

Virtual Threads vs Reactive Programming en Quarkus

Jose Diaz | 2026

Resumen

1. Introducción
2. Blocking I/O
3. Reactive Programming
4. Virtual Threads
5. Quarkus Demo
6. Comparación
7. Conclusiones

1. Introducción

Historia

Timeline

- 1997 -> Green threads
- 2000 -> Platform Threads
- 2023 -> Virtual Threads
 - Preview en JDK 19
 - Liberado en JDK 21
- ???? -> Concurrencia Estructurada

2. Blocking

I/O

- I/O (Input/Output) existe en todas partes:
 - Llamar a un servicio remoto
 - Interactuar con una base de datos
 - Enviar mensajes a un tópico
 - Etc.

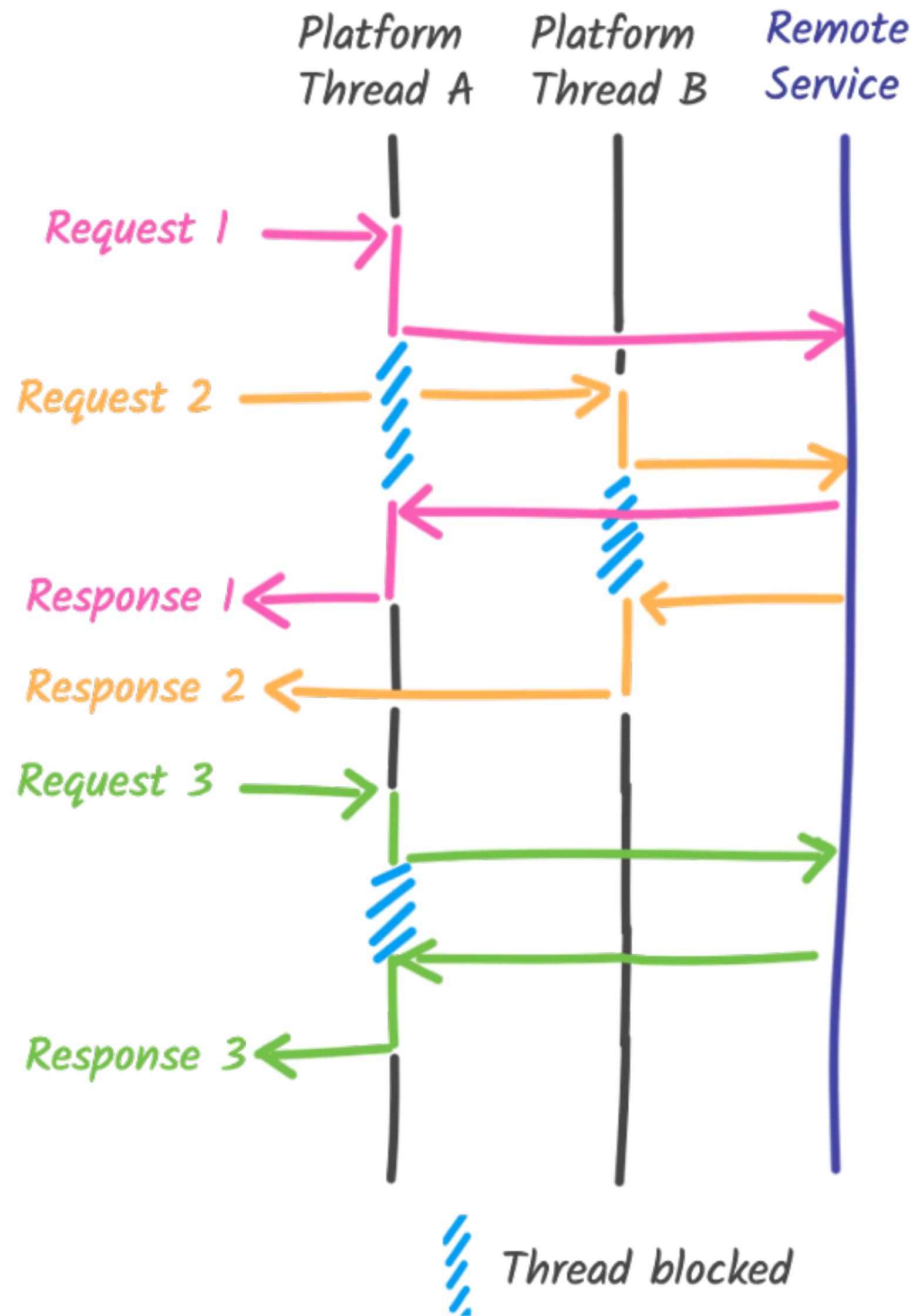
Blocking I/O

Esperar por siempre

