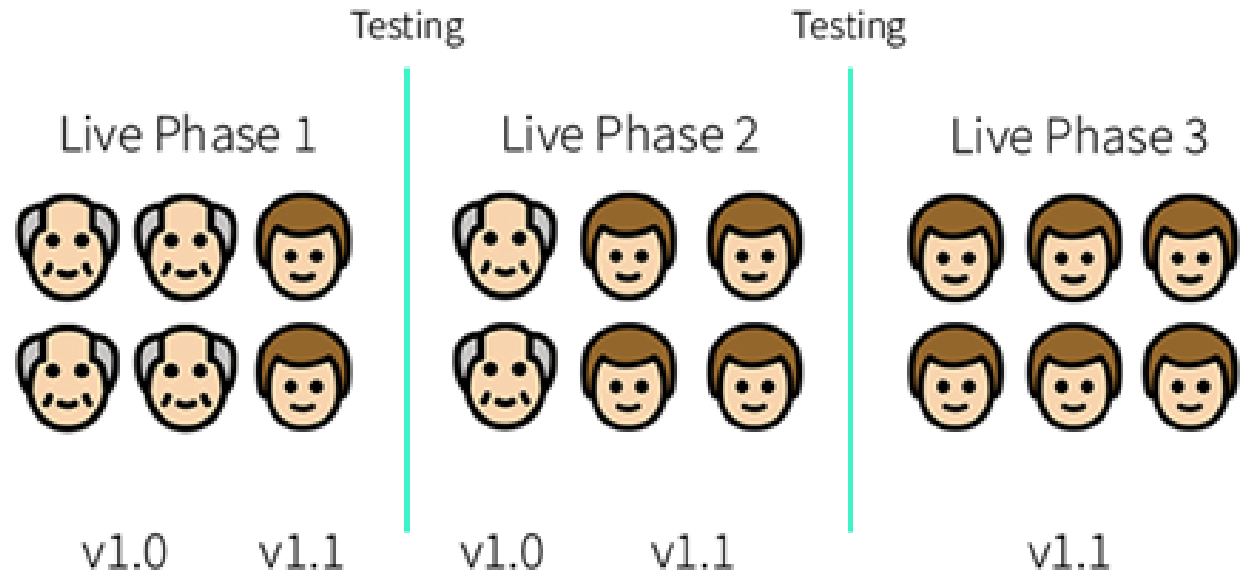


Canary

Before

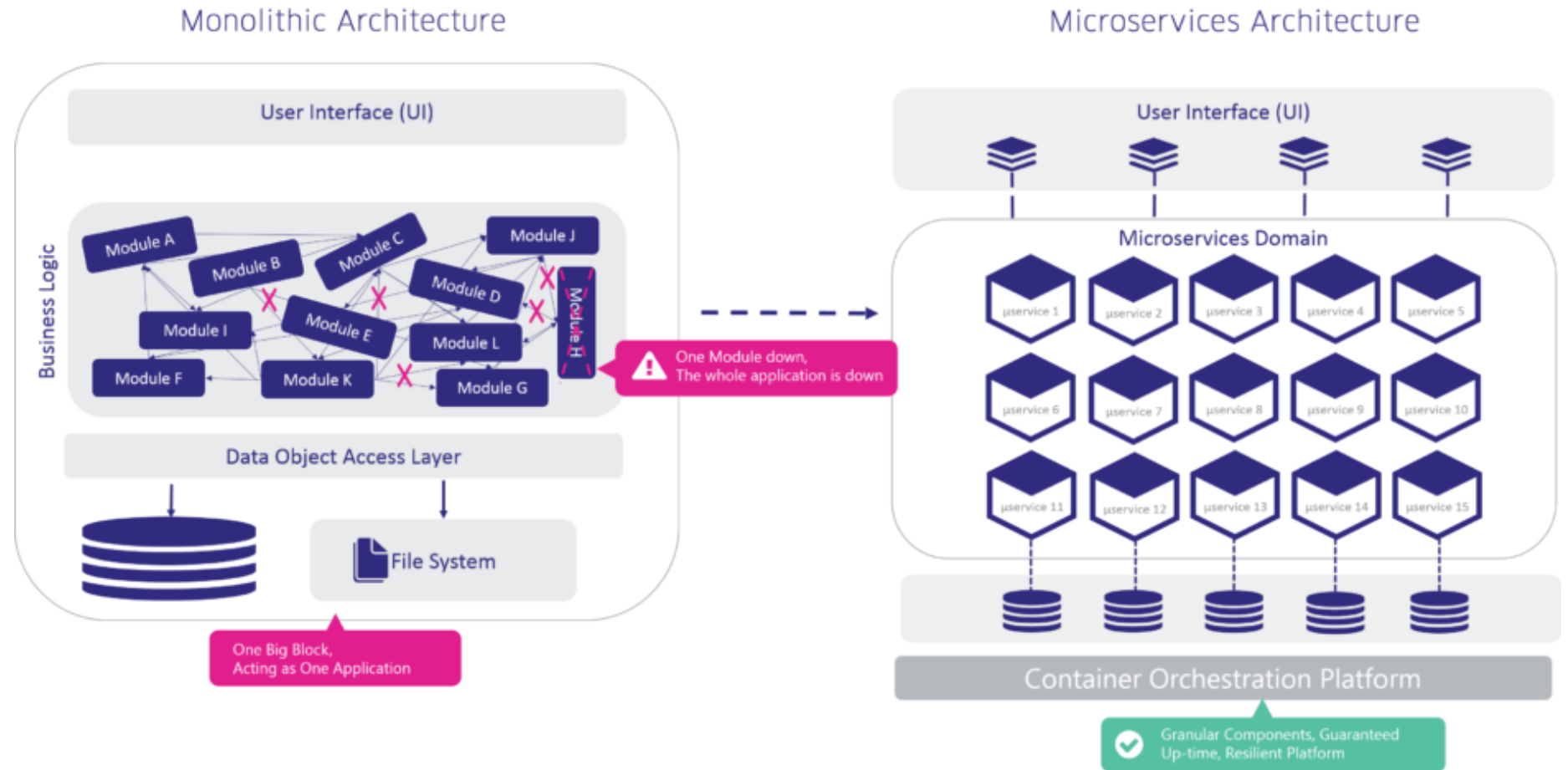


After



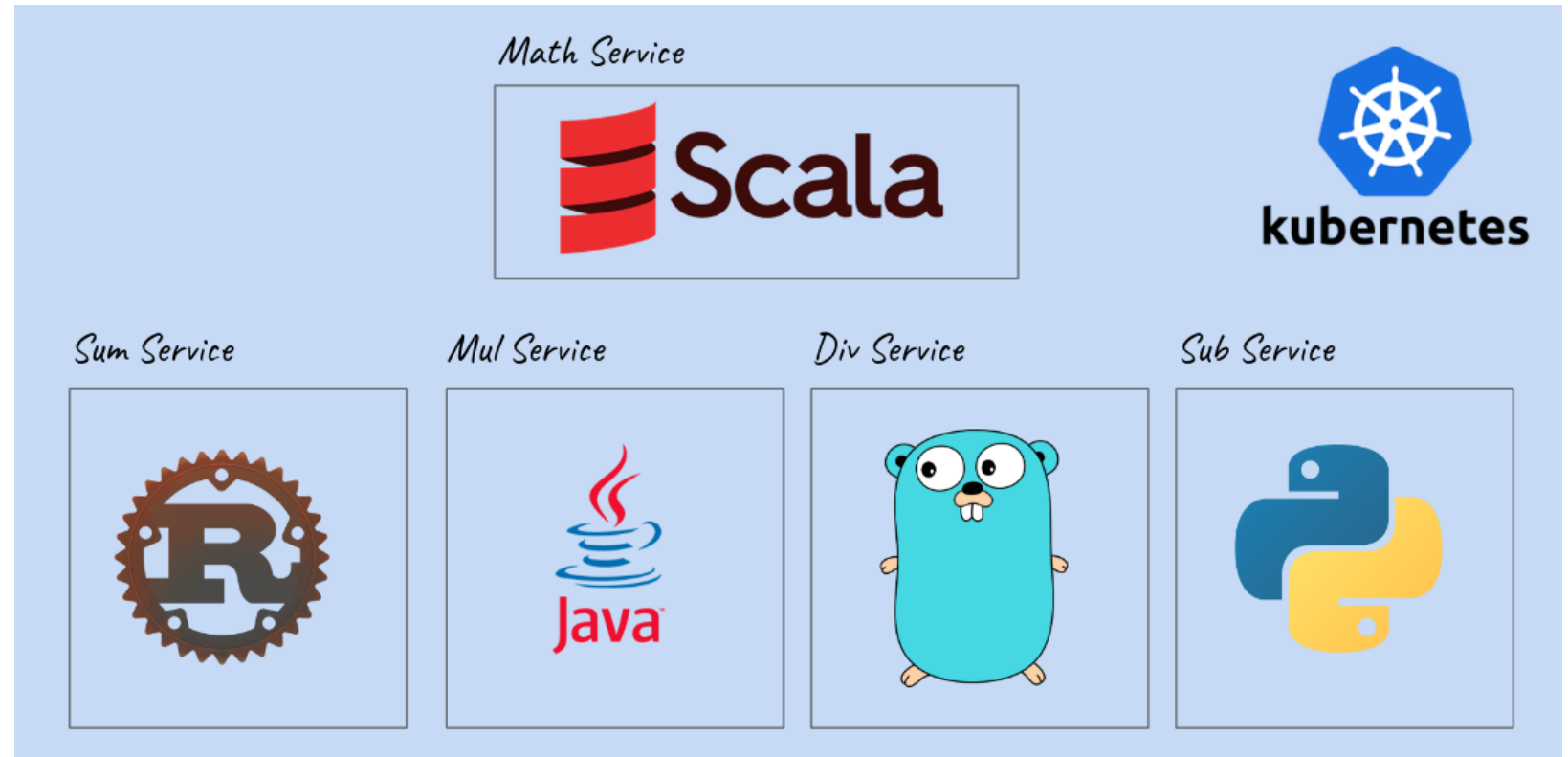
Beneficios de la arquitectura de Microservicios

Resiliencia



Beneficios de la arquitectura de Microservicios

Flexibilidad
Tecnológica



Complejidad

Those “microservices” may be simple individually, but the system itself is a complexity hell!

