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.

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)

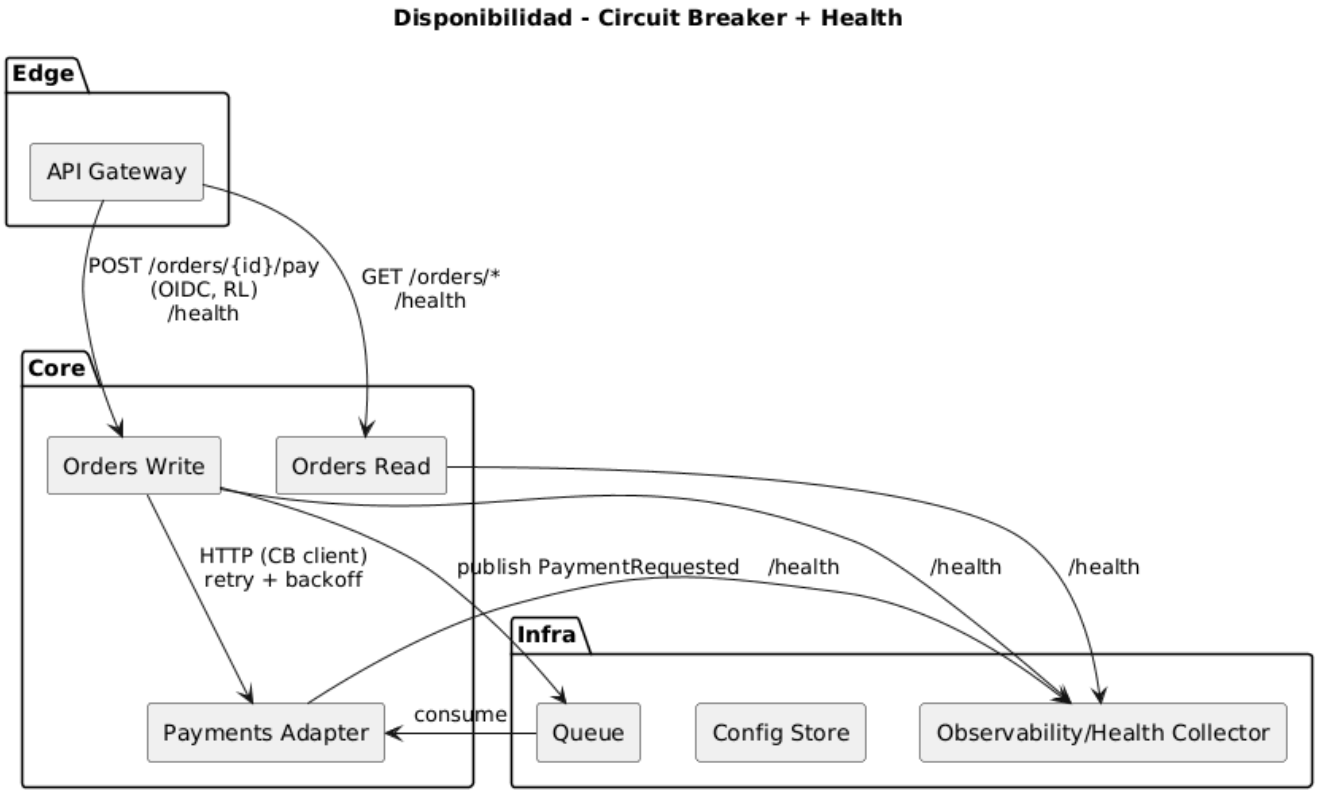
Componentes

- 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.