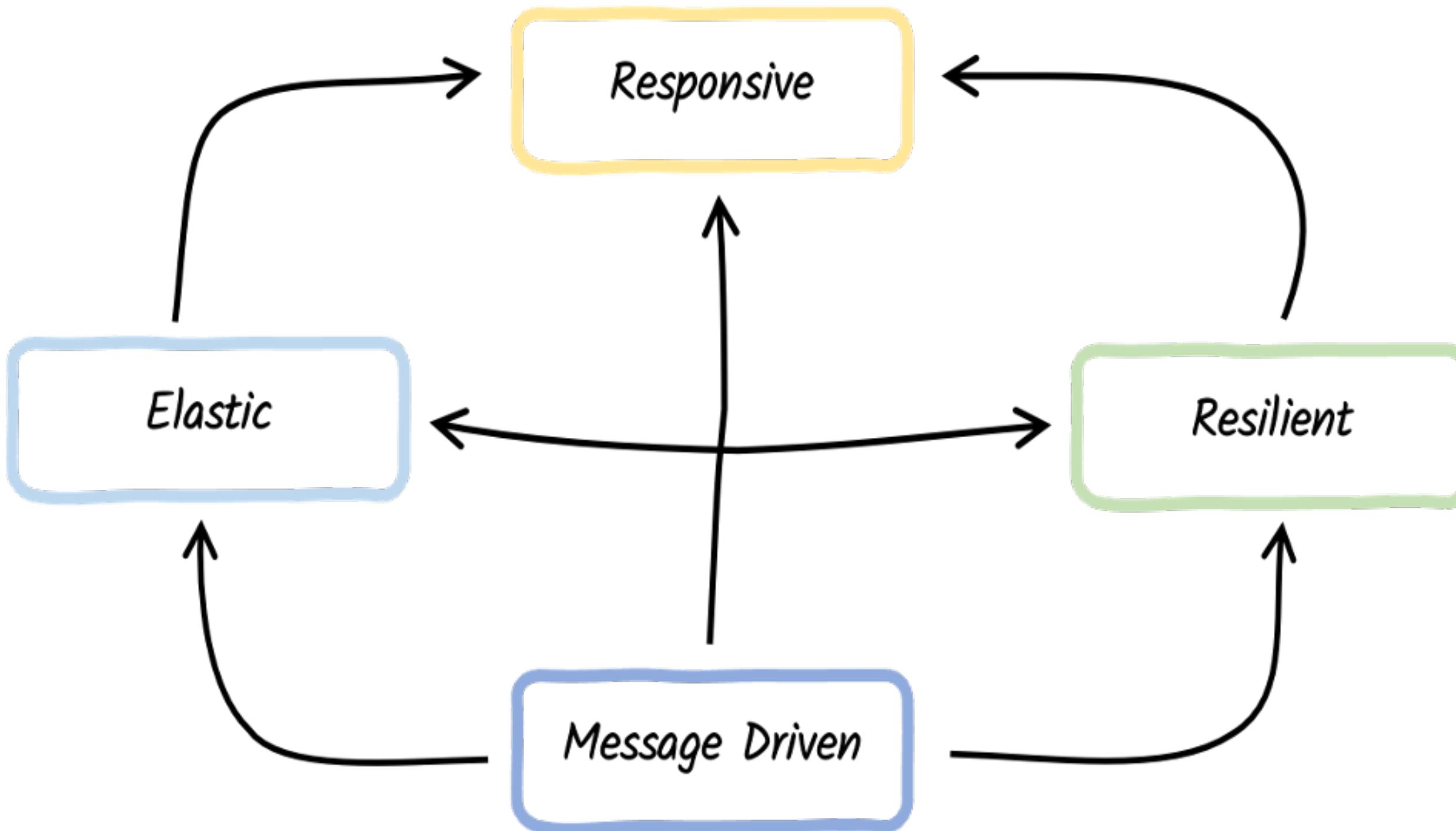
- Preguntar y esperar por algo
- No hacemos nada hasta que llegue la respuesta

Platform Threads

- Los hilos de plataforma se rapean directamente a hilos del sistema operativo (OS).
- Aproximadamente necesitan 1 MB de memoria de stack cada uno.
- Son costosos y lentos de crear.
- El sistema operativo se encarga de la planificación (scheduling)
- Por eso normalmente usamos pools de hilos de tamaño fijo para:
 - Peticiones web
 - Pool de conexiones a base de datos
 - Cálculos costos o intensivos
 - Etc.



3. Reactive



Responsivos (Responsive): deben responder de manera oportuna y consistente.

