UNIVERSIDAD POLITÉCNICA SALESIANA ECUADOR

POSGRADOS

MAESTRÍA EN
**SOFTWARE CON MENCIÓN EN DISEÑO DE ARQUITECTURA DE SISTEMAS**
RPC-SO-34-NO.778-2021

OPCIÓN DE TITULACIÓN:
PROYECTO DE TITULACIÓN CON COMPONENTES DE INVESTIGACIÓN APLICADA Y/O DE DESARROLLO

TEMA:
# EVALUACIÓN DE ENFOQUES IMPERATIVO Y REACTIVO EN EL DESARROLLO DE UNA API INTELIGENTE DE RECURSOS HUMANOS CON PREDICCIÓN DE ASISTENCIA EN UN ENTORNO DE MICROSERVICIOS

AUTORES:
CARLOS DANIEL ÁLVAREZ ZHAPA
ADRIÁN RODRIGO TENE GUAMAN

DIRECTOR:
GABRIEL ALEJANDRO LEÓN PAREDES

CUENCA – ECUADOR
2026

**Autores:**

**Carlos Daniel Álvarez Zhapa**
Ingeniero de Sistemas.
Candidato a Magíster en Software con Mención en Diseño de Arquitectura de Sistemas por la Universidad Politécnica Salesiana – Sede Cuenca.
calvarezz@est.ups.edu.ec

**Adrián Rodrigo Tene Guaman**
Ingeniero de Sistemas.
Candidato a Magíster en Software con Mención en Diseño de Arquitectura de Sistemas por la Universidad Politécnica Salesiana – Sede Cuenca.
ateneg1@est.ups.edu.ec

**Dirigido por:**

**Gabriel Alejandro León Paredes**
Ingeniero de Sistemas.
Máster Universitario en Ingeniera Web.
Doctor en Tecnologías de Información.
gleon@ups.edu.ec

***DEDICATORIA***

Dedicamos este trabajo, en primer lugar, a Dios, por concedernos la vida, la salud y las capacidades necesarias para avanzar con firmeza y culminar este proyecto. Su guía y su fortaleza han sido nuestro sostén en cada paso del camino, iluminando nuestro esfuerzo y dándonos paz en los momentos de mayor desafío.

De manera muy especial, dedicamos esta tesis a nuestras familias, quienes han sido nuestro apoyo incondicional durante todo el proceso. A nuestros padres, por su amor, sacrificio y confianza, por acompañarnos con palabras de ánimo y por brindarnos la fuerza emocional que necesitábamos para seguir adelante. A nuestros hermanos, por su compañía, comprensión y motivación constante, que nos impulsó a perseverar incluso cuando las dificultades parecían mayores.

Esta obra es el reflejo del esfuerzo compartido, del cariño y del apoyo recibido que cada uno que nos ha brindado. A ustedes, con profundo agradecimiento, dedicamos este logro.

Carlos Daniel Alvarez Zhapa & Adrián Rodrigo Tene Guamán

***AGRADECIMIENTO***

Expresamos nuestro más sincero agradecimiento a la Universidad Politécnica Salesiana, institución que nos brindó la oportunidad de formarnos académica y profesionalmente. Durante este proceso de estudios, la UPS nos ofreció un entorno de aprendizaje riguroso, humano y orientado a la excelencia, que ha contribuido de manera significativa a nuestro crecimiento personal y a nuestra formación como profesionales en el ámbito de la ingeniería de software.

De igual manera, extendemos nuestro reconocimiento a todos los docentes de la Maestría en Ingeniería de Software, quienes, a través de su compromiso, experiencia y dedicación, dejaron en nosotros una enseñanza profunda tanto en el aspecto técnico como en el humano. Cada clase, cada orientación y cada desafío académico representaron un aporte valioso que enriqueció nuestra visión y fortaleció nuestras competencias.

Nuestro agradecimiento especial es para nuestro tutor de tesis, cuyo acompañamiento constante, guía minuciosa y esfuerzo compartido fueron fundamentales para la realización de este proyecto. Su dedicación, criterio profesional y atención detallada nos permitieron avanzar con claridad y asegurar la calidad de este trabajo.

Finalmente, queremos agradecer de manera muy especial a nuestras familias, quienes nos ofrecieron apoyo incondicional durante todo este proceso. Su paciencia, motivación y respaldo emocional fueron pilares esenciales que nos permitieron culminar esta etapa con éxito. A ellos, por su amor y confianza, les debemos gran parte de este logro.

Con gratitud,

Carlos Daniel Alvarez Zhapa & Adrián Rodrigo Tene Guamán

# Tabla de Contenido

# Índice de Figuras

# Índice de Código

# Evaluación de Enfoques Imperativo y Reactivo en el Desarrollo de una API Inteligente de Recursos Humanos con Predicción de Asistencia en un Entorno de Microservicios


Autor(es):

Carlos Daniel Álvarez Zhapa
Adrián Rodrigo Tene Guamán

# Resumen

El presente trabajo que se ha desarrolla y evalúa una arquitectura basada en microservicios desplegados en contenedores Docker, que fue diseñada para comparar dos paradigmas de programación ampliamente utilizados en el desarrollo de software moderno: el paradigma imperativo y el paradigma reactivo. Para ello, fue implementado dos versiones independientes de un mismo conjunto de servicios orientados a un modelo de negocio de registro de asistencias y ausencias de Recursos Humanos, implementando una predicción de ausencias laborales. La primera implementación se ha construido sobre Spring Boot utilizando el modelo imperativo tradicional con Spring MVC, mientras que el segundo desarrollo fue programado empleando Spring WebFlux como parte del paradigma reactivo. Ambas arquitecturas comparten la misma infraestructura tecnológica, incluyendo un motor de predicción integrado y almacenamiento en bases de datos PostgreSQL, garantizando así igualdad de condiciones para su evaluación comparativa. Sobre esta base, los servicios fueron sometidos tanto a pruebas de estrés con alta concurrencia como a escenarios de carga real basados en patrones operativos del entorno empresarial, con el fin de medir y analizar métricas clave tales como rendimiento, escalabilidad, consumo de recursos y mantenibilidad. Los resultados obtenidos muestran diferencias cuantitativas relevantes. En escenarios de carga normal, la arquitectura reactiva alcanzó un P95 de 89 ms y un throughput de 28,67 req/s, mientras que la arquitectura imperativa registró un P95 de 345 ms y un throughput de 12,45 req/s. Bajo pruebas de estrés extremo, la arquitectura imperativa presentó un P95 de 1250 ms, un throughput de 8,23 req/s y una tasa de error del 8%, mientras que la arquitectura reactiva mantuvo un P95 de 156 ms, un throughput de 45,89 req/s y una tasa de error del 0%. No obstante, en cuanto al consumo de recursos, el enfoque imperativo mostró menores niveles de uso de CPU y memoria durante la carga normal, evidenciando un desempeño más eficiente en escenarios sin concurrencia elevada.

Estos resultados cuantitativos permiten identificar con precisión las fortalezas y limitaciones de cada paradigma.

**Palabras clave:** Microservicios, Docker, Spring Boot, Spring MVC, Spring WebFlux, Programación Reactiva, Programación Imperativa, R2DBC, PostgreSQL, Predicción de Ausencias, Recursos Humanos, Arquitectura de Software, Rendimiento, Escalabilidad, Concurrencia, Mantenibilidad.