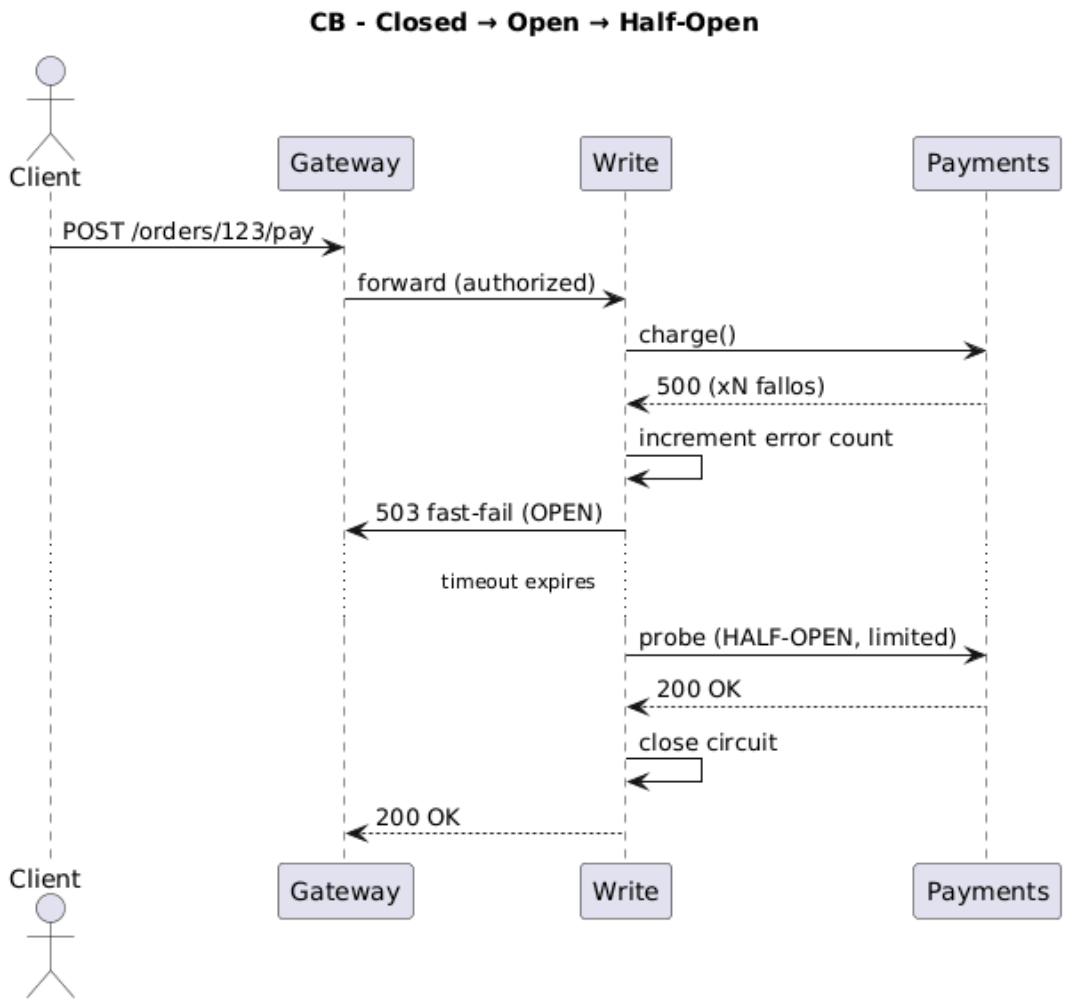
Implementación

- 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

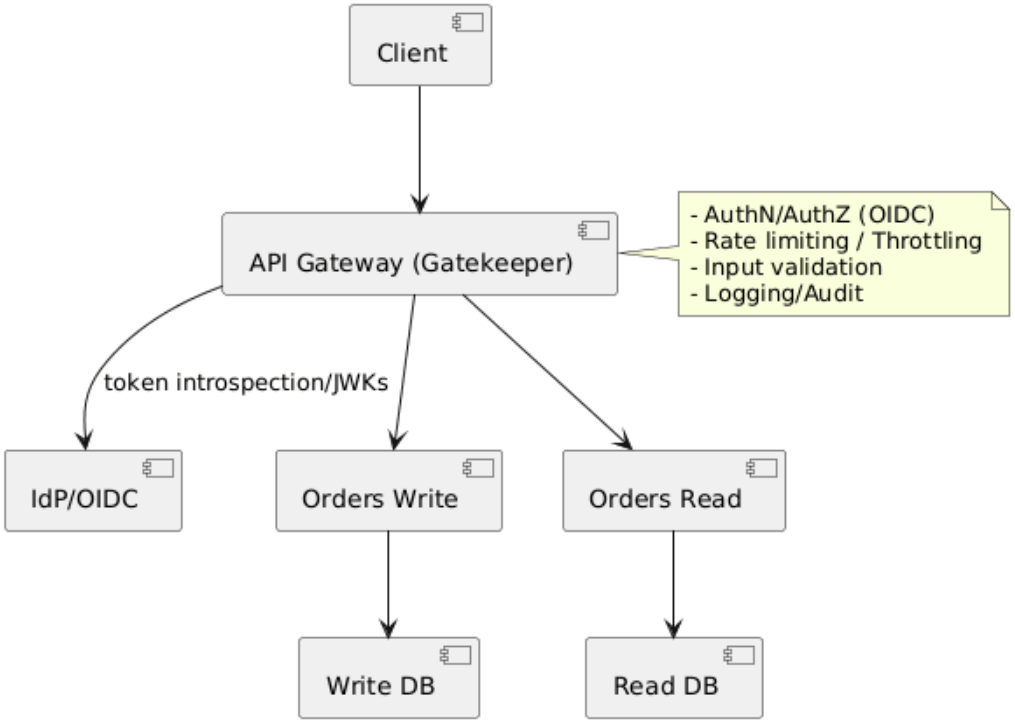
CB: simular 5xx sostenidos en payments-adapter y verificar:

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)