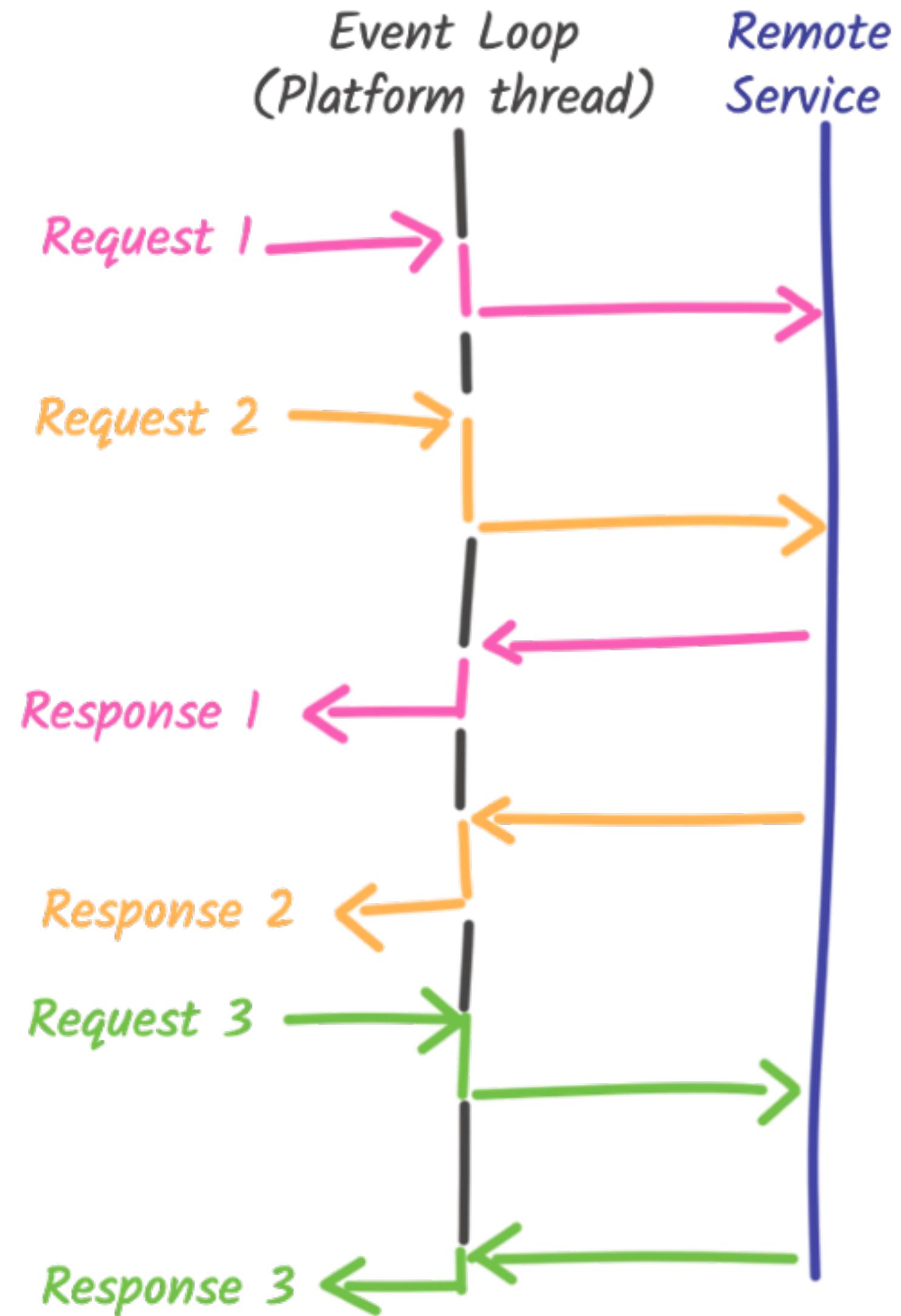
Elásticos (Elastic): se adaptan automáticamente a las fluctuaciones de carga.

Resilientes (Resilient): manejan las fallas de forma elegante, sin colapsar el sistema.

Orientados a mensajes (Message-driven): los componentes de un sistema reactivo interactúan entre sí mediante el intercambio de mensajes

