Análisis y Diseño de Aplicaciones 2

Facultad de Ingeniería

UT4 - TFU

Estudiantes

Agustín Schlechter

Diego Vázquez

Leandro Barral

Andrés Dandrau

Iván Lomando

Profesores

Ángel Mamberto

Esteban Roballo

Intro

Selección de 7 patrones exactamente: 2 de disponibilidad (CB, Health), 2 de rendimiento (QBLL, CQRS/MV), 2 de seguridad (Gatekeeper, Gateway Offloading + OIDC/Federated Identity), 1 de mod/despliegue (External Config Store). Todo alineado con la unidad.

Para la parte 1 cada patrón se presenta con su diagrama de despliegue y diagrama de secuencia + explicación + medición empírica, y la parte 2 es una app prueba de concepto para demostrar estos patrones.

Repo

https://github.com/diegolagreca/UT4TFU

Parte 1 - Diseño

Disponibilidad (Circuit Breaker + Health Endpoint Monitoring)

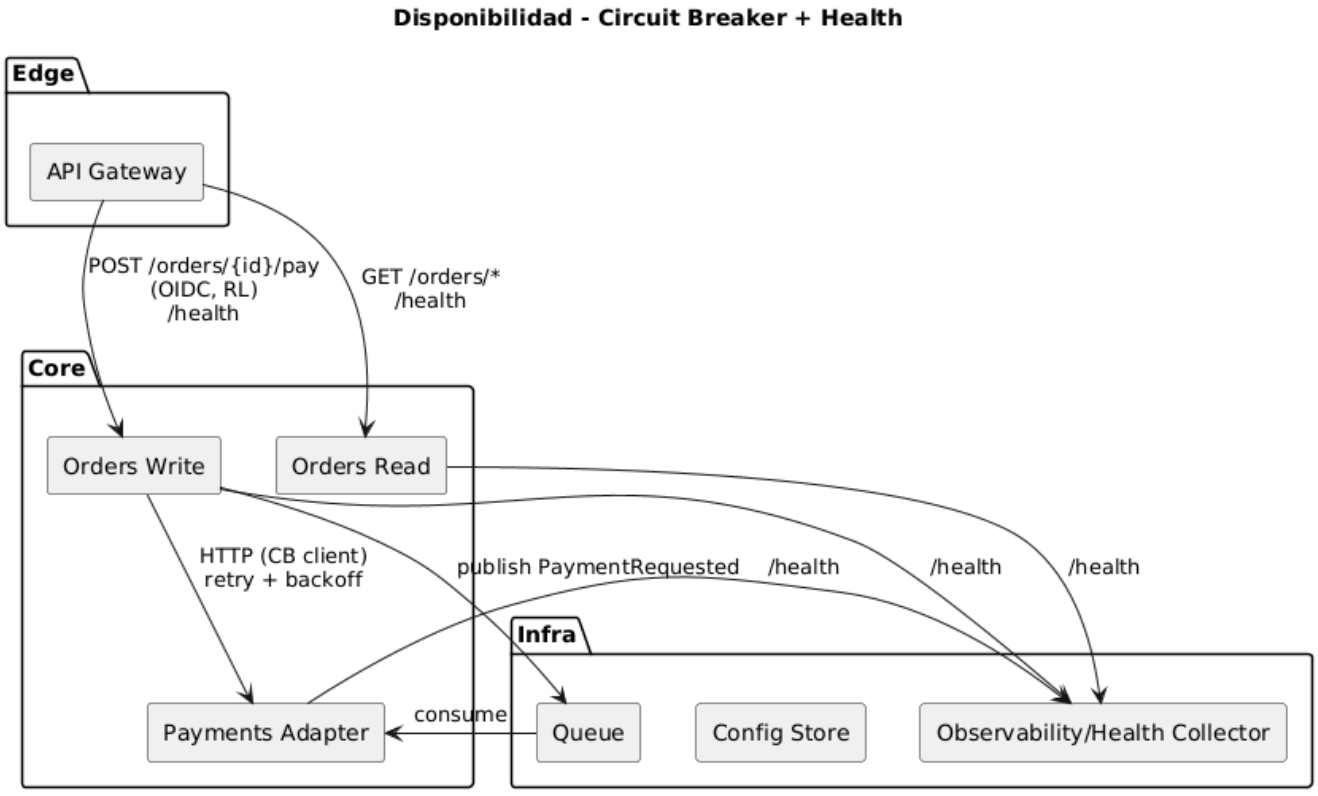
Componentes

- Circuit Breaker (CB) entre orders-write ↔ payments-adapter para evitar cascadas ante fallos; estados Closed → Open → Half-Open con fallback/degradación. (Defensa ante fallos y aislamiento del problema).
- Health Endpoint Monitoring en todos los servicios para que el orquestador/ELB y el gateway detecten y aíslen instancias malas.

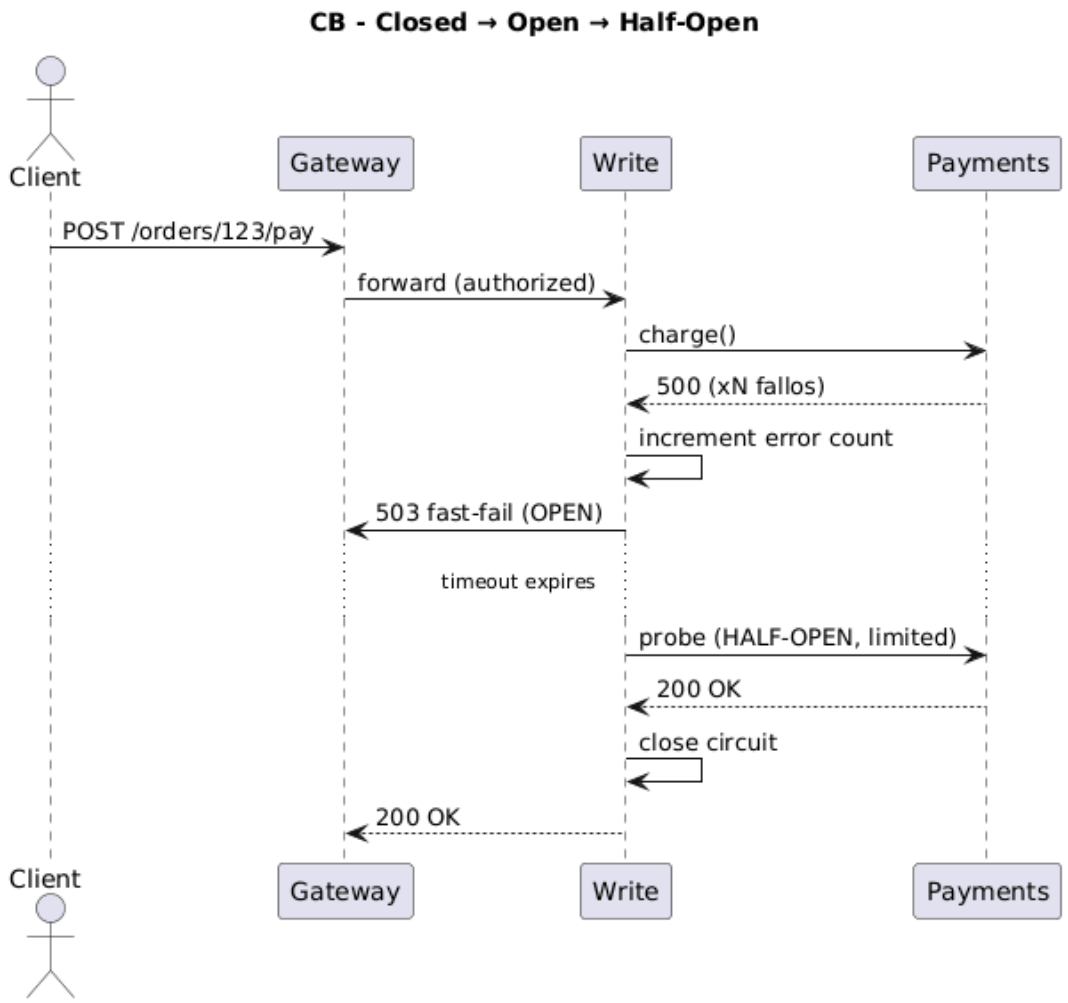
Implementación

- CB reduce el tiempo en que el usuario queda esperando un backend roto; “corta” y responde predecible, protegiendo recursos sanos.
- Health endpoints permiten detectar degradación/caídas y retirar nodos del pool.