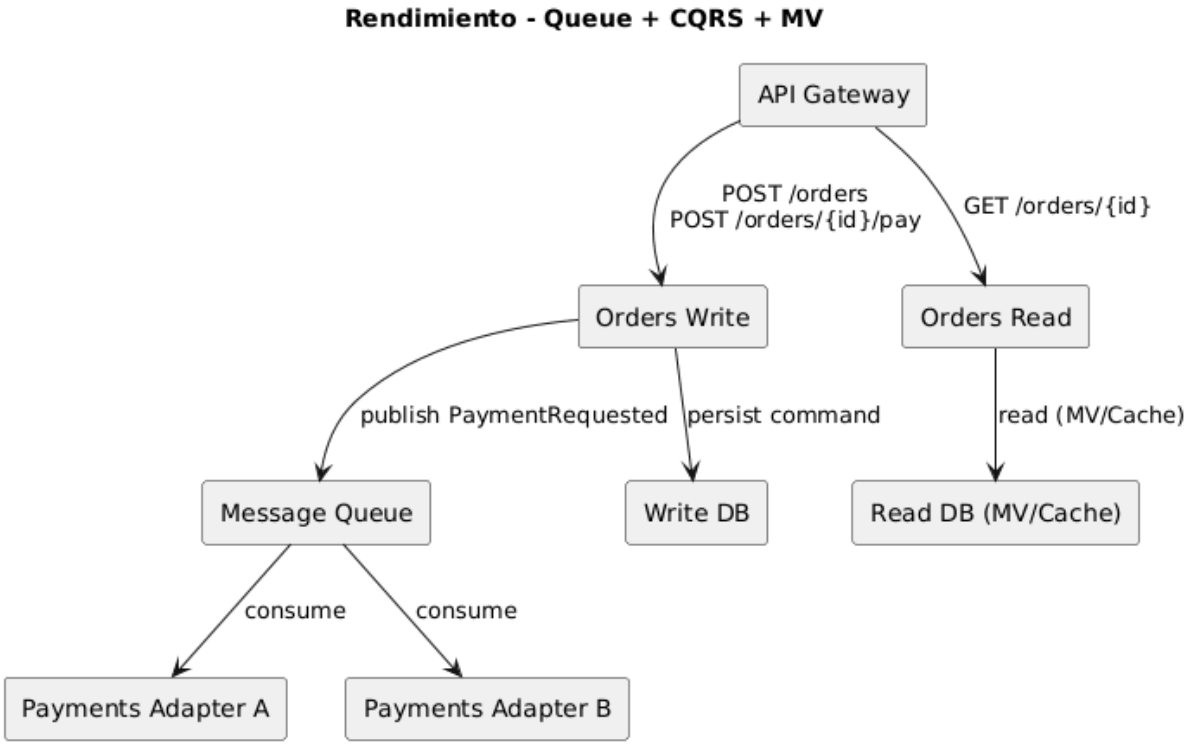
Componentes

- 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).

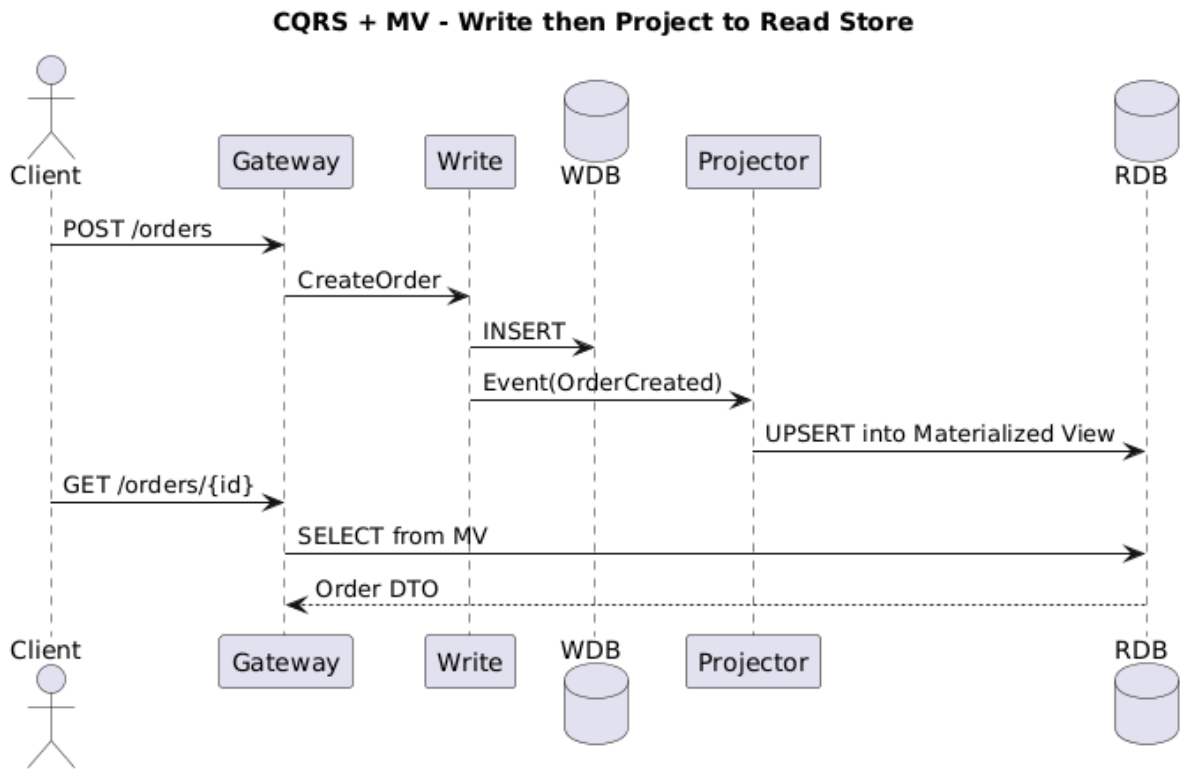
Implementación

- 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)