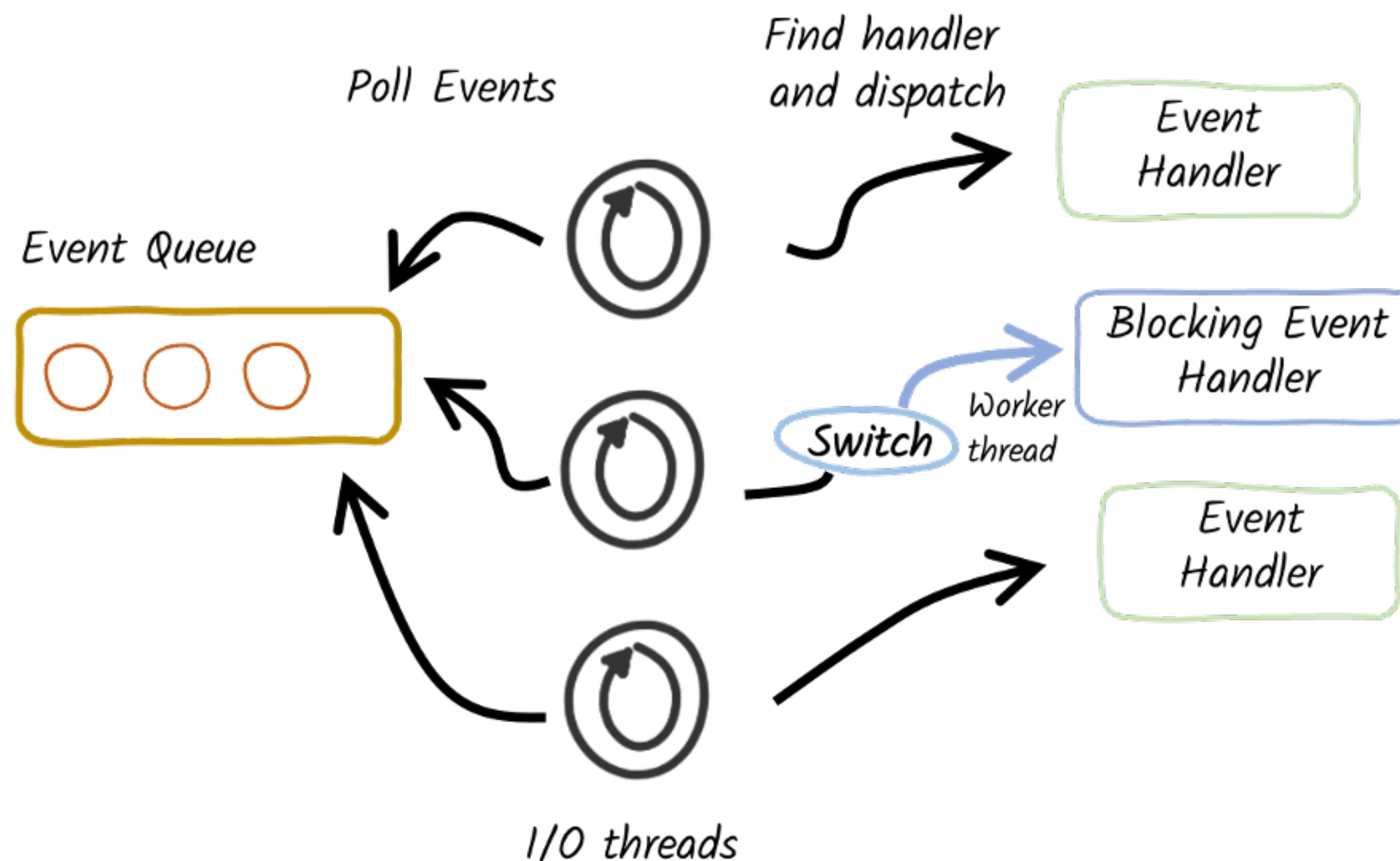
Sistemas Reactivos

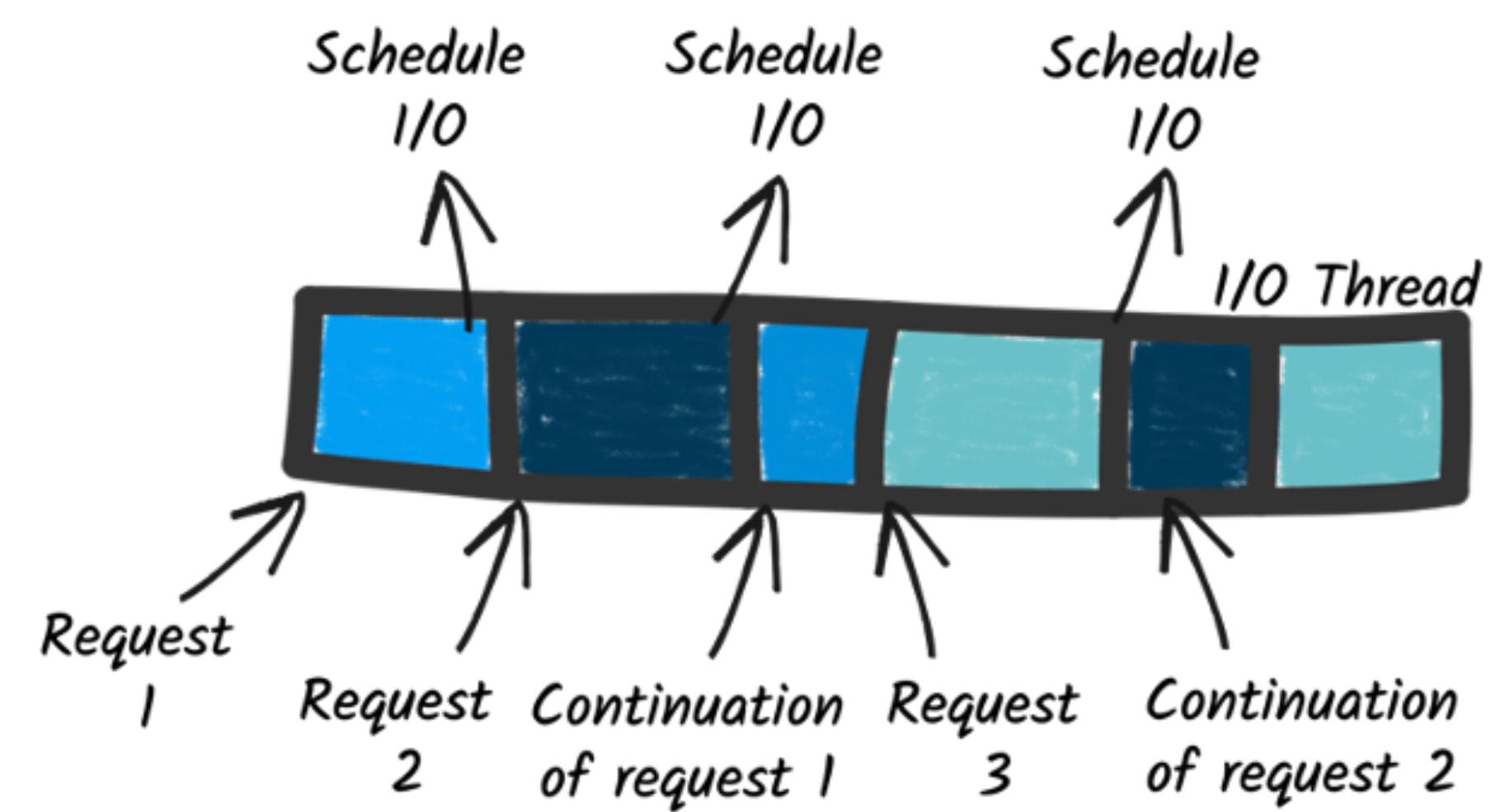
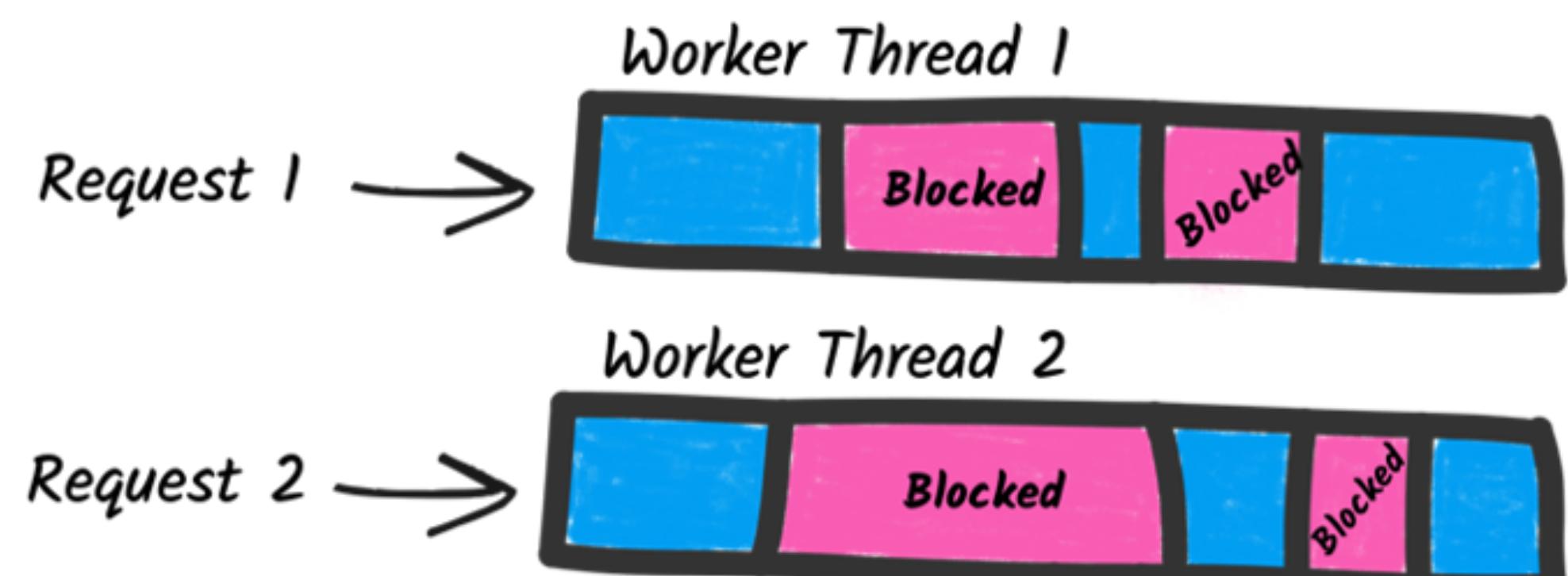
- No bloqueante (Non-blocking)
- Maneja los recursos de CPU y memoria de forma eficiente
- Un solo hilo puede manejar múltiples solicitudes
- Solo se necesitan unos pocos hilos de tipo event loop en la aplicación
- No es necesario usar thread pools



Unificación de Imperativo y Reactivo

- Quarkus permite usar tanto código bloqueante como no bloqueante dentro de una misma aplicación, facilitando la migración progresiva y el uso eficiente de recursos según el caso de uso.





Programación Reactiva

Tus opciones

- Múltiples opciones en Quarkus:
 - `io.smallrye.mutiny.Uni` y `io.smallrye.mutiny.Multi` (Mutiny)
 - `java.util.concurrent.CompletionStage` (CompletableFuture)
 - `org.reactivestreams.Publisher` (RxJava, Akka Streams y también Mutiny)
 - O usar corutinas de Kotlin con funciones *suspend*.