# ABSTRACT

This work develops and evaluates a microservices-based architecture deployed in Docker containers, designed to compare two programming paradigms widely used in modern software development: the imperative paradigm and the reactive paradigm. To accomplish this, two independent versions of the same set of services were implemented, oriented toward a Human Resources system for recording employee attendance and absences, including a predictive model for forecasting absenteeism. The first implementation was built using Spring Boot with the traditional imperative model based on Spring MVC, while the second was developed using Spring WebFlux as part of the reactive paradigm. Both architectures share the same technological infrastructure, including an integrated prediction engine and PostgreSQL database storage, thus ensuring equal conditions for an objective comparative evaluation. Based on this setup, the services were subjected to both high-concurrency stress tests and real-world workload scenarios modeled after operational patterns found in enterprise environments, with the goal of measuring and analyzing key metrics such as performance, scalability, resource consumption, and maintainability. The results reveal notable quantitative differences. Under normal load conditions, the reactive architecture achieved a P95 latency of 89 ms and a throughput of 28.67 req/s, whereas the imperative architecture recorded a P95 of 345 ms and a throughput of 12.45 req/s. Under extreme stress tests, the imperative model reached a P95 of 1250 ms, a throughput of 8.23 req/s, and an error rate of 8%, while the reactive model maintained a P95 of 156 ms, a throughput of 45.89 req/s, and a 0% error rate. However, regarding resource usage, the imperative approach showed lower CPU and memory consumption during normal load, demonstrating more efficient behavior in scenarios without high concurrency.

These quantitative results allow for a precise identification of the strengths and limitations of each paradigm.

**Keywords:** Microservices, Docker, Spring Boot, Spring MVC, Spring WebFlux, Reactive Programming, Imperative Programming, R2DBC, PostgreSQL, Absence Prediction,

Human Resources, Software Architecture, Performance, Scalability, Concurrency, Maintainability.

# 1. Introducción

## 1.1 Antecedentes

La transformación de los sistemas hacia arquitecturas distribuidas y escalables ha impulsado el replanteamiento de los paradigmas tradicionales en el desarrollo de software. En particular, el enfoque imperativo caracterizado por una ejecución secuencial, control explícito del flujo y un fuerte acoplamiento entre componentes ha mostrado limitaciones crecientes en contextos de alta concurrencia y procesamiento masivo de datos. Estas limitaciones incluyen cuellos de botella en la escalabilidad, consumo elevado de recursos y dificultades en la gestión de latencias. (Mochniej & Badurowicz, 2023)

Frente a estas restricciones, la programación reactiva ha emergido como paradigma alternativo. Este enfoque, sustentado en asincronía, orientación a eventos y operaciones no bloqueantes, busca mejorar la resiliencia, la eficiencia en el uso de recursos y la capacidad de adaptación a picos de demanda. El fundamento de esta orientación a eventos, clave en la arquitectura reactiva, se remonta a trabajos seminales como el de Elmquist y Varshney (1992), quienes establecieron sus ventajas para la escalabilidad y el manejo de concurrencia en sistemas distribuidos. En particular, frameworks como Spring WebFlux (basado en el estándar Reactive Streams y Project Reactor) permiten construir microservicios que operan con menor uso de memoria RAM y mejor tolerancia a la variabilidad en la latencia de I/O.

Mochniej y Badurowicz (2023) compararon microservicios desarrollados en Java utilizando Spring Web (imperativo) y Spring WebFlux (reactivo), evaluando operaciones de lectura/escritura en base de datos, procesamiento de datos y transferencia de archivos bajo distintas cargas. Sus resultados indican que los microservicios reactivos sorprenden en entornos con latencia I/O elevada, mostrando mejor rendimiento y menor consumo de RAM en comparación con los imperativos.

En el dominio académico, otros estudios recogen que, si bien los sistemas reactivos ofrecen ventajas al manejar muchas solicitudes concurrentes y cargas de I/O, estas

mejoras no siempre se trasladan a tareas intensivas en CPU, donde la sobrecarga para gestionar flujos reactivos puede superar los beneficios.

El paradigma reactivo también encuentra justificación teórica. En el artículo Reactive Programming Paradigms in High-Throughput Distributed Systems, Kumar (2025) discute cómo los principios de eventos, contrapresión, resiliencia y arquitectura basada en flujos permiten mantener sistemas altamente responsivos y tolerantes a fallos.

Por otra parte, para abordar la problemática de la gestión, la planificación operativa, la optimización de la dotación de personal y la reducción de costos asociados a la improductividad de los Recursos Humanos, las organizaciones modernas dependen cada vez más de servicios digitales capaces de procesar grandes volúmenes de información en tiempo real. Esta dependencia ha impulsado tanto el uso de arquitecturas basadas en microservicios como la implementación de métodos para anticipar patrones de comportamiento de los empleados.

No obstante, a pesar del crecimiento de estas tecnologías, persiste una escasez de estudios que comparen de manera sistemática el impacto de distintos paradigmas de programación como son Imperativo y Reactivo, en entornos donde intervienen servicios predictivos distribuidos.

Diversos trabajos previos han demostrado que la adopción de microservicios y contenedores permite mejorar la escalabilidad, la flexibilidad y la capacidad de despliegue en sistemas empresariales. Asimismo, investigaciones en áreas como el análisis predictivo, los flujos de eventos y la orquestación de servicios sugieren que los enfoques reactivos pueden ofrecer ventajas en cargas altamente concurrentes debido a su naturaleza no bloqueante y su capacidad para gestionar operaciones Entrada/Salida intensivas.

No obstante, aunque estos estudios aportan evidencia relevante, la mayoría se ha realizado con casos de uso heterogéneos y bajo infraestructuras no equivalentes, lo que limita la posibilidad de extraer conclusiones comparables entre ambos paradigmas. Por ello, aún persiste la necesidad de ejecutar evaluaciones controladas donde las dos aproximaciones imperativa y reactiva se midan sobre el mismo caso de negocio y en condiciones idénticas de software, hardware e interacción, especialmente en dominios poco explorados como los servicios predictivos distribuidos.

En respuesta a esta brecha, el presente proyecto plantea una arquitectura basada en microservicios desplegados en contenedores Docker que canaliza las solicitudes hacia dos implementaciones independientes de un servicio de Recursos Humanos orientado a la predicción de ausencias. La primera está desarrollada con Spring Boot sobre el paradigma imperativo utilizando Spring MVC, mientras que la segunda emplea Spring WebFlux como representación del paradigma reactivo. Ambas implementaciones comparten un motor de predicción integrado y una misma base de datos PostgreSQL, garantizando así condiciones equitativas para el análisis comparativo.

Este enfoque experimental permite evaluar de manera controlada las diferencias en rendimiento, escalabilidad, consumo de recursos y mantenibilidad entre ambos paradigmas, utilizando tanto pruebas de estrés con alta concurrencia como escenarios de carga real basados en comportamiento empresarial. De esta forma, podemos justificar la necesidad de estudiar cómo la elección del paradigma de programación puede influir en la eficacia de sistemas distribuidos para la gestión de Recursos Humanos.

## 1.2 Objetivos

El objetivo de este trabajo es evaluar los enfoques de programación imperativa y reactiva en el desarrollo de una API inteligente orientada a la gestión de recursos humanos, con énfasis en la predicción de asistencia de empleados dentro de un entorno de microservicios. Para ello, se propone analizar el estado del arte sobre ambos enfoques de programación, sus aplicaciones en arquitecturas de microservicios y su integración con sistemas de predicción basados en inteligencia artificial. Asimismo, se diseñarán e implementarán dos versiones de la API, una utilizando programación imperativa y otra basada en programación reactiva, ambas integrando un modelo predictivo de asistencia. Finalmente, se realizará una evaluación comparativa del rendimiento, la eficiencia y la escalabilidad de ambas implementaciones bajo distintos escenarios de carga y uso.

## Objetivos General

Evaluar los enfoques de programación imperativa y reactiva en el desarrollo de una API inteligente orientada a la gestión de recursos humanos, con énfasis en la predicción de asistencia de empleados dentro de un entorno de microservicios.