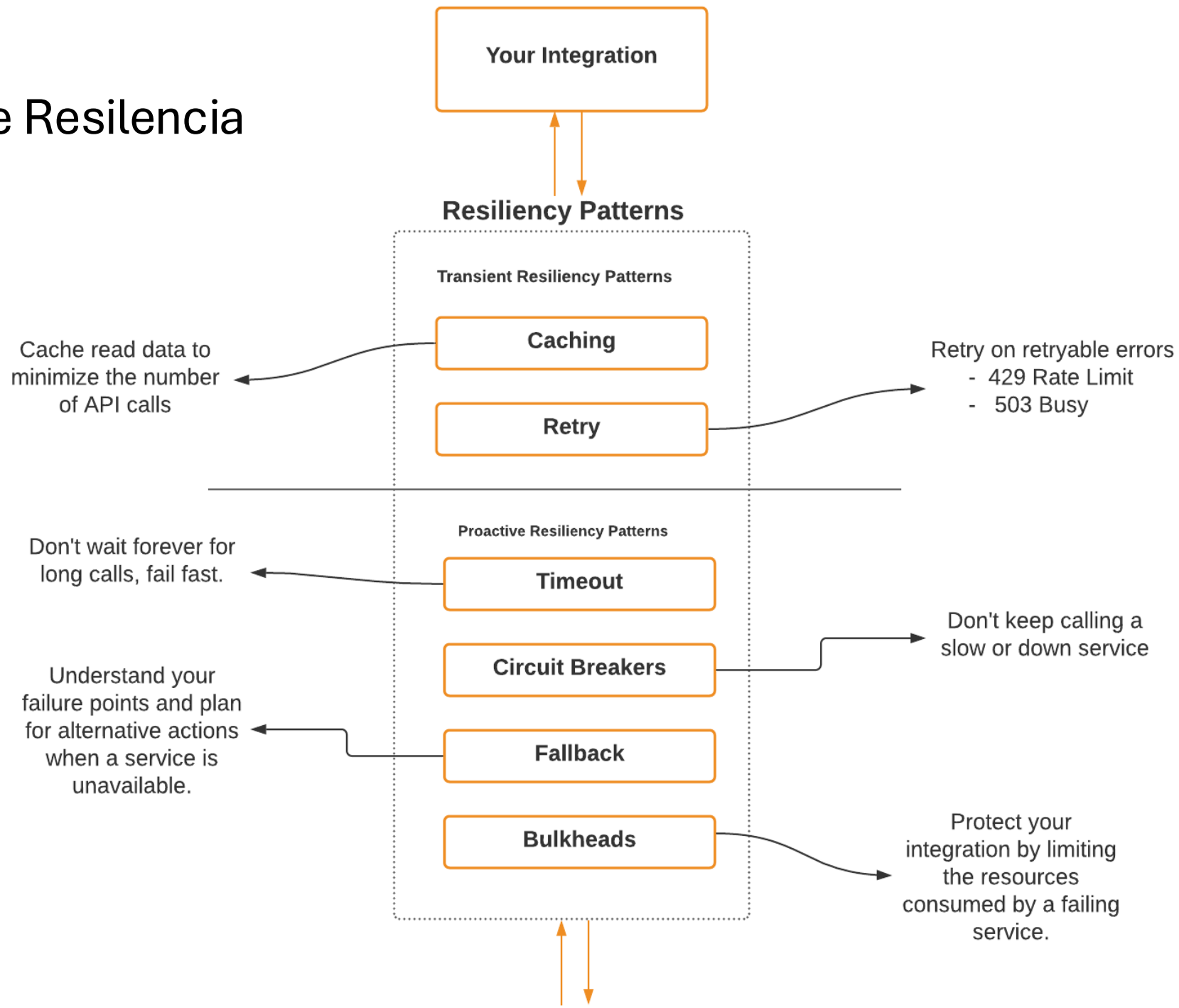
Integración por eventos (Asíncrona) es más difícil de controlar y monitorear que por procesos (Síncrona)

(¿estás seguro que llegó el evento? ¿Dónde se perdió?)

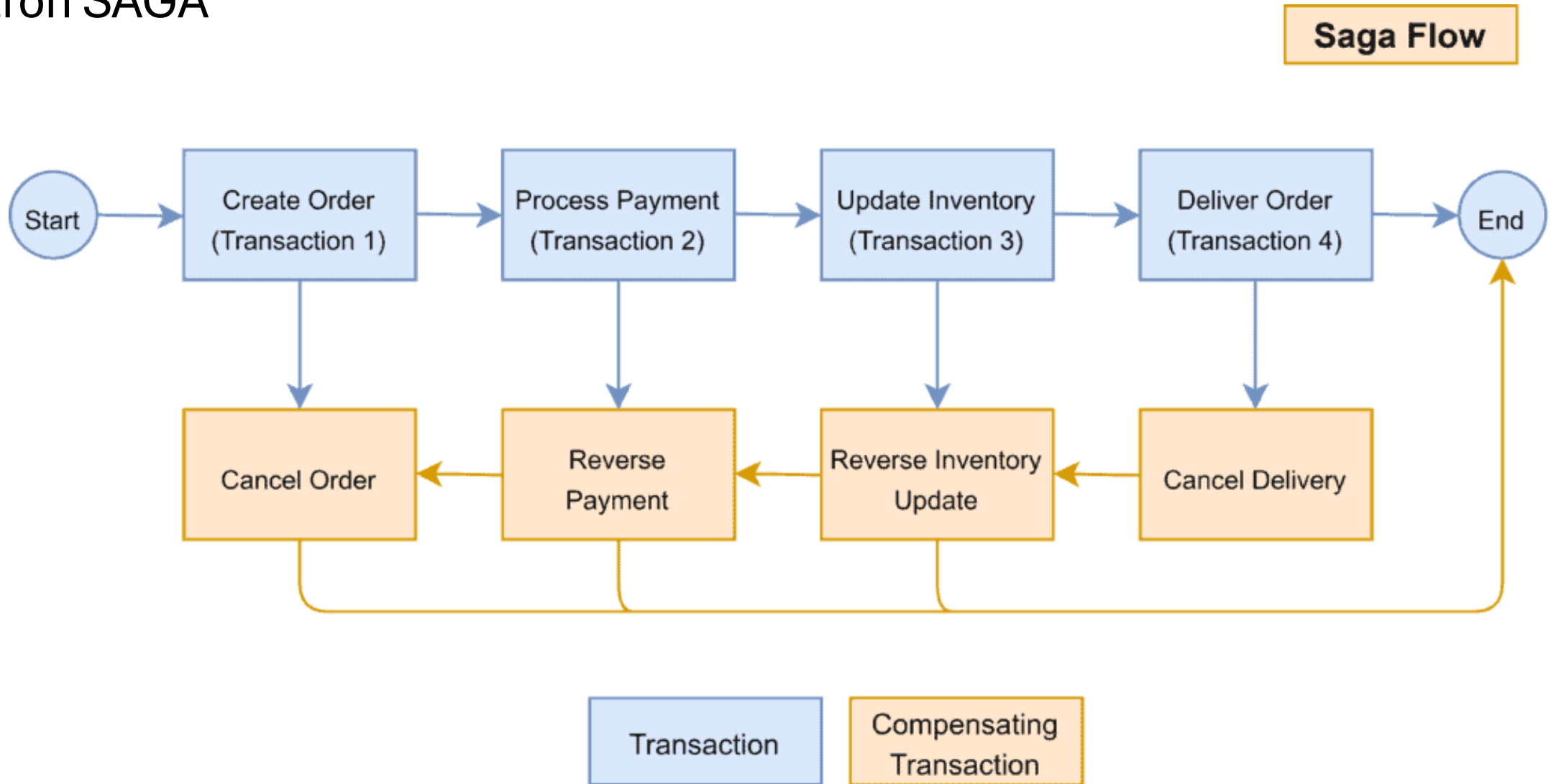
Despliegue de Servicios es un gran desafío por sí solo