Mutiny

Uni & Multi

- Mutiny es una librería de programación reactiva
 - **Uni** → emite un simple evento (item o falla)
 - **Multi** → emite multiples eventos (0 o n items)
- Similar a Mono y Flux como en Spring WebFlux

```
1 Uni.createFrom().item(1)
2     .onItem().transform(i -> "hello-" + i)
3     .onItem().delayIt().by(Duration.ofMillis(100))
4     .subscribe().with(System.out::println);
```

```
1 Multi.createFrom().items(1, 2, 3, 4, 5)
2     .onItem().transform(i -> i * 2)
3     .select().first(3)
4     .onFailure().recoverWithItem(0)
5     .subscribe().with(System.out::println);
```

Extensiones

Quarkus

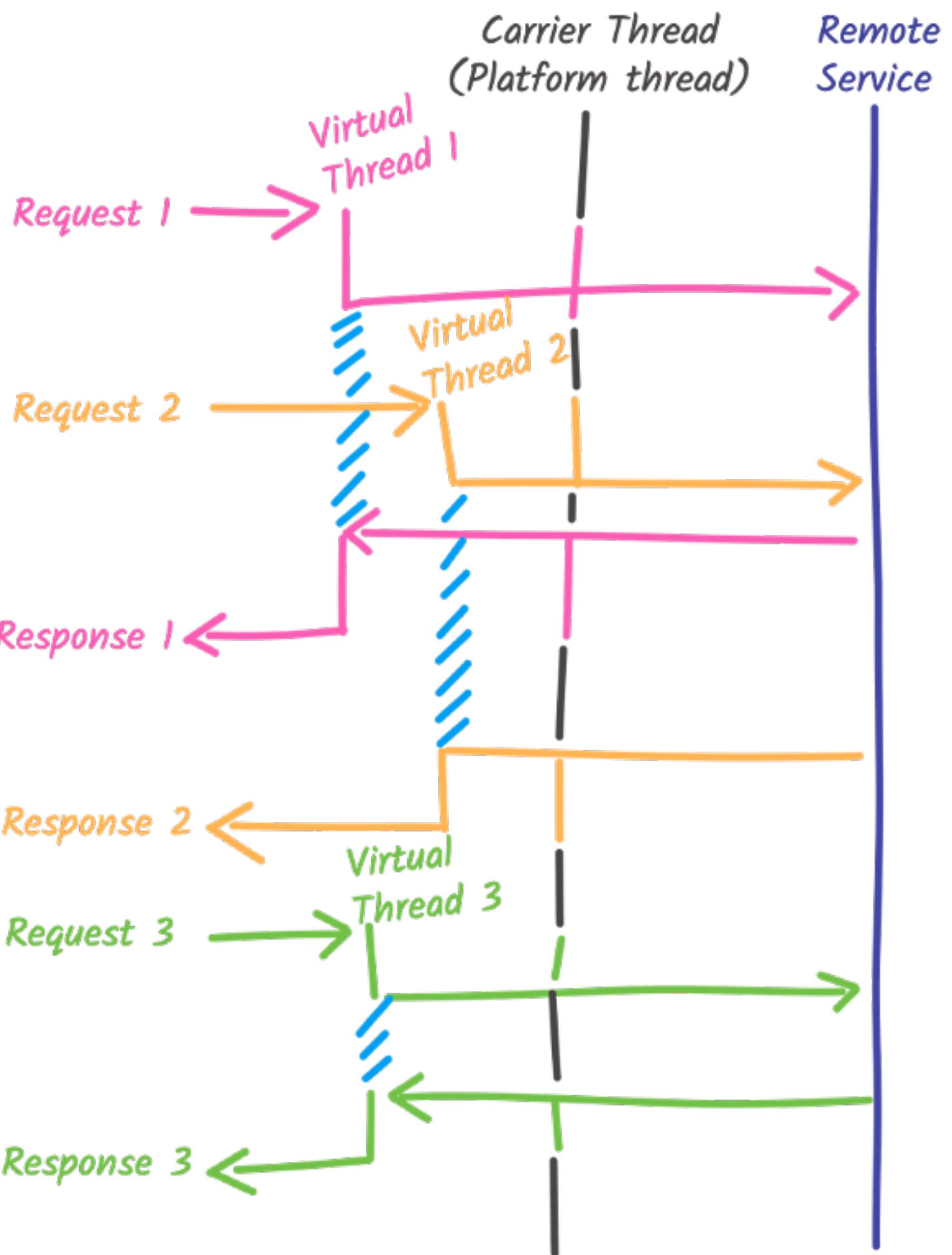
- Extensiones Reactivas:
 - Quarkus REST
 - Quarkus REST Client
 - Quarkus Hibernate Reactive
 - Quarkus Reactive Messaging
 - Quarkus Qute
 - Etc.
- Mas información en:
<https://code.quarkus.io>