## Objetivo Específicos

1) Analizar el estado del arte sobre los enfoques de programación imperativa y reactiva, así como sus aplicaciones en arquitecturas de microservicios y sistemas de predicción mediante inteligencia artificial.
2) Diseñar e implementar dos versiones de una API inteligente de recursos humanos, una basada en programación imperativa y otra en programación reactiva, integrando un modelo de predicción de asistencia de empleados.
3) Evaluar comparativamente el rendimiento, la eficiencia y la escalabilidad de ambas implementaciones en un entorno de microservicios, bajo distintos escenarios de carga y uso.

# 2. Marco teórico referencial

## 2.1 Arquitectura de Microservicios con Dockerización

La arquitectura de microservicios se ha convertido en uno de los estilos arquitectónicos principales en el desarrollo de sistemas modernos, esto debido a su capacidad para promover la escalabilidad, resiliencia y mantenibilidad, en aplicaciones distribuidas. (Newman, 2015). Según Newman (2015), los microservicios consisten en "un conjunto de servicios pequeños y autónomos que trabajan juntos" (p.2), cada uno con responsabilidades bien definidas y comunicándose a través de mecanismos ligeros, siendo los principales protocolos HTTP o mensajería asíncrona. Las arquitecturas monolíticas tradicionales chocan con este estilo, debido a que estos sistemas están ajustados para que se despliegue una única unidad.

En este choque la arquitectura de microservicios puede desarrollarse, versionarse, desplegarse y escalarse de manera independiente, facilitando la evolución continua del sistema sin afectar a los demás componentes. La independencia de despliegue es uno de los principales factores que han popularizado esta arquitectura en organizaciones que necesitan ciclos de entrega rápidos y capacidad de adaptación constante (Richards, 2020). La adopción de esta arquitectura se relaciona directamente con los principios descritos en "The Reactive Manifesto" que habla sobre que los sistemas modernos deben ser responsivos, resilientes, elásticos y orientados a mensajes (Bonér, Farley, Kuhn, & Thompson, 2014).

Un sistema distribuido basado en microservicios tiene varios retos como es la concurrencia, latencia, fallos parciales en la carga e inestabilidad; todos estos factores motivan la adopción de modelos de programación reactiva que permitan tolerancia a fallos, comunicación no bloqueante y escalabilidad horizontal. La arquitectura de microservicios facilita la implementación del paradigma reactivo debido a sus características. En este contexto, la Dockerización es fundamental, ya que Docker es una plataforma que permite empaquetar aplicaciones junto con sus dependencias en contenedores ligeros, portables y reproducibles (Docker Inc., 2020), los contenedores

proporcionan aislamiento a nivel del sistema operativo, tiempos de despliegue reducidos entre entornos de desarrollo, prueba y producción. Esto resulta muy importante en arquitecturas de microservicios, donde cada servicio puede ejecutarse en su propio contenedor, garantizando la independencia tecnológica y la capacidad de escalar horizontalmente únicamente en los componentes que requieran.

La combinación de esta arquitectura y docker, simplifica la infraestructura necesaria para sistemas reactivos, ya que permite desplegar servicios independientes sin necesidad de acoplamiento, y poder crear entornos reproducibles para pruebas comparativas, además de escalar servicios reactivos de forma dinámica ante incrementos de carga, e integrar herramientas de monitoreo de latencia, y consumo de recursos. Como señalan Newman (2015) y Bonér et al. (2014), la elasticidad y resiliencia dependen directamente de la capacidad del sistema para distribuir cargas y aislar fallos, capacidades que se ven potenciadas mediante la infraestructura basada en contenedores.

Desde una perspectiva arquitectónica, la dockerización facilita la experimentación y evaluación de rendimiento en sistemas basados en el paradigma de la programación reactiva, permitiendo medir de manera controlada la eficiencia del paradigma bajo condiciones realistas de despliegue distribuido. Esto convierte a los microservicios y los contenedores como elementos fundamentales para el desarrollo de software moderno, convirtiéndolas en, no solo unas herramientas tecnológicas, sino en herramientas para comprender el comportamiento, las ventajas y las limitaciones de los enfoques reactivos.

## 2.2 Paradigma de programación Imperativa

La programación imperativa nació directamente del modelo de von Neumann, donde los programas se conciben como secuencias de instrucciones que modifican una memoria centralizada. En su famoso documento, von Neumann describió esta arquitectura como una “máquina cuyo comportamiento se determina por instrucciones que alteran un estado interno almacenado en memoria” (von Neumann, *First Draft of a Report on the EDVAC*, 1945). Este principio influenció profundamente a lenguajes modernos como

C, Java y Python, estableciendo la programación imperativa como la base de gran parte de la computación contemporánea. El paradigma imperativo se caracteriza por la mutación explícita de estado a través de variables, estructuras de datos y procedimientos. David A. Watt explica que "la semántica imperativa se fundamenta en comandos que producen efectos en el estado, los cuales determinan el curso de la ejecución subsecuente" (Watt & Brown, *Programming Language Design Concepts*, 2000). Esto hace que su modelo sea especialmente expresivo para describir algoritmos de manera clara, aunque también introduce desafíos importantes, sobre todo cuando se trabaja en contextos concurrentes donde los cambios simultáneos de estado pueden producir inconsistencias difíciles de detectar. Uno de los rasgos más distintivos de este paradigma es el control directo del flujo de ejecución mediante estructuras como if, while, for y, en sus inicios, go-to. Donald Knuth, en su análisis del modelo imperativo, señalaba que "los lenguajes imperativos reflejan la lógica del pensamiento algorítmico secuencial" (Knuth, *Structured Programming with go to Statements*, 1974). Esta forma lineal de razonar permite al desarrollador imaginar la ejecución de su programa como una serie de pasos encadenados, lo cual resulta especialmente útil tanto en contextos educativos como industriales.

Otro punto importante es su estrecha relación con la teoría de autómatas y máquinas de estado. En esencia, un programa imperativo puede entenderse como una secuencia de transiciones entre estados de memoria. Hopcroft y Ullman afirman que "las máquinas de estado y los autómatas deterministas son bases formales ideales para describir sistemas imperativos ya que operan por modificaciones sucesivas del estado" (Hopcroft & Ullman, *Introduction to Automata Theory, Languages, and Computation*, 1979). Desde esta perspectiva, se puede observar que la lógica imperativa está profundamente entrelazada con la computación teórica que sustenta los lenguajes de programación modernos. Históricamente, lenguajes como C y C++ son plenamente imperativos por diseño. Ambos ofrecen un control granular sobre la memoria, el manejo de punteros y la interacción directa con el hardware. Dennis Ritchie sostenía que C "fue creado como un lenguaje flexible para manipular directamente estructuras de memoria y sistemas operativos reales" (Ritchie, *The Development of the C Language*, 1993). Esto evidencia que el paradigma imperativo ha sido esencial para el desarrollo de sistemas de alto rendimiento, tales como kernels, drivers, runtimes y software embebido. En la actualidad,

lenguajes ampliamente utilizados como Python, JavaScript y Java mantienen un núcleo imperativo, aunque integren características de otros paradigmas. Guido van Rossum reconoce que "Python es, en esencia, un lenguaje imperativo con azúcares sintácticos funcionales y orientados a objetos" (van Rossum, *History of Python*, 2007). De manera similar, Brendan Eich describe JavaScript como "un lenguaje multiparadigma cuyo corazón operativo sigue siendo imperativo y basado en mutación" (Eich, *The Birth and Evolution of JavaScript*, 2006). Estos testimonios muestran cómo la imperatividad sigue siendo la base conceptual incluso en lenguajes considerados modernos o híbridos.