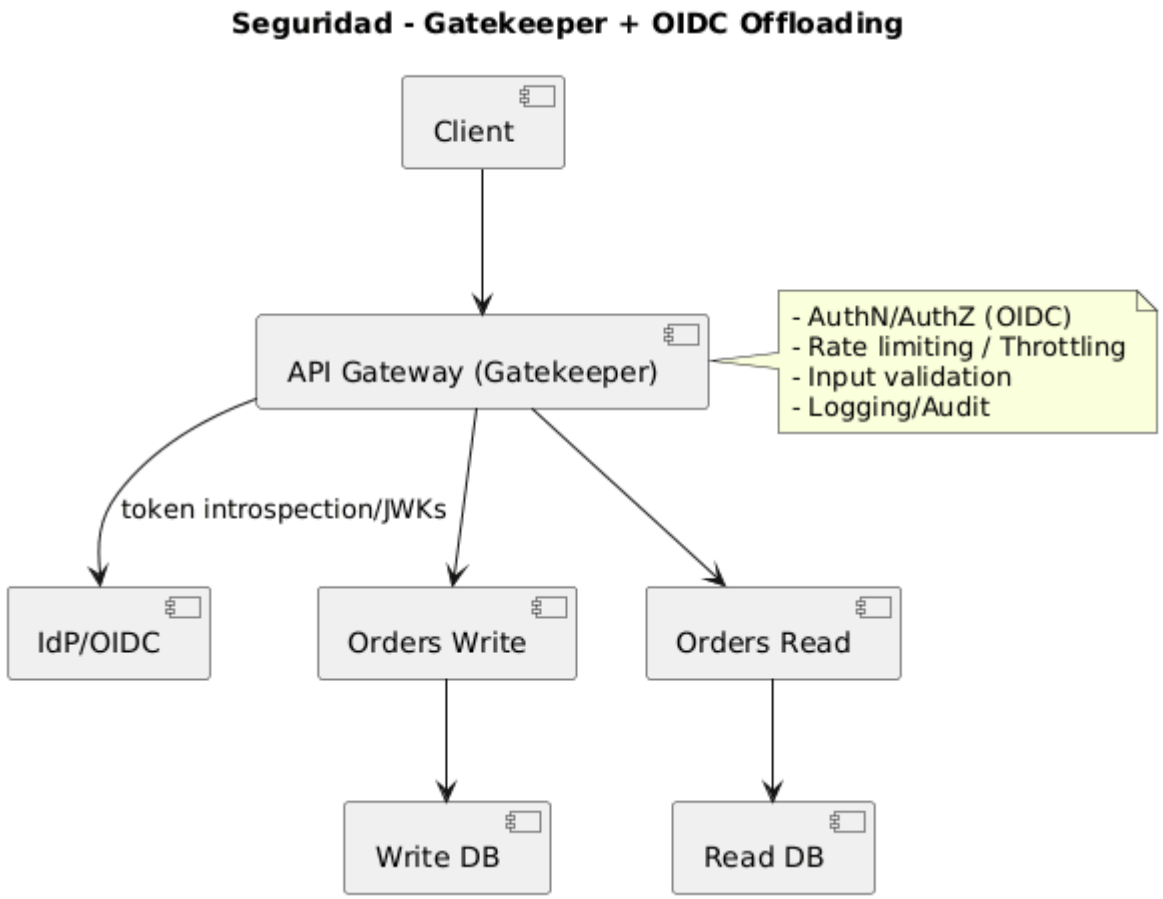
Componentes

- 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 focucen en 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

- Centraliza políticas, reduce superficie de ataque y simplifica auditoría; riesgos: SPOF/latencia si no se diseña HA.
- Offloading estandariza controles (cifrado, validación de tokens, RL).