UML CB + Health Monitor



Secuencia Circuit Breaker



Medición

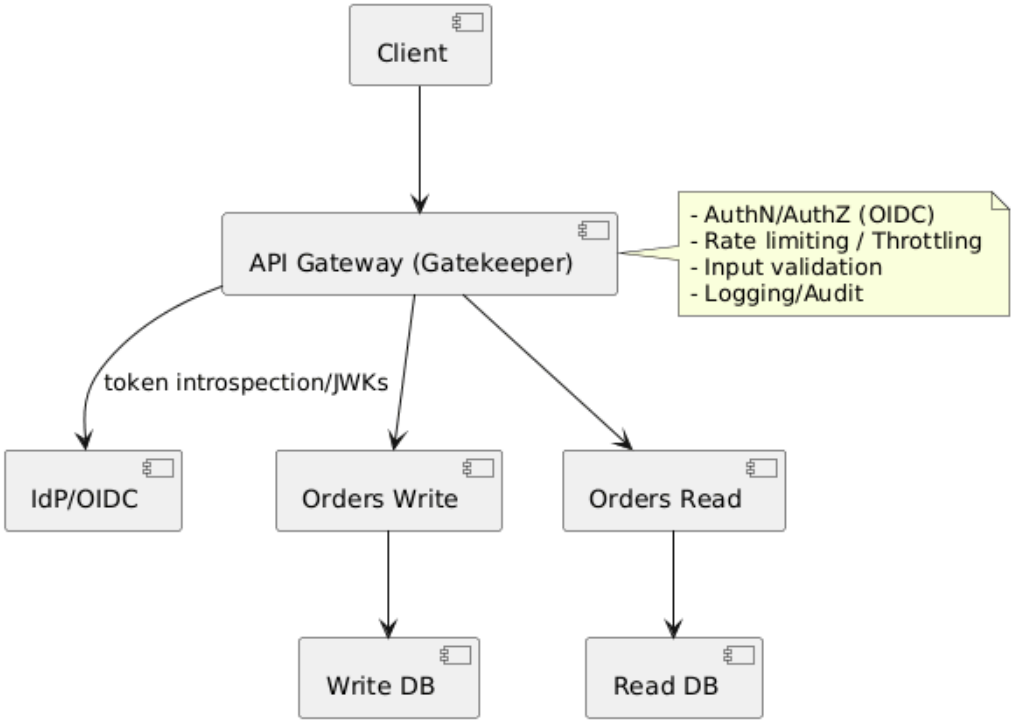
CB: simular 5xx sostenidos en payments-adapter y verificar:

1.1. tiempo hasta “open”

1.2. proporción de fast-fails vs intentos reales

1.3. éxito de sondas “half-open”.

Seguridad - Gatekeeper + OIDC Offloading



Rendimiento (Queue-Based Load Leveling + CQRS/Materialized View)