En plataformas empresariales como Spring Boot, este carácter imperativo se evidencia en tecnologías como JDBC, JPA/Hibernate, Servlets y Tomcat. Todas ellas fueron diseñadas originalmente bajo un modelo de ejecución secuencial y bloqueante. Brian Goetz afirma que "la mayor parte de las bibliotecas de Java están diseñadas bajo el supuesto de que las operaciones de E/S son bloqueantes y que los hilos gestionan la concurrencia de manera explícita" (Goetz et al., *Java Concurrency in Practice*, 2006). Esto refuerza que frameworks ampliamente adoptados, como Spring Boot, continúan basándose en un estilo imperativo a pesar de la aparición de paradigmas que promueven la asincronía y la reactividad. Sin embargo, uno de los mayores desafíos para este paradigma es la gestión correcta de la concurrencia. Leslie Lamport advierte que "la complejidad de los sistemas concurrentes proviene en gran medida de la interacción no controlada entre operaciones que modifican estados compartidos" (Lamport, *Specifying Systems*, 2002). Esta problemática ha impulsado la creación de paradigmas alternativos, como el funcional puro o el reactivo, que buscan minimizar el uso de estado mutable y reducir la dependencia del tiempo de ejecución. A pesar de esto, el paradigma imperativo ha demostrado una capacidad única de adaptación a lo largo del tiempo. Su influencia sigue siendo determinante en sistemas operativos, redes, lenguajes de programación, teoría de autómatas, arquitecturas de computación y métodos educativos. Aunque existen paradigmas alternativos que prometen mejoras en ciertas áreas específicas, el imperativo no ha sido desplazado. Robert Harper lo expresa de forma contundente: "el paradigma imperativo no desaparecerá; es la base misma del modelo computacional subyacente a la mayoría del hardware moderno" (Harper, *Practical Foundations for Programming Languages*, 2012).

Esto reafirma que la programación imperativa, lejos de ser un estilo obsoleto, sigue siendo un eje fundamental en el desarrollo tecnológico actual.

## 2.3 Paradigma de programación Reactiva

El paradigma de la programación reactiva se centra en el manejo de flujos eficientes de datos, la asincronía y la "automatización" de eventos dentro de un sistema. Se fundamenta con modelos basados en flujos continuos y arquitecturas dirigidas por eventos, los cuales comenzaron a investigarse en 1970. Fue adquiriendo mayor importancia durante la evolución de sistemas concurrentes y distribuidos, en los años 90 y 2000 (Elmquist & Varshney, 1992). La publicación de "Reactive Manifesto", establecieron los pilares para el diseño de sistemas modernos altamente escalables y resilientes (Bonér, Farley, Kuhn & Thompson, 2014).

Este paradigma se sustenta principalmente en:

- La programación funcional, que impulsa la composición y transformación declarativa de datos.
- Los flujos de datos continuos, donde la información se procesa como secuencias en movimiento.
- La arquitectura dirigida por eventos, que promueve procesos asincrónicos, desacoplados y distribuidos.

En los sistemas reactivos, las respuestas de los componentes, es automática al a ver algún cambio en las fuentes de datos, eliminando tener la necesidad de un control de flujo imperativo, dando como beneficio la escalabilidad y la reducción del costo en operaciones de Entrada/Salida, necesarios en aplicaciones con alta concurrencia (Meijer, 2012).

Para poder realizar el desarrollo de programación reactiva, Pivotal creo el "Project Reactor", que es el motor reactivo del ecosistema Spring. Implementando:

- Reactive Streams: Conjunto estándar para flujos asincrónicos con "backpressure "(mecanismo de control de flujo de datos) (Kuhn, 2015).
- Mono y Flux*:* Mono es una abstracción fundamental que modela operaciones asincrónicas que producen cero o un valor. Por otra parte, Flux representa los flujos de varios elementos donde pueden llegar a ser infinitos. Siendo estas dos abstracciones implementaciones de Reactive Streams, permitiendo un

procesamiento no bloqueante, declarativo y con un control de flujo de datos, necesario para sistemas altamente concurrentes (Pivotal Software, 2020).

- Spring WebFlux: Incorporado desde Spring Framework 5, representa la adopción oficial de Spring del enfoque reactivo de extremo a extremo. A diferencia de Spring MVC, que tiene un modelo tradicional bloqueante sobre servlets. WebFlux funciona sobre event loops, utilizando servidores como Netty (Spring, 2023). Su arquitectura está extremadamente integrada con Project Reactor, dando como resultado el manejo de un gran número de solicitudes concurrentes con un consumo de recursos significativamente menor.

Para resolver el problema del acceso a bases de datos relacionales, se creó R2DBC, ya que JDBC es completamente bloqueante. Esta propuesta permite consultas asincrónicas, retorno de resultados mediante Mono y Flux, y evita bloqueos en las operaciones de Entrada/Salida, permitiendo que sean completamente reactivas (R2DBC, 2020). Gracias a R2DBC, los sistemas pueden mantener un flujo reactivo coherente desde el cliente hasta la base de datos.

Esta tecnología posibilita mantener un flujo reactivo coherente desde el cliente hasta la capa de persistencia, eliminando la necesidad de hilos dedicados para operaciones de Entrada/Salida y permitiendo arquitecturas verdaderamente reactivas y escalables (R2DBC, 2020).

Desde el punto de vista arquitectónico, Project Reactor y R2DBC contribuyen a construir sistemas que cumplen con los principios del Reactive Manifesto: responsivos, elásticos, resilientes y orientados a mensajes (Bonér et al., 2014). Project Reactor provee un ecosistema completo de operadores, planificadores y mecanismos de backpressure basados en la especificación *Reactive Streams* (Kuhn, 2015; Pivotal Software, 2020), lo que permite gestionar miles de solicitudes concurrentes sin bloqueo.

Por su parte, R2DBC elimina uno de los cuellos de botella más críticos de los sistemas reactivas: el acceso bloqueante a bases de datos relacionales, ofreciendo un protocolo completamente asincrónico (R2DBC, 2020). Este enfoque es especialmente relevante en arquitecturas de microservicios y sistemas distribuidos, donde la eficiencia y la capacidad de manejar alto tráfico son fundamentales (Newman, 2021; Kleppmann, 2017). En conjunto, ambos proyectos han sido ampliamente estudiados y

documentados en la literatura técnica y académica, consolidándolos como pilares esenciales para arquitecturas modernas basadas en flujos y eventos (Meijer, 2012).

## 2.4 Comparación entre paradigmas

El paradigma imperativo y reactivo representan un diseño distinto del software moderno. Ambos arraigando un historial profundamente diferentes proponiendo soluciones de forma opuesta, Mientras el desarrollo imperativo surge de Von Neumann y del modelo secuencial de computación, el desarrollado reactivo emerge de la necesidad de manejar grandes volúmenes de eventos de asincronía masiva y sistemas distribuidos.

Desde el punto de vista conceptual, el paradigma imperativo se basa en la manipulación explícita del estado. Von Neumann describió este modelo como una secuencia estructurada de instrucciones que transforman memoria mutable, lo que se convirtió en el modelo mental predominante durante décadas. Como señalan Watt y Brown, la semántica imperativa se fundamenta en comandos que modifican el estado y cuya ejecución define el comportamiento del sistema. En contraste, el paradigma reactivo considera el software como un conjunto de flujos continuos de datos que se transforman declarativamente. En lugar de pensar en pasos secuenciales, los sistemas reactivos responden automáticamente a los cambios en los datos, siguiendo un estilo funcional, no bloqueante y orientado a eventos.

En términos de flujo de control, las diferencias son aún más evidentes. El enfoque imperativo coloca al desarrollador en control explícito del flujo mediante estructuras como if, while o for, siguiendo la lógica secuencial que Knuth defendía como la base del pensamiento algorítmico. La programación reactiva elimina este control explícito. Los operadores reactivos permiten componer pipelines de transformación donde el flujo “se empuja” automáticamente por el sistema cuando ocurre un evento. Meijer describe esta diferencia como un cambio del “pull-based model” (imperativo) hacia un “push-based model” (reactivo), donde el sistema notifica automáticamente al consumidor sin intervención del desarrollador.