Patrones de Resiliencia

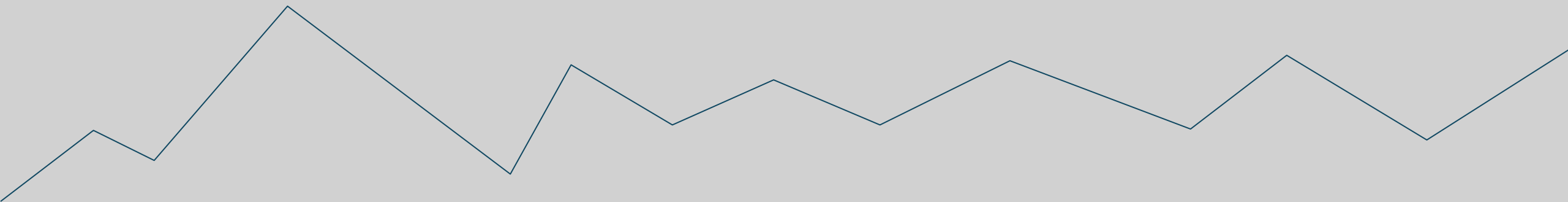


Patrón SAGA



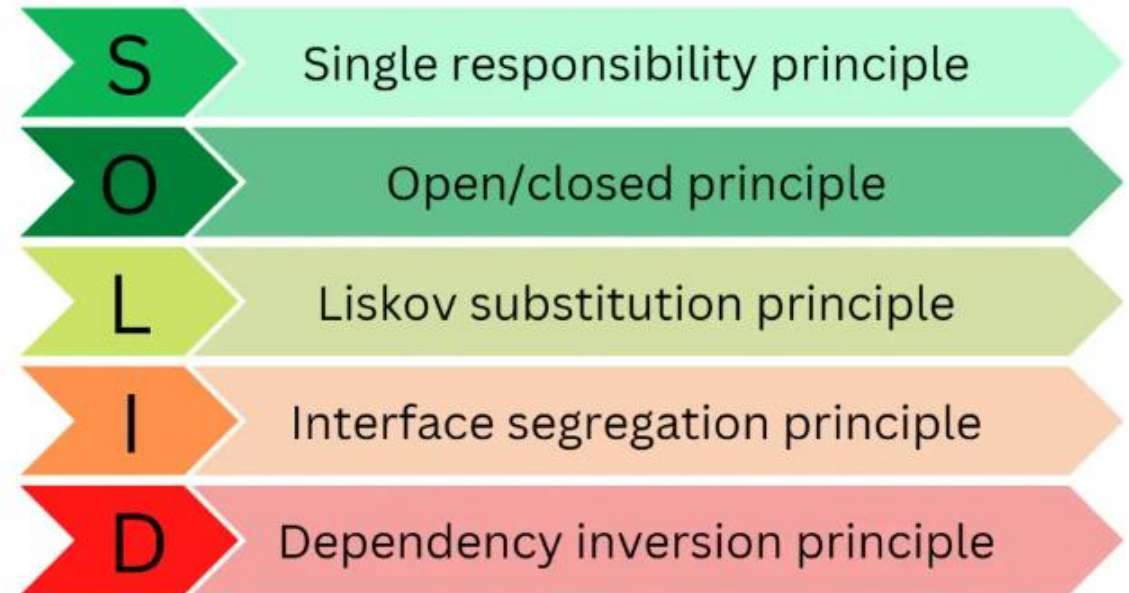
Principios SOLID

Mejores Prácticas para un Diseño de Código Eficiente



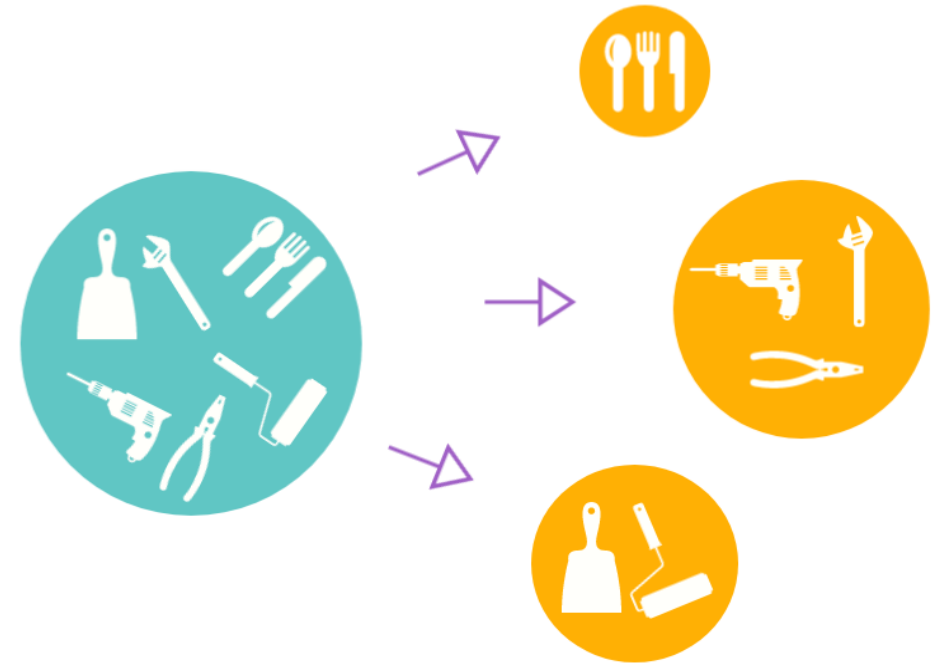
Principios SOLID

- **Definición:** Conjunto de cinco principios de diseño de software destinados a mejorar la mantenibilidad y flexibilidad del código.
- **Objetivo:** Facilitar el desarrollo de software robusto, escalable y fácil de mantener.



1. Principio de Responsabilidad Única (SRP)

- **Definición:** Una clase debe tener una, y solo una, razón para cambiar.
- **Propósito:** Garantizar que una clase tenga una única responsabilidad o propósito.
- **Ejemplo:**
 - **Clase Incorrecta:** ClaseUsuario que maneja datos de usuario y también realiza validación de entrada.
 - **Clase Correcta:** Separar en ClaseDatosUsuario y ClaseValidacionUsuario.



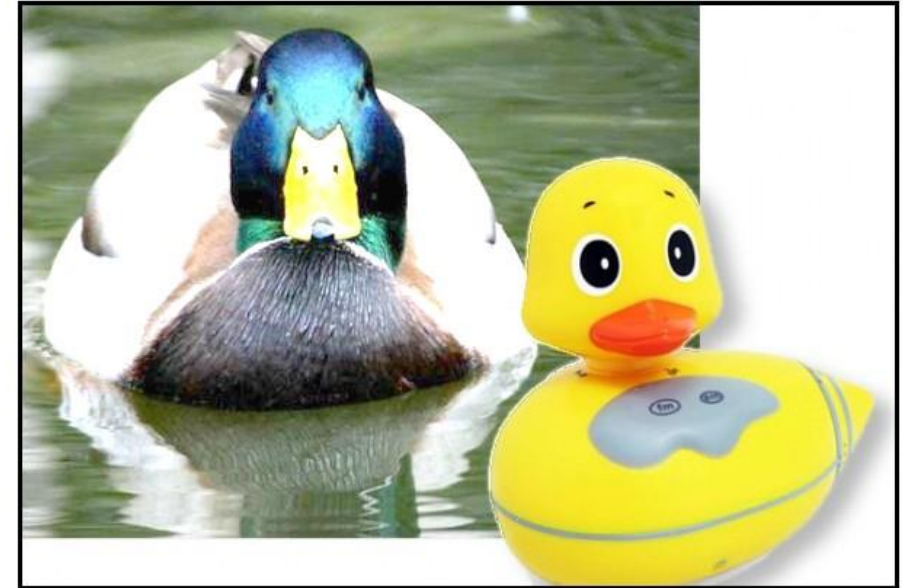
2. Principio de Abierto/Cerrado (OCP)

- **Definición:** Las entidades de software (clases, módulos, funciones, etc.) deben estar abiertas para la extensión pero cerradas para la modificación.
- **Propósito:** Permitir que el comportamiento de una entidad se extienda sin modificar su código fuente.
- **Ejemplo:**
 - ❑ Uso de interfaces y herencia para agregar nuevas funcionalidades sin modificar el código existente.



3. Principio de Sustitución de Liskov (LSP)