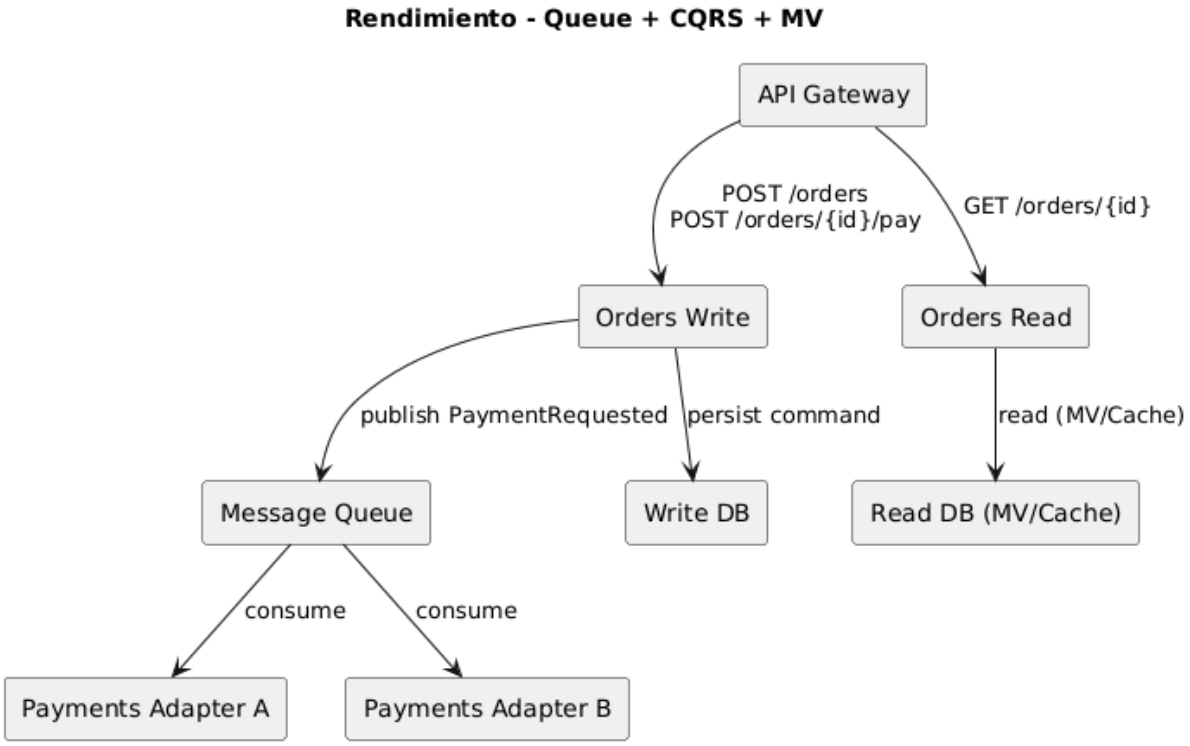
Componentes

- Queue-Based Load Leveling (QBLL) desacopla picos: orders-write encola pagos y payments-adapter los procesa estable; escalás con Competing Consumers.
- CQRS + Materialized View separa escritura/lectura y sirve lecturas rápidas desde vistas precalculadas (consistencia eventual).