En cuanto al manejo del estado, la programación imperativa depende fuertemente de la mutación. Esto facilita la representación concreta del hardware, pero introduce complejidad significativa en entornos concurrentes. Lamport advirtió que gran parte de

la complejidad de la concurrencia proviene precisamente de estados compartidos mutables. La programación reactiva intenta reducir esta fricción adoptando flujos inmutables, composición funcional y backpressure, permitiendo que el sistema administre la concurrencia sin exponer al desarrollador a sincronizaciones manuales o bloqueos explícitos.

La asincronía es otro punto clave de contraste.

- En el modelo imperativo tradicional (JDBC, servlets, threads), cada operación bloquea y ocupa recursos hasta completarse. Brian Goetz señala que gran parte del ecosistema Java fue diseñado asumiendo operaciones bloqueantes, lo cual funciona bien para cargas moderadas, pero no escala de manera eficiente cuando el número de conexiones concurrentes crece.
- En contraste, la programación reactiva utiliza event loops, flujos no bloqueantes y control de backpressure, permitiendo manejar decenas de miles de conexiones concurrentes. Tecnologías como Project Reactor, WebFlux y R2DBC adoptan este enfoque, eliminando hilos bloqueados y optimizando el uso de CPU y memoria. Bonér y Farley argumentan que este modelo—alineado con el Reactive Manifesto—permite construir sistemas más responsivos, elásticos y resilientes.

En términos de modelo mental, la programación imperativa es más intuitiva para la mayoría de los desarrolladores, ya que coincide con la forma natural de expresar algoritmos paso a paso. Este enfoque ha sido la base de la educación en ciencias de la computación. Sin embargo, este mismo modelo mental se vuelve difícil en presencia de asincronía compleja. La programación reactiva, aunque más difícil de aprender, ofrece mecanismos de composición más robustos para manejar la complejidad de datos en movimiento y sistemas distribuidos modernos.

A nivel arquitectónico, los sistemas tradicionales basados en Spring Boot —MVC, Tomcat, JDBC— se apoyan en un modelo imperativo y bloqueante. Funcionan bien bajo cargas moderadas y para arquitecturas monolíticas. Sin embargo, la tendencia hacia microservicios, APIs de alta concurrencia y arquitectura cloud-native exige modelos más eficientes. Aquí es donde WebFlux, Project Reactor y R2DBC proporcionan un camino hacia sistemas completamente no bloqueantes. Estos permiten que un microservicio sea escalable sin necesidad de incrementar linealmente los recursos del servidor, algo que es difícil de lograr con la arquitectura imperativa clásica.

Finalmente, desde una perspectiva filosófica y evolutiva, la programación imperativa no desaparecerá; sigue siendo la base del hardware moderno y de lenguajes como C, Java o Python. Harper señala que es esencial para la computación contemporánea. Sin embargo, la programación reactiva tampoco es simplemente una moda: es una respuesta arquitectónica al crecimiento exponencial de los datos, usuarios y dispositivos distribuidos.

Ambas coexisten, se complementan y responden a problemas distintos. La imperativa domina el cómputo estructural y los algoritmos secuenciales, mientras la reactiva gobierna los sistemas dinámicos, asincrónicos y con alta demanda de concurrencia.

## 2.5 Trabajos Relacionados

La evolución de las arquitecturas de software en la última década ha impulsado una transición significativa desde modelos monolíticos hacia arquitecturas distribuidas basadas en microservicios. Este cambio estructural ha motivado la adopción de nuevos paradigmas de programación que se adaptan mejor a entornos altamente concurrentes y distribuidos, como es el caso de muchas aplicaciones empresariales actuales.

En este contexto, dos enfoques han sido ampliamente estudiados: la programación imperativa y la programación reactiva. La programación imperativa, caracterizada por una secuencia explícita de instrucciones que modifican el estado del programa, ha sido tradicionalmente la base del desarrollo de software empresarial (Goetz et al., 2006). Su simplicidad y claridad en la lógica de control la hacen adecuada para aplicaciones donde la predictibilidad, el mantenimiento y la depuración son esenciales. Sin embargo, en escenarios donde predominan operaciones I/O intensivas y concurrencia elevada, este enfoque presenta limitaciones de escalabilidad debido al bloqueo de hilos y la contención de recursos, desafíos ampliamente descritos en sistemas distribuidos modernos (Elmquist & Varshney, 1992).

Por otro lado, la programación reactiva, centrada en la manipulación de flujos de datos asíncronos y el manejo no bloqueante de operaciones, ha demostrado ser una alternativa eficiente para mejorar la capacidad de respuesta de los sistemas. Este enfoque permite controlar el backpressure y manejar múltiples eventos concurrentes

utilizando menos hilos, tal como establece la especificación Reactive Streams (Kuhn, 2015) y como desarrollan las tecnologías basadas en Project Reactor (Pivotal Software, 2020).

Estudios experimentales, como el de Kacper Mochniej y Marcin Badurowicz (2023), han comparado implementaciones de microservicios utilizando Spring MVC (modelo imperativo) y Spring WebFlux (modelo reactivo). En su análisis, se desarrolló una aplicación compuesta por múltiples microservicios que ejecutaban operaciones CRUD y llamadas a servicios auxiliares. Bajo pruebas de carga, se observó que las implementaciones con WebFlux mantenían latencias más bajas y un uso reducido de hilos en escenarios I/O-bound, confirmando los beneficios de este modelo en contextos donde la eficiencia y la escalabilidad son críticas.

En años recientes, también se han explorado alternativas como las Virtual Threads, introducidas en Project Loom, que buscan cerrar la brecha entre los modelos imperativos y reactivos. Documentación oficial de OpenJDK muestra que las Virtual Threads permiten manejar alta concurrencia con un modelo imperativo tradicional sin incurrir en el costo de los hilos pesados del sistema operativo, ofreciendo un rendimiento competitivo para aplicaciones I/O-bound (OpenJDK, 2023). Esto sugiere que la elección del paradigma no debe ser únicamente técnica, sino también estratégica, considerando factores operativos, de adopción tecnológica y experiencia del equipo de desarrollo.

Comparativas recientes, como las de Sukhambekova (2025), profundizan en las diferencias arquitectónicas y los patrones de diseño entre Spring MVC y Spring WebFlux, resaltando los trade-offs clave: mientras que Spring MVC proporciona un ecosistema maduro y sencillo de implementar, WebFlux ofrece una mejor escalabilidad y aprovechamiento de recursos bajo cargas concurrentes y dominadas por I/O.

En el ámbito de las APIs empresariales, también ha cobrado relevancia la integración de modelos de aprendizaje automático (ML) para funciones como la predicción de ausentismo en Recursos Humanos. Investigaciones contemporáneas muestran que

técnicas como SVM, árboles de decisión, ensembles y redes neuronales pueden alcanzar niveles de precisión significativos en este tipo de problemas, aunque gran parte de estos desarrollos aún se limita a procesos batch o pipelines no integrados en arquitecturas de microservicios (Nath et al., 2022). El desafío actual consiste en integrar modelos de ML en sistemas distribuidos que operen en tiempo real y bajo alta concurrencia sin comprometer rendimiento.