- **Definición:** Los objetos de una clase derivada deben ser sustituibles por objetos de la clase base sin alterar el funcionamiento del programa.
- **Propósito:** Asegurar que una clase derivada pueda reemplazar a su clase base sin causar errores.
- **Ejemplo:**
 - ☐ Clases derivadas deben respetar el contrato de la clase base y no violar sus expectativas.



LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

Ejemplo LSP: Abstracción Incorrecta

```
public class PatoConBaterias extends Pato {  
    private boolean bateriaAgotada = false;  
  
    @Override  
    public void nadar() {  
        if (!bateriaAgotada) {  
            System.out.println("El pato con baterías está nadando");  
        } else {  
            System.out.println("La batería está agotada, el pato no  
puede nadar");  
        }  
    }  
  
    public void recargarBateria() {  
        bateriaAgotada = false;  
    }  
}
```

```
public class Pato {  
    public void nadar() {  
        System.out.println("El pato está nadando");  
    }  
}
```

- Un Pato puede nadar siempre.
- Un **PatoConBaterias** puede no nadar si la batería está agotada.
- El comportamiento de **nadar()** no es consistente entre la clase base y la clase derivada.
- **Consecuencias:**
- Esto viola el principio LSP porque **PatoConBaterias** no puede sustituir a **Pato** sin alterar el comportamiento esperado.