4. Virtual Threads

Virtual Threads

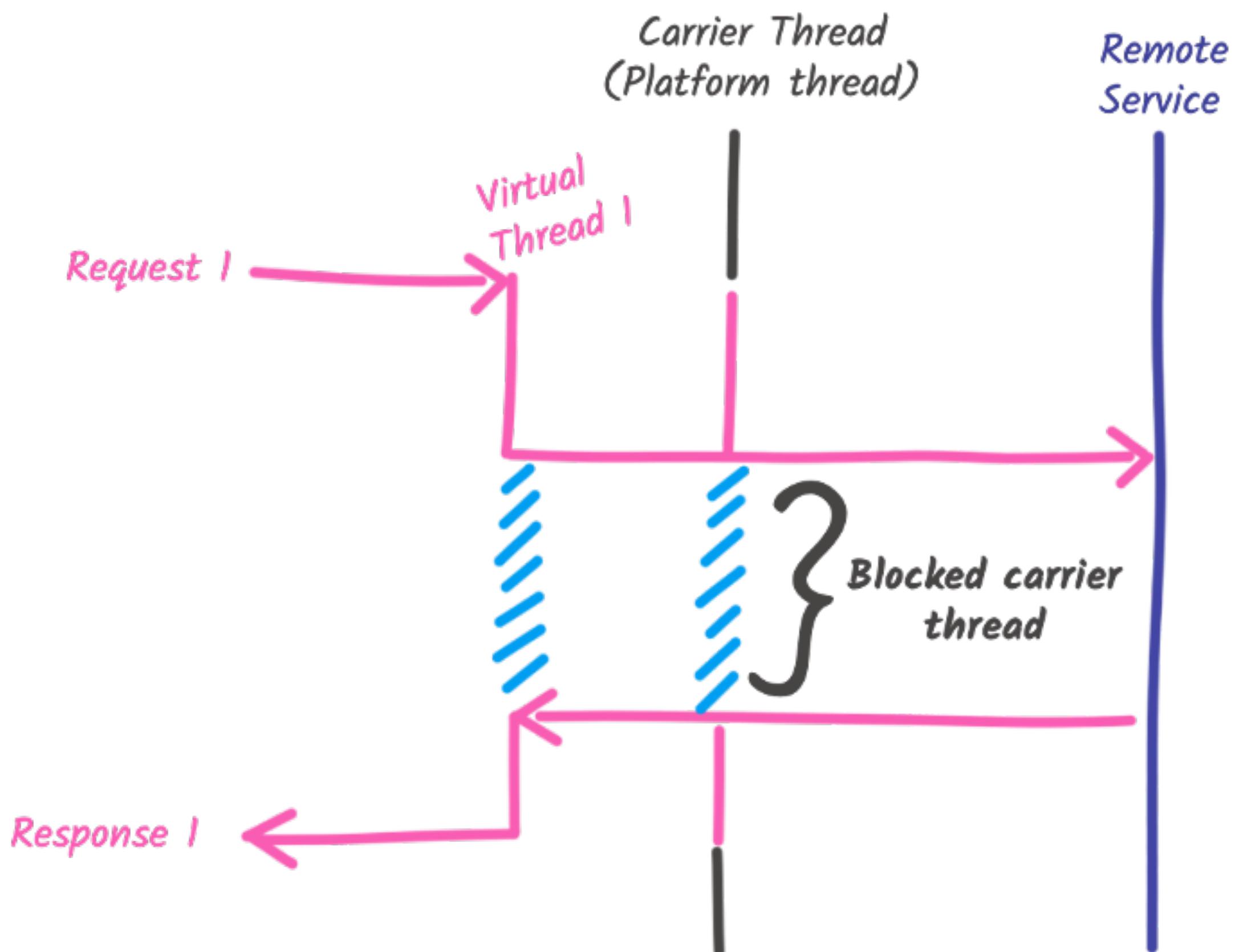
Al rescate

- Baratos de crear
- Usan menos memoria
- Administrados por la JVM
- Baratos de bloquear
- Escribes código imperativo
- Solo útiles para tareas limitadas por I/O (I/O-bound)



Pinning Y synchronized

- Muchas librerías usaban bloques **synchronized** internamente, por ejemplo las librerías JDBC.
- La mayoría de estas librerías ya han sido corregidas.
- Los bloques **synchronized** pueden causar pinning y bloquear tu aplicación.