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)

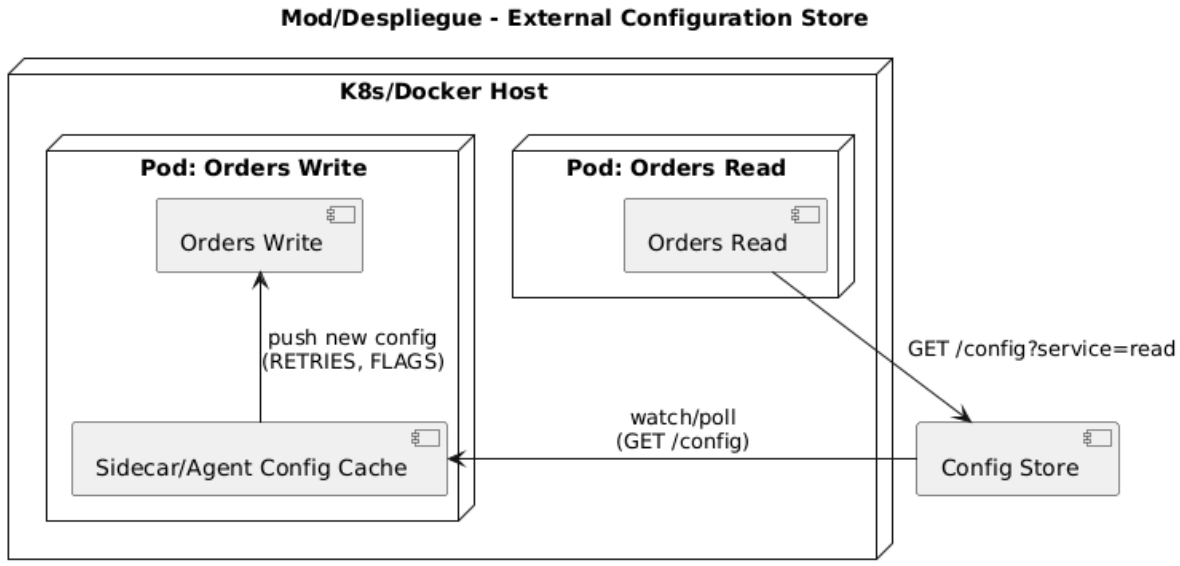
Componentes

- External Configuration Store: configuración centralizada versionable (flags de features, PAYMENT_MAX_RETRIES, límites de RL) consumida por los servicios en runtime. (Patrón cubierto en el set de despliegue/modificación; lo usamos para controlar comportamientos sin redeploy).

Implementación

- Cambios sin recompilar (feature toggles), rollback rápido, trazabilidad de cambios (auditoría).

UML Config central + hot-reload



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`

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: Resiliencia (Graceful Degradation)

Objetivo: evidenciar que los módulos siguen funcionando parcialmente bajo fallas.
Descripción:

- Si el adapter de pagos se cae, las órdenes se siguen registrando.
- Los mensajes quedan en la cola hasta que el consumidor vuelva a estar disponible.

Comandos:

```
docker compose stop payments-adapter

docker compose start payments-adapter
```

Aprendizaje: los sistemas resilientes degradan la funcionalidad sin colapsar.

Demo 4: Observabilidad y Rendimiento

Objetivo: analizar el comportamiento del sistema bajo carga y visualizar métricas de rendimiento.
Descripción:

- Analizar tiempos de respuesta, errores y throughput con los logs de Orders-Write.
- Comparar el rendimiento con y sin Circuit Breaker activo.

Comandos:

```
docker compose logs orders-write | grep "response"

docker stats
```

Aprendizaje: la observabilidad permite medir el impacto real de los patrones de resiliencia y optimizar capacidad.

Conclusiones

- **Circuit Breaker:** mejora disponibilidad reduciendo cascadas de fallos.
- **RabbitMQ:** incrementa rendimiento y estabilidad bajo carga.
- **Graceful Degradation:** mantiene la experiencia de usuario estable frente a errores.
- **Observabilidad:** permite evaluar y ajustar los patrones de diseño con datos reales.