Ejemplo LSP: Abstracción Correcta

```
public class PatoConBaterias implements Nadador {
    private boolean bateriaAgotada = false;

    @Override
    public void nadar() {
        if (!bateriaAgotada) {
            System.out.println("El pato con baterías está nadando");
        } else {
            System.out.println("La batería está agotada, el pato no puede nadar");
        }
    }

    public void recargarBateria() {
        bateriaAgotada = false;
    }
}
```

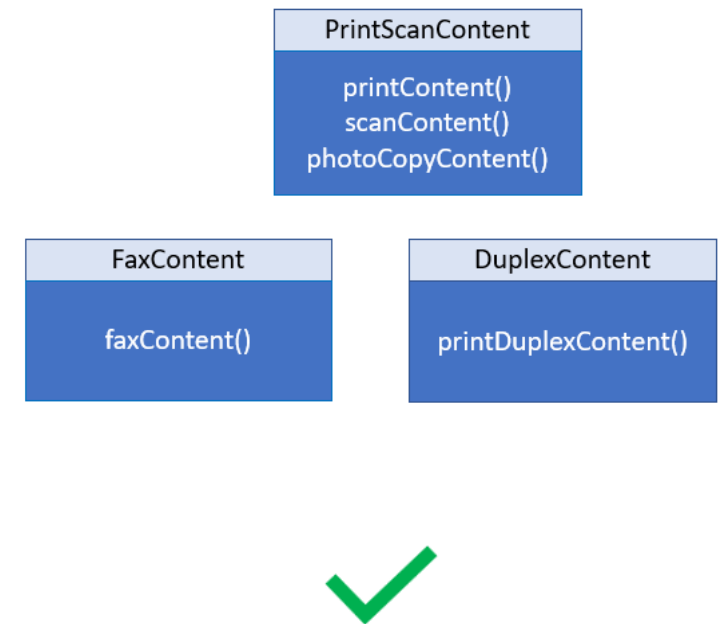
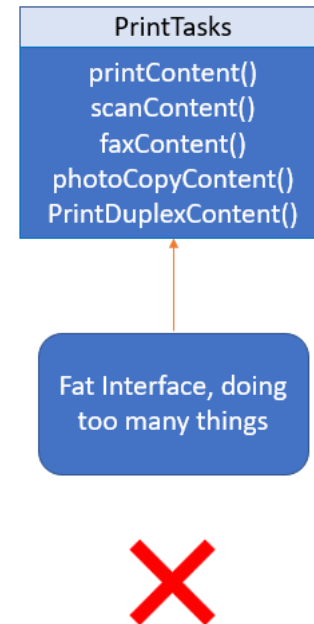
```
public class Pato implements Nadador {
    @Override
    public void nadar() {
        System.out.println("El pato está nadando");
    }
}
```

```
public interface Nadador {
    void nadar();
}
```

- Ambos **Pato** y **PatoConBaterias** implementan la interfaz **Nadador**, asegurando un comportamiento consistente.
- Permite agregar nuevos tipos de nadadores sin cambiar el código existente.
- Promueve un diseño más limpio y desacoplado, facilitando pruebas y mantenimiento.