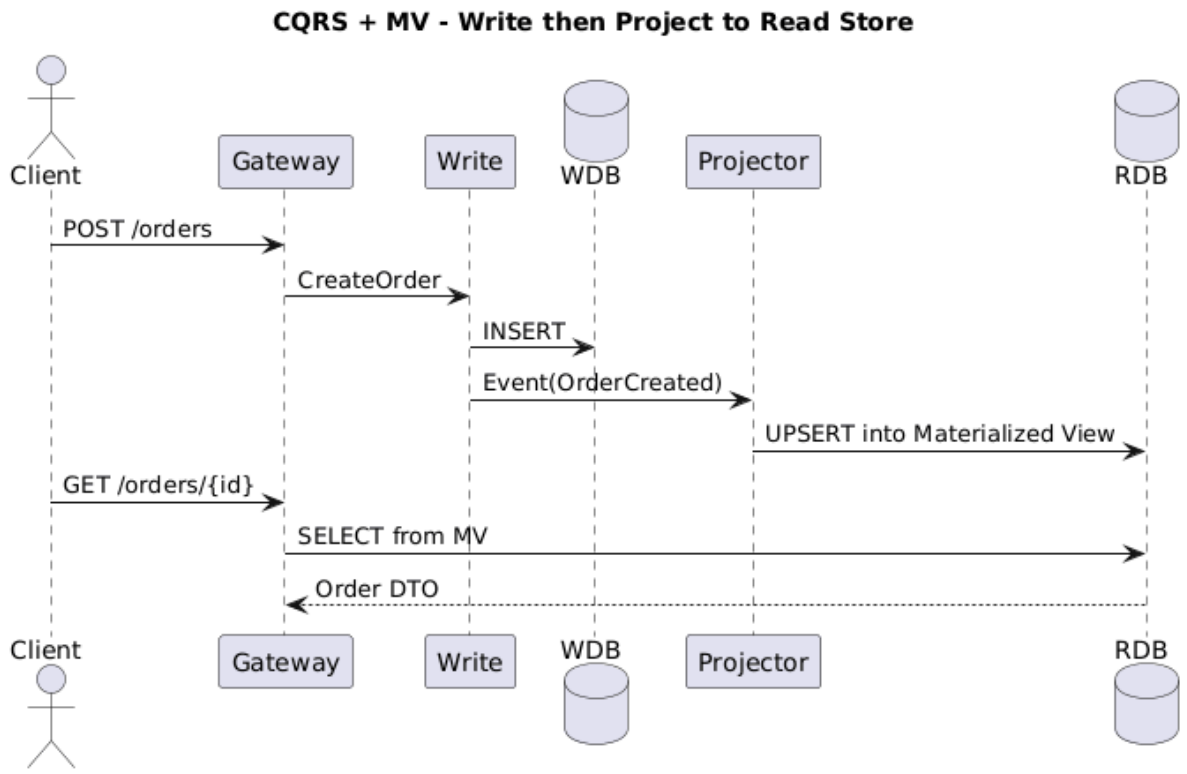
Implementación

- QBLL absorbe ráfagas, mantiene baja la latencia del endpoint productor y evita saturación de servicios críticos.
- CQRS/MV reduce sobrecarga de consultas complejas; P95/P99 bajan al servir desde vista optimizada.

UML QBLL + Competing Consumers + CQRS



Secuencia - Escritura y proyección a vista



Medición

- QBLL: inyectar 1000 pagos burst; medir latencia del POST (productor) y tiempo de vaciado de la cola; escalar consumidores y comparar throughput.
- CQRS/MV: comparr P95 de GET /orders/{id} contra lectura directa en DB; documentar consistencia.

Seguridad (Gatekeeper + Gateway Offloading + Federated Identity)

Componentes

- Gatekeeper (API Gateway) como punto único de entrada que valida, autentica, autoriza, rate-limitea, registra, y protege servicios internos.
- Gateway Offloading: el gateway descarga autenticación, validación de mensajes, cifrado, logging, etc., para que los microservicios se enfoquen en la lógica de negocio.
- Federated Identity (OIDC): el gateway confía en un IdP (p.ej., Keycloak) y aplica políticas por scopes/roles. (El material lista la familia de patrones y tácticas).

Implementación

¿Cómo se logra el atributo de calidad (Seguridad)?

Gatekeeper: Centraliza políticas de seguridad en un único punto auditado, evitando implementación inconsistente en cada servicio. Reduce superficie de ataque al mantener servicios internos inalcanzables desde internet.

Gateway Offloading: Estandariza controles (validación JWT, rate limiting por cliente, logging de accesos) y permite actualizaciones de seguridad (rotación de certificados, nuevos algoritmos de cifrado) sin modificar servicios.

Federated Identity: Delega autenticación a un IdP especializado (MFA, auditoría de accesos, políticas de contraseñas), reduce riesgo de fuga de credenciales y permite SSO entre aplicaciones.