En cuanto a la evaluación del rendimiento en sistemas distribuidos, la herramienta k6 se ha consolidado como una de las soluciones más robustas para pruebas de carga orientadas a desarrolladores. Según Grafana Labs (2023), k6 permite ejecutar pruebas de carga, estrés y picos, proporcionando métricas como percentiles de latencia (P95, P99), throughput sostenido y tasas de error. Su integración con Docker (Docker Inc., 2020) facilita la creación de entornos reproducibles esenciales para comparativas experimentales. De hecho, estudios como los de Mochniej y Badurowicz (2023) han utilizado k6 para evaluar arquitecturas imperativas y reactivas, mientras que análisis arquitectónicos recientes lo emplean en microbenchmarks para identificar trade-offs entre modelos basados en hilos y modelos basados en eventos (Sukhambekova, 2025). El uso de herramientas como k6 se alinea con las recomendaciones del Reactive Manifesto (Bonér et al., 2014) y con patrones de diseño de microservicios ampliamente documentados (Newman, 2015), donde la evaluación empírica bajo carga constituye un elemento crítico para la toma de decisiones arquitectónicas.

A pesar de estos avances, persisten desafíos relevantes en el diseño e implementación de APIs empresariales, especialmente en cuanto a estandarización, modularidad, reutilización de código y escalabilidad. Es crucial desarrollar APIs genéricas, extensibles y robustas que integren las mejores prácticas actuales en términos de seguridad, documentación automática, orquestación eficiente y soporte para cargas dinámicas. Solo así se podrá cerrar la brecha entre las capacidades técnicas disponibles y las necesidades reales de los entornos empresariales modernos.

# 3. Desarrollo del proyecto

## 3.1 Introducción

En el ámbito de la ingeniería de software, el concepto de arquitectura se entiende como la organización fundamental de un sistema y las relaciones entre sus componentes, así como los principios que guían su diseño y evolución (Bass et al., 2021). Estudios recientes destacan que una arquitectura bien definida no solo mejora la mantenibilidad y la escalabilidad, sino que también habilita una mayor resiliencia y capacidad de adaptación frente a cambios futuros (Wolff, 2023). Asimismo, investigaciones modernas enfatizan que las arquitecturas modulares y orientadas a servicios permiten gestionar de manera más efectiva la complejidad, promoviendo la cohesión interna y la separación de responsabilidades (Richards & Ford, 2020).

El desarrollo de sistemas de gestión de datos operativos en tiempo real representa uno de los desafíos más significativos en la ingeniería de software contemporánea, particularmente cuando se requiere balancear escalabilidad, eficiencia y mantenibilidad. En la siguiente imagen (Ilustración 1) se presenta la arquitectura general del proyecto, la cual constituye la base estructural sobre la que se desarrolla todo el sistema. Esta representación permite visualizar de manera clara cómo se encuentran organizados los diferentes componentes, así como la forma en que interactúan entre sí desde una perspectiva arquitectónica.

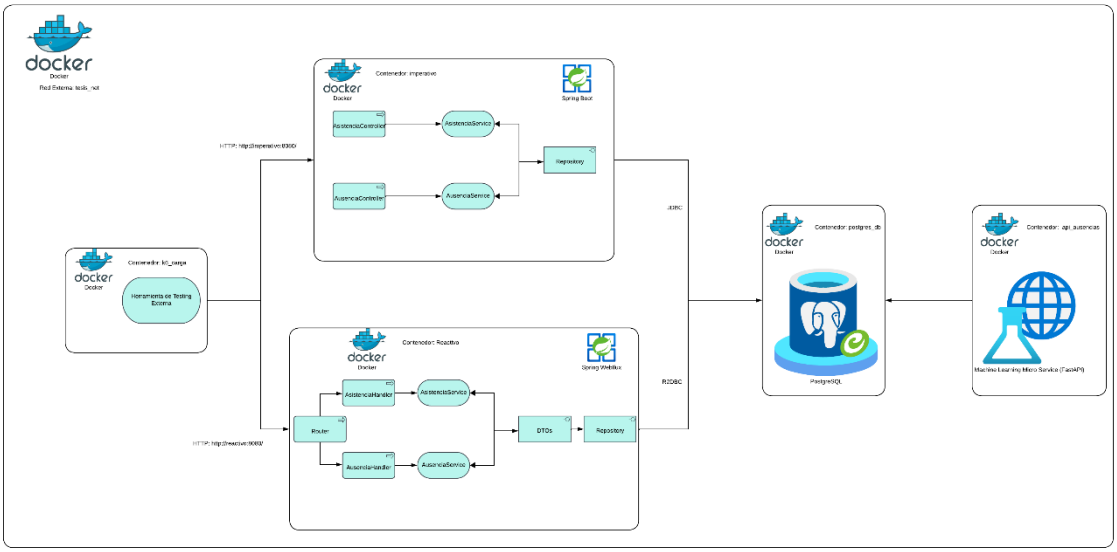


*Ilustración 1 Arquitectura del Sistema*

Esta sección presenta la implementación integral de un sistema de control de asistencias que constituye el núcleo experimental para realizar una comparativa objetiva entre dos paradigmas de programación fundamentales: el enfoque imperativo tradicional y el paradigma reactivo moderno. Estudios recientes destacan que la elección del paradigma influye directamente en la latencia, el manejo de concurrencia y la eficiencia bajo escenarios de alta carga (Richards & Ford, 2020; Wolff, 2023).

El análisis comienza con el diseño y modelado de la base de datos utilizando PostgreSQL 16, donde se establece una arquitectura robusta y normalizada que soporta tanto operaciones transaccionales tradicionales mediante JDBC como acceso no bloqueante a través de R2DBC. Investigaciones contemporáneas señalan que los modelos reactivos basados en IO no bloqueante permiten un uso más eficiente de recursos del sistema al reducir el *context switching* y mejorar el *throughput* (Šelajev & Winkler, 2022).

La implementación mediante contenedores Docker garantiza entornos aislados y reproducibles, facilitando las pruebas de rendimiento bajo condiciones controladas, una práctica ampliamente recomendada para entornos de experimentación científica en ingeniería de software moderna (Merkel, 2022).

Posteriormente, se detalla el desarrollo del sistema bajo el paradigma imperativo, implementando una arquitectura en capas con controladores REST, servicios de negocio y repositorios de datos, siguiendo principios de ingeniería de software para garantizar

escalabilidad y mantenibilidad. En contraste, la implementación reactiva utiliza Spring WebFlux con un enfoque funcional basado en routers y handlers, aprovechando flujos asíncronos no bloqueantes para maximizar la eficiencia bajo cargas concurrentes. Literatura reciente resalta que los frameworks reactivos basados en programación declarativa proporcionan mayor resiliencia y utilización de CPU en escenarios con miles de conexiones simultáneas (Banerjee et al., 2021).

Como componente innovador, se incorpora un sistema de Machine Learning que predice ausencias laborales mediante un modelo Random Forest entrenado con datos históricos, desplegado como servicio independiente mediante FastAPI y contenedores Docker. El uso de microservicios desacoplados para modelos de aprendizaje automático se ha consolidado como una práctica recomendada para mejorar la mantenibilidad y escalabilidad en arquitecturas empresariales modernas (Chollet, 2021).

Finalmente, se presenta una metodología exhaustiva de pruebas de rendimiento utilizando K6, diseñada para evaluar ambos paradigmas bajo escenarios que van desde operación normal hasta estrés masivo con 4,000 usuarios concurrentes, estableciendo las bases para un análisis comparativo cuantitativo que determine el impacto real de cada arquitectura en términos de rendimiento, escalabilidad y consumo de recursos. Según investigaciones recientes, las pruebas de estrés y carga son esenciales para evaluar comportamientos emergentes en arquitecturas reactivas y tradicionales bajo condiciones reales de producción (Tsigkanos et al., 2022).

Esta implementación integral no solo demuestra las capacidades técnicas de cada enfoque, sino que proporciona evidencia empírica valiosa para la toma de decisiones arquitectónicas en el desarrollo de sistemas empresariales de alta demanda.