4. Principio de Segregación de Interfaces (ISP)

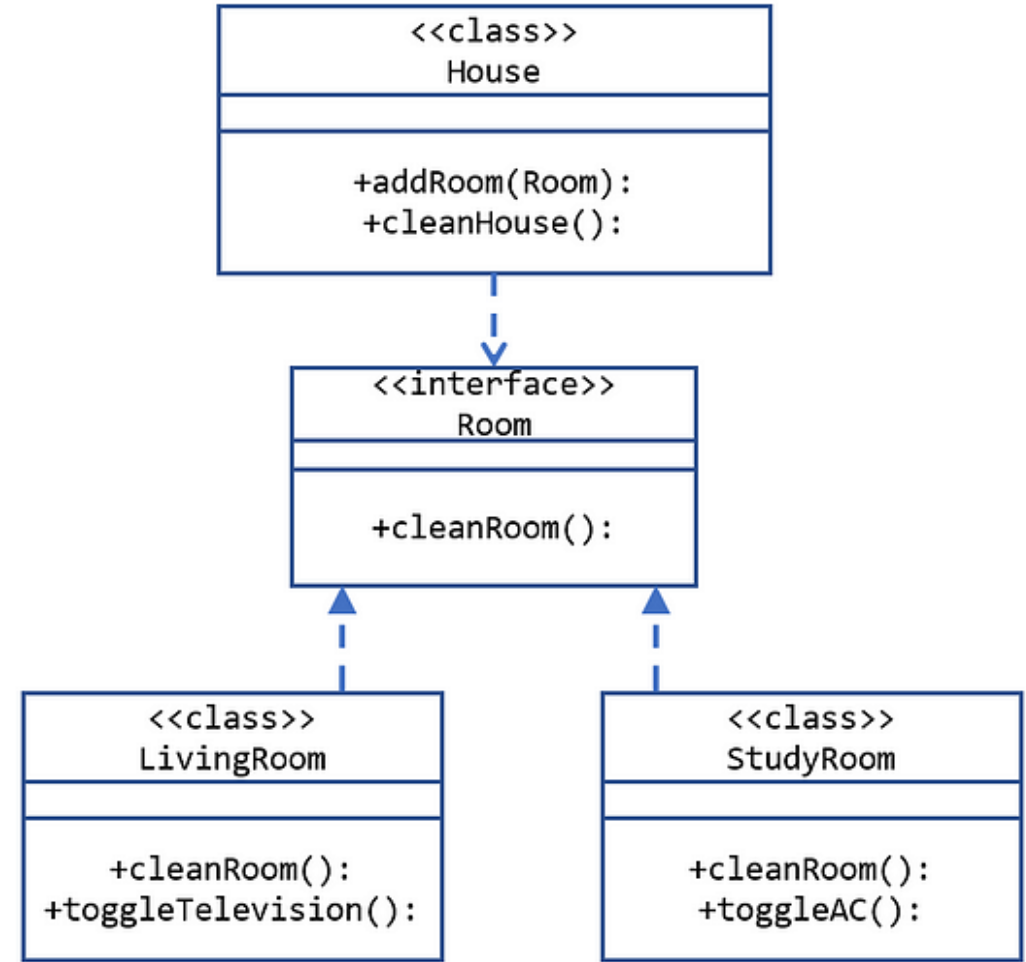
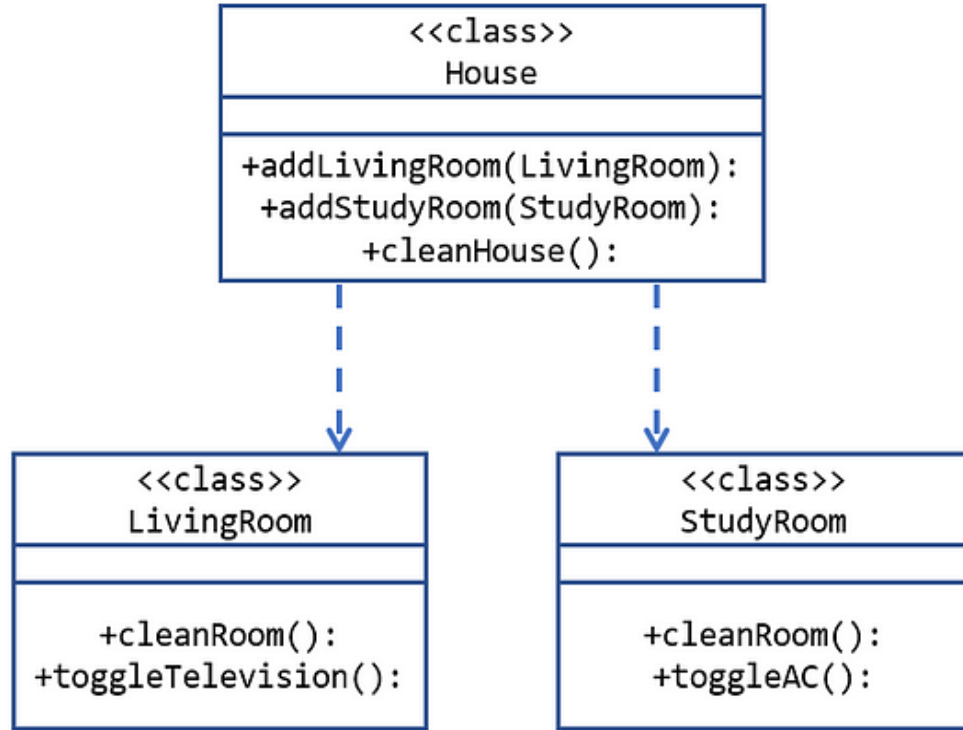
- **Definición:** Los clientes no deben estar forzados a depender de interfaces que no utilizan.
- **Propósito:** Crear interfaces específicas para cada conjunto de clientes en lugar de una interfaz general para todos.
- **Ejemplo:**
 - Dividir una interfaz grande en varias interfaces más pequeñas y específicas.