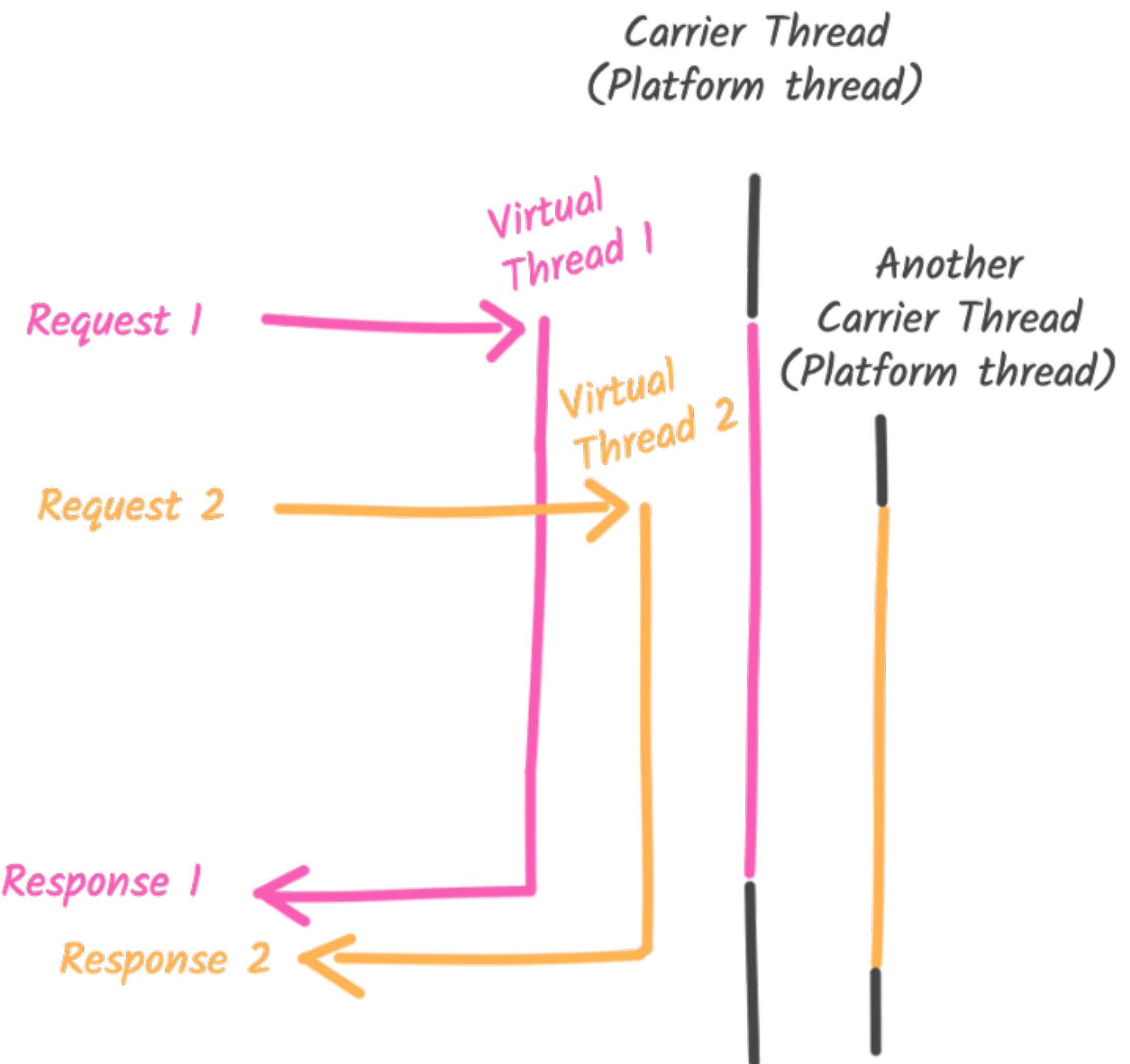
Pinning Y soluciones

- Habilitar detección con argumentos ivm
 - `-Djdk.tracePinnedThreads`
- Resuelto en JDK 24 con JEP 491
 - <https://openjdk.org/jeps/491>
 - Disponible en JDK 25
- O usar `java.util.concurrent.locks`
 - `ReentrantLock`

Monopolización

Con carga de CPU

- Puede ocurrir cuando tienes cargas de trabajo ligadas a CPU (CPU - bound) sin operaciones de I/O.
- Probablemente sea mejor delegarlas a un pool de hilos dedicado.



Virtual Threads

Y testing

- En Quarkus cada virtual thread tiene un nombre:
 - `quarkus-virtual-thread-xxx`
 - Facil de debuggear
- Extensión Junit para detectar el pinning
 - `io.quarkus.junit5:junit5-virtual-threads`
- Anotaciones disponibles para tests:
 - `@VirtualThreadUnit`
 - `@ShouldNotPin`
 - `@ShouldPin (atMost = 1)`

Concurrencia estructurada

- Hace más fácil escribir código concurrente
 - Evita thread leaks
 - Estrategias de cancelación
 - Etc.
- 5to. Preview en JDK 25
 - <https://openjdk.org/jeps/505>
- Fecha de liberación ???

6. Comparación

Blocking	Reactive	Virtual Threads
Programación interactiva	Manejo de múltiples requests en el mismo thread	Programación imperativa
Ejecución secuencial de items	Alto throughput	Facilidad de comprensión
Facilidad de comprensión	Baja latencia	Poco uso de memoria
Alto uso de memoria	Poco uso de memoria	Ligeramente menos eficiente en rendimiento
Limitada cantidad de threads	Más difícil de escribir y depurar cuando se usan callback o código reactivo	Concurrencia estructurada no está lista todavía

7. Conclusiones

Conclusiones

¿Cuál eliges?

- Bloqueante para código legacy
- Programación reactiva cuando necesitas máximo rendimiento y control
- Hilos virtuales siempre que quieras
 - Prueba primero, y cuidado con las sorpresas inesperadas.
 - Aprovecha la concurrencia estructurada cuando esté disponible
- Combina y mezcla en Quarkus

Referencias

- <https://quarkus.io/guides/quarkus-reactive-architecture>
- <https://quarkus.io/guides/getting-started-reactive>
- <https://quarkus.io/guides/virtual-threads>
- <https://quarkus.io/blog/virtual-thread-1/>
- <https://dl.acm.org/doi/10.1145/3583678.3596895> (paper)