## 3.2 Análisis, modelado e implementación de Base de Datos

El análisis, modelado e implementación de la base de datos es una parte importante en el diseño de cualquier sistema de TI, especialmente en sistemas que dan gestión de datos operativos en tiempo real y que dependen de una arquitectura que sea escalable y eficiente. En el contexto de este proyecto, orientado a la comparativa entre el

paradigma de la programación imperativa y programación reactiva, fue necesario establecer una base de datos robusta, normalizada y alineada con lo que cumpla con el objetivo funcional, y no funcionales.

Para llevar a cabo este propósito se utilizó PostgreSQL 16, una de las versiones más maduras y avanzadas del motor de bases de datos relacional PostgreSQL. La elección de la base de datos se fundamentó en su capacidad para gestionar grandes volúmenes de información, su compatibilidad con modelos transaccionales complejas, su eficiencia en consultas y su integración con tecnologías como R2DBC, la cual es necesaria para la implementación reactiva. PostgreSQL ofrece características que resultaron esenciales para garantizar la integridad, consistencia y rendimiento del sistema.

Para la implementación de la base de datos se utilizó contenedores Docker, lo que permitió desplegar entornos aislados, reproducibles y portables. Esta implementación permitió la gestión del ciclo de vida de la base de datos, eliminó problemas de compatibilidad entre entornos y aseguró que la ejecución del sistema fuera consistente durante las fases de desarrollo, pruebas y análisis de rendimiento. Además, Docker permitió automatizar la configuración de PostgreSQL 16, optimizar la estructura inicial de tablas, aplicar scripts de inicialización y garantizar un entorno para las pruebas y simulaciones realizadas en la etapa experimental.

En conjunto, esta sección describe de manera detallada el diseño del modelo conceptual y lógico, normalizar las entidades, definir las relaciones clave y finalmente implementar la base de datos en un entorno contenerizado. Estos elementos constituyen la base estructural y permiten comparar de manera objetiva el impacto del acceso relacional tradicional mediante JDBC frente al acceso no bloqueante de R2DBC.El modelo de datos diseñado para el sistema está compuesto por cinco entidades principales:

- Empleado
- Departamento
- Turno
- Asistencia
- Ausencia

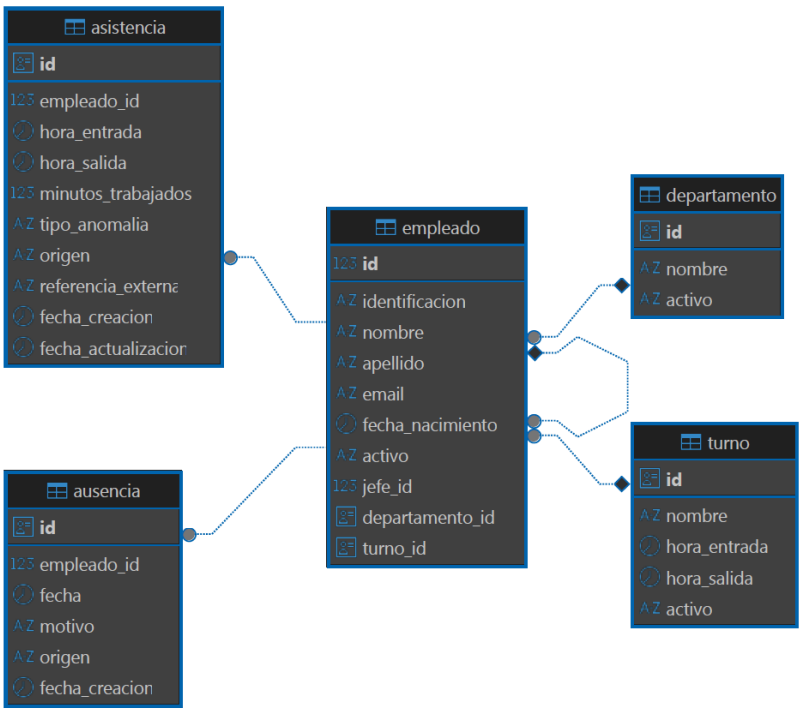


*Ilustración 2 Diseño de la Base de Datos*

La entidad empleada constituye el eje principal del sistema, ya que todas las operaciones se relacionan directamente con los trabajadores registrados. Incluye atributos esenciales como identificación, nombre, apellido, email, fecha_nacimiento y un estado lógico de activo, lo que permite controlar la vigencia de cada registro sin eliminar datos históricos.

Además, incorpora tres claves foráneas:

- ✓ jefe_id: Permite modelar jerarquías internas o estructuras organizacionales.
- ✓ departamento_id: Vincula al empleado con la entidad departamento.
- ✓ turno_id: Asocia a cada empleado con su horario laboral vigente.

Esta estructura facilita la segmentación por áreas, control de horarios, análisis por dependencias y posterior aplicación de modelos predictivos basados en patrones de asistencia.

La entidad departamento cumple la función de categorizar a los empleados dentro de áreas administrativas u operativas. Se compone de un id, un nombre y un campo activo, lo que permite mantener un historial de departamentos sin eliminar información relevante. Su relación con la entidad empleado permite un análisis organizacional directo, clave para estudios comparativos y evaluaciones de rendimiento por áreas.

La entidad turno almacena la configuración horaria asociada al empleado. Incluye los atributos hora_entrada, hora_salida y el indicador activo. Este componente es fundamental para calcular correctamente la asistencia, los minutos trabajados, las anomalías y la detección automática de ausencias. La relación de uno a muchos, entre turno y empleado permite reutilizar la misma definición de turno para múltiples trabajadores, manteniendo consistencia y reduciendo la duplicación de información.

La entidad asistencia representa el registro diario de entrada y salida de cada empleado. Incluye datos críticos como:

- ✓ hora_entrada
- ✓ hora_salida
- ✓ minutos_trabajados
- ✓ tipo_anomalia (tardanza, marcación faltante, salida temprana, etc.)
- ✓ origen (API, marcación física, integración externa)
- ✓ referencia_externa (códigos del sistema externo o del biométrico)

Además, contiene fecha_creacion y fecha_actualizacion para auditoría.

La relación con empleado es de tipo uno a muchos: un empleado puede tener múltiples registros de asistencia a lo largo del tiempo. Esto permite realizar análisis históricos, aplicar algoritmos de predicción y evaluar tiempo efectivo de trabajo.

La entidad ausencia complementa al registro de asistencia permitiendo registrar días en los que el empleado no asistió. Incluye los campos:

- ✓ fecha de la ausencia
- ✓ motivo (vacación, permiso, enfermedad, falta injustificada)
- ✓ origen
- ✓ fecha_creacion.

Al igual que asistencia, mantiene una relación de muchos a uno con empleado, permitiendo construir un historial completo de ausencias. Esta entidad es muy importante para el modelo predictivo, ya que proporciona ejemplos reales de comportamientos y patrones que pueden ser detectados o anticipados por el sistema.

El modelado lógico consta de relaciones y Cardinalidades:

- ✓ Departamento 1:N Empleado
    - Un departamento puede tener muchos empleados; cada empleado pertenece solo a un departamento.

- ✓ Turno 1:N Empleado
    - Un turno puede asignarse a múltiples empleados; cada empleado posee un turno vigente.
- ✓ Empleado 1:N Asistencia
    - Un empleado puede generar múltiples registros de asistencia a lo largo del tiempo.
- ✓ Empleado 1:N Ausencia
    - Un empleado puede registrar múltiples ausencias.
- ✓ Empleado (jefe) 1:N Empleado (subordinado)

La clave “jefe_id” permite modelar estructuras jerárquicas internas.