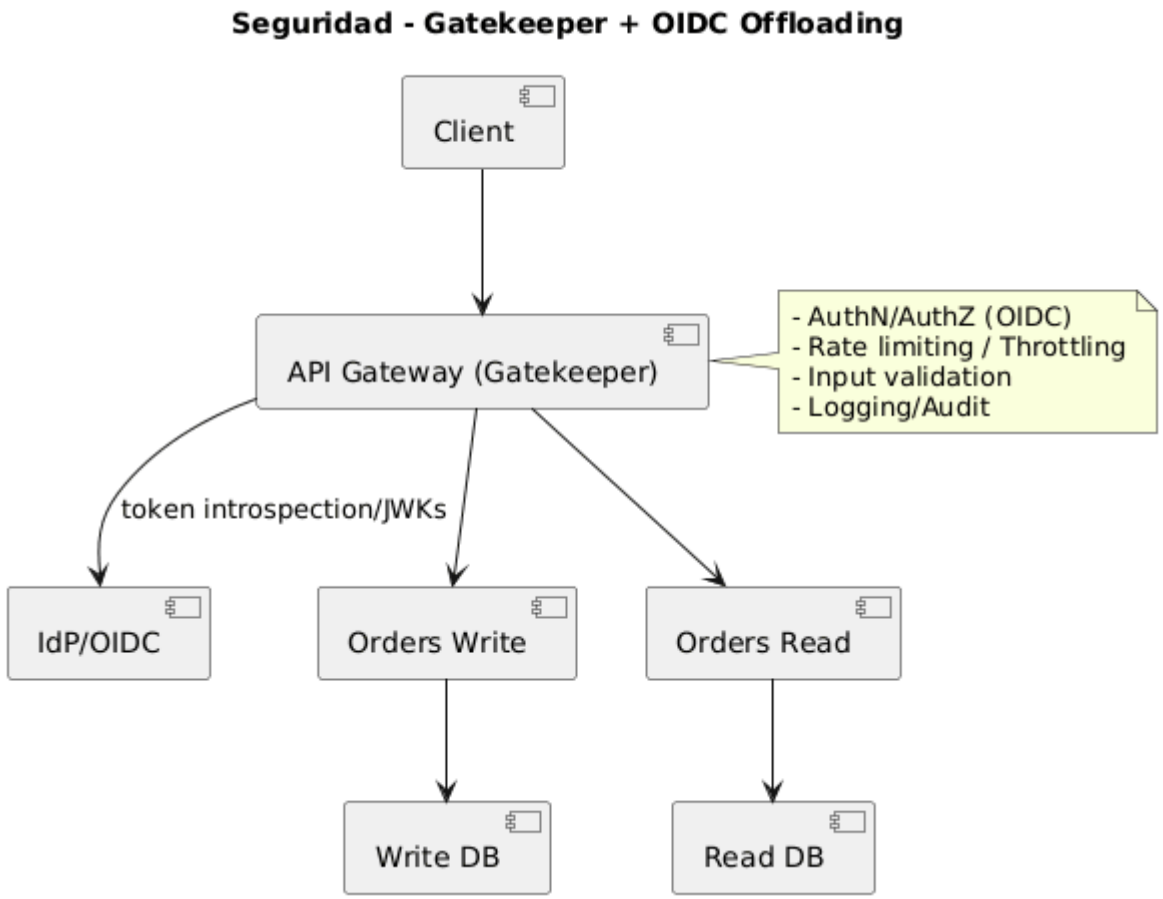
Riesgos mitigados:

Sin Gatekeeper: cada servicio debe validar tokens → inconsistencias, duplicación de código.

Sin Offloading: servicios implementan criptografía ad-hoc → vulnerabilidades (ej. timing attacks).

Sin Federated Identity: credenciales en base de datos propia → mayor superficie de ataque.

UML Gatekeeper + IdP



Medición

- AuthZ: sin token → 401, token inválido → 403, scope insuficiente → 403.
- RL: 100 req/10s → observar 429 a partir del umbral configurado en el gateway.

Facilidad de modificación/despliegue (External Configuration Store)

Componentes:

External Configuration Store:

Sistema de configuración centralizada que almacena parámetros de aplicación de forma versionable y externalizada al código fuente. Incluye feature flags, valores operacionales (ej. PAYMENT_MAX_RETRIES), umbrales de rate limiting y otros parámetros que los servicios consumen dinámicamente en runtime. Este patrón, documentado en el conjunto de patrones de despliegue y modificación de la unidad, permite ajustar comportamientos del sistema sin requerir recompilación o redespliegue de componentes.