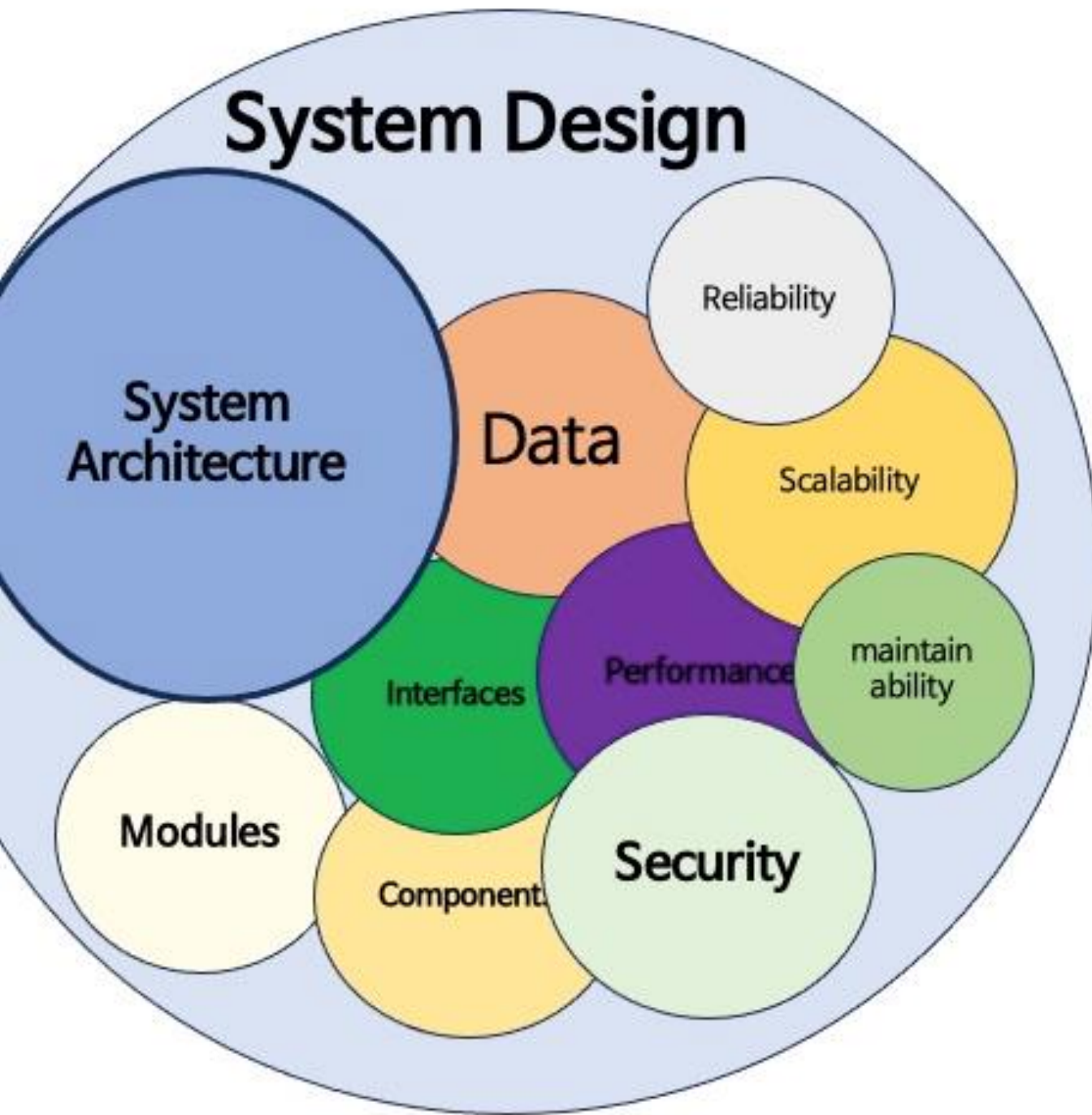


5. Principio de Inversión de Dependencias (DIP)

- **Definición:** Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones.
- **Propósito:** Aislar módulos y permitir que los detalles dependan de las abstracciones.
- **Ejemplo:**
 - Uso de interfaces o abstracciones para desacoplar clases concretas.

5. Principio de Inversión de Dependencias (DIP)





Beneficios de Aplicar SOLID en el diseño de sistemas

- **Mantenibilidad:** Código más fácil de leer, entender y modificar.
- **Flexibilidad:** Facilita la extensión de funcionalidades sin afectar el código existente.
- **Reusabilidad:** Promueve la creación de componentes reutilizables.
- **Testabilidad:** Simplifica la escritura de pruebas unitarias y de integración.



PONTIFICIA
UNIVERSIDAD
CATÓLICA
DE CHILE

IIC3103

Taller de Integración

Profesores

Arturo Tagle / Daniel Darritchon

BIBLIOGRAFÍA

Introducción y Fundamentos

Libro: *Software Engineering* de Ian Sommerville

Artículo: ["The Importance of Software Design"](#), Geeks4Geeks

Artículo: ["What is system design and why it is necessary?"](#), SegWiz

Artículo: [16 System Designs Concepts I wish I Knew](#), Medium



BIBLIOGRAFÍA

Domain-Driven Design (DDD)

Libro: [*Domain-Driven Design: Tackling Complexity in the Heart of Software*](#) de Eric Evans

Libro: [*"Domain-Driven Design Quickly"*](#) en InfoQ

Video: [Introduction to Domain-Driven Design](#) por Eric Evans en YouTube

Artículo: ["Domain Driven Design \(DDD\): Core concepts and Enterprise Architecture"](#) por Alok Mishra

Artículo: ["Domain-driven design practice — Modelling the payments system"](#) en Medium

Artículo: ["The Concept of Domain-Driven Design Explained"](#) en Medium



BIBLIOGRAFÍA

Arquitectura Hexagonal (Ports and Adapters)

Artículo: ["Hexagonal Architecture"](#) por Alistair Cockburn

Libro: *Clean Architecture: A Craftsman's Guide to Software Structure and Design* de Robert C. Martin

Video: [Alistair in the Hexagone](#) en YouTube

Artículo: ["Why Hexagonal Architecture doesn't suit me"](#) en Medium



BIBLIOGRAFÍA

Arquitectura de Microservicios

Libro: *Building Microservices* de Sam Newman

Artículo: "[Microservices - a definition of this new architectural term](#)" por Martin Fowler

Video: [What Are Microservices Really All About](#) en YouTube

Código de ejemplo: [calc-services](#) de Diego Pacheco

Artículo: [Beneficios de AWS AutoScaling](#) de AWS

Artículo: "[Untangling Microservices](#)" de Vladikk

Artículo: "[Building Resiliency into your Cloud Integration Patterns](#)" de Genesys



BIBLIOGRAFÍA

Principios SOLID

Libro: *Agile Principles, Patterns, and Practices in C#* de Robert C. Martin y Micah Martin

Artículo: ["What are the SOLID Principles in Software Engineering"](#) en Digital Ocean

Video: [SOLID Principles in 8 Minutes](#) en YouTube

Artículo: ["Mastering SOLID Principles"](#) en InRythm

Artículo: ["What is Dependency Injection"](#) en Medium