Implementación:

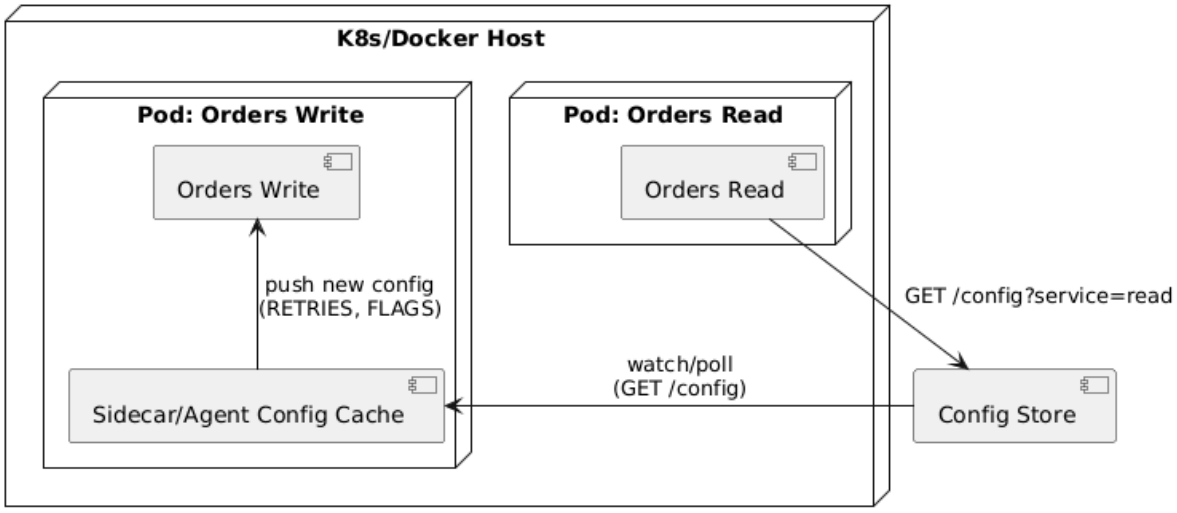
La implementación de este patrón proporciona tres capacidades fundamentales para la facilidad de modificación:

Cambios sin recompilar: mediante feature toggles y configuración dinámica, es posible habilitar/deshabilitar funcionalidades y ajustar parámetros operacionales sin intervenir el código fuente ni las imágenes de contenedores

Rollback rápido: ante configuraciones erróneas o cambios problemáticos, la reversión se realiza actualizando valores en el almacén centralizado, evitando ciclos completos de despliegue

Trazabilidad de cambios: el versionado de configuraciones provee auditoría completa de modificaciones (quién, cuándo, qué cambió), cumpliendo requisitos de governance y facilitando análisis post-incidente

Mod/Despliegue - External Configuration Store



Parte 2 - Prueba de Concepto

Objetivo

Simular un sistema distribuido y tolerante a fallos que permita demostrar patrones de arquitectura aplicados a escenarios reales:

- Alta disponibilidad
- Rendimiento bajo carga
- Desacoplamiento mediante colas
- Recuperación automática ante fallas externas

El entorno representa una arquitectura orientada a eventos (event-driven), con servicios independientes que se comunican mediante HTTP y RabbitMQ.

Componentes principales

Servicio	**Rol**	**Patrones asociados**
API Gateway (Kong)	Entrada unificada. Autenticación, ruteo y rate-limiting.	API Gateway, Rate Limit
Orders-Write	Crea órdenes, ejecuta pagos, publica eventos.	Circuit Breaker, Command/Query Separation
Orders-Read	Mantiene vistas optimizadas para lectura.	CQRS, Cache Aside
Payments-Adapter	Simula un proveedor externo de pagos (controlado con /toggle).	Circuit Breaker, Fault Injection
Projector	Procesa eventos de pago para actualizar proyecciones.	Event-Driven Architecture
RabbitMQ	Broker para comunicación asíncrona.	Queueing, Competing Consumers
Config Store / Redis / Postgres	Servicios de configuración, cache y persistencia.	Config Server, Data Replication

Setup paso a paso

1. **Clonar el proyecto**
   a. `git clone <repo> ut4_tfu_demo`
   b. `cd ut4_tfu_demo`
2. **Limpiar entorno Docker (solo la primera vez)**
   a. `docker stop $(docker ps -aq) 2>/dev/null; docker rm -f $(docker ps -aq) 2>/dev/null`
   b. `docker system prune -a --volumes -f`
3. **Levantar todos los servicios**
   a. `docker-compose up -d --build`
4. **Verificar estado**
   a. `docker-compose ps`

Todos deben aparecer como Up.

Accesos útiles

- RabbitMQ: http://localhost:15672 (usuario: guest / passwd: guest)
- Config Store: http://localhost:8088/config
- API Gateway: http://localhost:8080

Ver logs en vivo

- `docker compose logs -f orders-write payments-adapter`

Clean startup:

```
# 0) limpiar

docker-compose down

docker-compose rm -f

# 1) infra primero

docker-compose up -d queue redis writedb readdb config auth

# 2) microservicios de app

docker-compose up -d --build orders-write orders-read
payments-adapter projector

# 3) gateway al final

docker-compose up -d api-gateway

# 4) checks

docker-compose ps
```

Demos preparadas

Para ejecutar todas las demo juntas, se puede tirar el comando `./demo.sh PART=all` en la raíz del proyecto

Demo 1: Disponibilidad (Circuit Breaker)

Objetivo: mostrar cómo el sistema evita la saturación cuando un servicio externo falla.
Descripción:

- Orders-Write llama al Payments-Adapter, que simula un servicio externo.
- Al forzar fallos (make cb-open), el Circuit Breaker se abre y las llamadas fallan rápido con 503.
- Al restablecer (make cb-close), el sistema prueba gradualmente hasta cerrar el breaker y recuperar el flujo.

Comandos:

```
make cb-open
```

```
make cb-close
```

Aprendizaje: el patrón Circuit Breaker protege la disponibilidad y mejora el tiempo de recuperación.

Demo 2: Rendimiento (Queue + Competing Consumers)

Objetivo: observar cómo RabbitMQ desacopla la carga y permite procesar picos de tráfico.
Descripción:

- Orders-Write publica eventos de pago en la cola payments.
- Payments-Adapter los consume de forma asíncrona.
- Durante un burst de órdenes, la cola amortigua la carga.

Comando:

```
tests/load/burst_payments.sh
```

Visualización: abrir RabbitMQ UI (http://localhost:15672/#/) → Queue payments → observar “Ready / Unacked” subir y bajar.
Aprendizaje: los patrones Queueing y Competing Consumers permiten desacople, paralelismo y control de backpressure.

Demo 3: Seguridad (API Key + Rate Limiting)

Objetivo: verificar que el gateway protege los servicios mediante autenticación y limita el tráfico abusivo.

Descripción:

- Kong valida API keys antes de permitir acceso a los endpoints.
- Sin API key válida, las peticiones son rechazadas con 401.
- Rate limiting configurado: 30 req/min para lecturas, 60 req/min para escrituras.
- Al exceder el límite, Kong responde 429 sin tocar los servicios backend.

Comando:

tests/security/auth_and_rl.sh

Visualización: el script muestra:

1. Petición sin API key → 401 Unauthorized
2. Peticiones válidas → 200 OK
3. Burst de 50 requests → aparecen 429 Too Many Requests al exceder el quota

Aprendizaje: el patrón Gatekeeper centraliza autenticación y rate limiting en un único punto, protegiendo servicios internos de tráfico no autorizado o abusivo.

Demo 4: Facilidad de modificación (External Configuration Store)

Objetivo: cambiar parámetros operacionales sin redesplegar servicios.

Descripción:

- Config Store expone API REST en puerto 8088.
- Servicios consultan configuración al inicio y recargan en runtime.
- Parámetros configurables: `paymentMaxRetries` (default: 3), `cacheTtlSec` (default: 10).

Comando:

```
curl -s http://localhost:8088/config

curl -X POST http://localhost:8088/config -H "Content-Type: application/json" --data '{"paymentMaxRetries":1,"cacheTtlSec":3}'
```

Visualización: el script muestra la config actual, aplica cambios (`paymentMaxRetries=1, cacheTtlSec=3`) y muestra la nueva config sin reiniciar contenedores.

Aprendizaje: el patrón External Configuration Store permite ajustar comportamiento del sistema sin recompilar código, habilitando cambios rápidos y rollbacks en segundos.

Demo integrada

Para ejecutar todas las demos en secuencia se puede usar:

```
./demo.sh PART=all
```

O individualmente:

```
./demo.sh PART=availability

./demo.sh PART=performance

./demo.sh PART=security

./demo.sh PART=config
```