El modelo Físico de mi base de datos tiene tipo de datos como:

- ✓ Identificadores generales: UUID para asegurar unicidad global en entornos distribuidos.
- ✓ Campos de texto: VARCHAR, adecuado para nombres, correos, descripciones y motivos.
- ✓ Valores booleanos: BOOLEAN para el campo activo y banderas de estado.
- ✓ Fechas y tiempos: DATE, TIMESTAMP y TIME, necesarios para cálculos de horas trabajadas y análisis temporal.
- ✓ Campos numéricos: INTEGER para minutos trabajados y otras métricas.

Claves Primarias, foráneas y restricciones:

- ✓ empleado.departamento_id = departamento(id)
- ✓ empleado.turno_id = turno(id)
- ✓ empleado.jefe_id = empleado(id)
- ✓ asistencia.empleado_id = empleado(id)
- ✓ ausencia.empleado_id = empleado(id)

Adicionalmente, se definieron restricciones NOT NULL, reglas de unicidad para campos como la identificación del empleado y valores por defecto para estados lógicos.

La implementación física de la base de datos no se limitó únicamente a la creación de tablas y relaciones; también se incluyó la definición de extensiones, restricciones avanzadas, índices especializados, funciones y triggers orientados a garantizar integridad, optimización del rendimiento y coherencia temporal de los registros. Todo este conjunto de elementos permitió que la base de datos en PostgreSQL 16 operara

correctamente bajo altos volúmenes de carga, especialmente durante las simulaciones con 4000 empleados y miles de registros diarios de asistencia en los microservicios imperativo (JDBC) y reactivo (R2DBC).

Se incorporaron dos extensiones nativas de PostgreSQL:

- ✓ pgcrypto: utilizada para la generación de identificadores UUID mediante la función gen_random_uuid().
- ✓ btree_gist: necesaria para implementar restricciones de exclusión con operadores GIST, fundamentales para el control de solapamientos de horarios.

Estas extensiones proporcionan capacidades que no están disponibles en la instalación básica del motor, habilitando funcionalidades avanzadas en integridad y modelado temporal.

```
CREATE EXTENSION IF NOT EXISTS btree_gist;
CREATE EXTENSION IF NOT EXISTS pgcrypto;
```

*Código 1 Extensiones utilizadas de PostgreSQL*

La entidad departamento se modeló como una tabla sencilla que contiene un identificador único basado en UUID, el nombre del departamento y un estado lógico que indica si se encuentra activo o no. Su propósito es clasificar organizacionalmente a los empleados, facilitando análisis por áreas, unidades o divisiones administrativas. La restricción UNIQUE en el campo nombre garantiza la no duplicidad, mientras que el campo activo se valida mediante una restricción CHECK.

```
CREATE TABLE IF NOT EXISTS departamento (
  id      uuid PRIMARY KEY DEFAULT gen_random_uuid(),
  nombre  text NOT NULL UNIQUE,
  activo  text NOT NULL CHECK (activo IN ('ACTIVE', 'INACTIVE'))
);
```

*Código 2 Creación de la Tabla Departamento*

La tabla turno define los distintos horarios laborales asignables a los empleados. Incluye un identificador UUID único, un nombre descriptivo del turno y los campos hora_entrada y hora_salida, que permiten calcular las horas efectivas de trabajo y detectar anomalías temporales. El atributo activo permite gestionar turnos históricos

sin eliminarlos físicamente. La estructura asegura que los turnos puedan reutilizarse por múltiples empleados y sean configurables según necesidades operativas.

```
CREATE TABLE IF NOT EXISTS turno (
  id            uuid PRIMARY KEY DEFAULT gen_random_uuid(),
  nombre        text NOT NULL UNIQUE,
  hora_entrada  time NOT NULL,
  hora_salida   time NOT NULL,
  activo        text NOT NULL CHECK (activo IN ('ACTIVE', 'INACTIVE'))
);
```

*Código 3 Creación de la Tabla Turno*

La entidad empleado constituye el núcleo del sistema y contiene información identificatoria, personal y administrativa del trabajador. A diferencia de otras tablas, utiliza una secuencia numérica incremental (empleado_seq) en lugar de UUID, lo cual permite un manejo más eficiente en grandes volúmenes de datos. Incluye claves foráneas para modelar relaciones jerárquicas (jefe_id) y organizacionales (departamento_id, turno_id). Los índices creados optimizan consultas frecuentes por departamento, turno o estado del trabajador, mejorando el rendimiento en escenarios de carga masiva o filtros operativos.

```
CREATE SEQUENCE IF NOT EXISTS empleado_seq START 1;

CREATE TABLE IF NOT EXISTS empleado (
  id               numeric PRIMARY KEY DEFAULT nextval('empleado_seq'),
  identificacion   text UNIQUE,
  nombre           text NOT NULL,
  apellido         text NOT NULL,
  email            text UNIQUE,
  fecha_nacimiento date NOT NULL,
  activo           text NOT NULL CHECK (activo IN ('ACTIVE','INACTIVE')),
  jefe_id          numeric REFERENCES empleado(id) ON DELETE SET NULL,
  departamento_id  uuid REFERENCES departamento(id),
  turno_id         uuid REFERENCES turno(id)
);

CREATE INDEX IF NOT EXISTS idx_empleado_departamento ON
empleado(departamento_id);
CREATE INDEX IF NOT EXISTS idx_empleado_turno        ON empleado(turno_id);
CREATE INDEX IF NOT EXISTS idx_empleado_activo       ON empleado(activo);
```

*Código 4 Creación de la Tabla, Secuencia e Índice de Empleado*

La tabla asistencia almacena los registros de entrada y salida del empleado. El diseño incluye un identificador UUID, la referencia al empleado, timestamps de entrada y salida, y un campo calculado (GENERATED ALWAYS AS) para determinar automáticamente los minutos trabajados a partir de la diferencia entre hora_salida y hora_entrada. Además, incorpora campos como tipo_anomalia, origen y referencia_externa para auditorías y análisis internos.

Se añadieron índices para optimizar consultas por fecha, empleado y rangos temporales. El índice GIST permite aplicar restricciones de exclusión temporal, garantizando que un empleado no pueda registrar fichajes solapados.

```
CREATE TABLE IF NOT EXISTS asistencia (
  id                  uuid PRIMARY KEY DEFAULT gen_random_uuid(),
  empleado_id         numeric NOT NULL REFERENCES empleado(id),
  hora_entrada        timestamptz NOT NULL,
  hora_salida         timestamptz,
  minutos_trabajados  int
    GENERATED ALWAYS AS (
      CASE
        WHEN hora_salida IS NOT NULL
          THEN floor(extract(epoch FROM (hora_salida - hora_entrada)) /
60)::int
        ELSE NULL
      END
    ) STORED,
  tipo_anomalia       text NOT NULL DEFAULT 'NINGUNA'
    CHECK (tipo_anomalia IN
('NINGUNA','TARDE','SALIDA_TEMPRANA','AUSENTE')),
  origen              text NOT NULL DEFAULT 'sistema_unico',
  referencia_externa  text,
  fecha_creacion      timestamptz NOT NULL DEFAULT now(),
  fecha_actualizacion timestamptz NOT NULL DEFAULT now()
);

CREATE UNIQUE INDEX IF NOT EXISTS uq_asistencia_unica
  ON asistencia (empleado_id, hora_entrada);

CREATE INDEX IF NOT EXISTS idx_asistencia_emp_entrada ON asistencia
(empleado_id, hora_entrada DESC);
CREATE INDEX IF NOT EXISTS idx_asistencia_entrada     ON asistencia
(hora_entrada DESC);
CREATE INDEX IF NOT EXISTS idx_asistencia_timerange   ON asistencia USING
GIST (empleado_id, tstzrange(hora_entrada, hora_salida, '[)'));
```

*Código 5 Creación de la Tabla, e Índices de Asistencia*

La tabla ausencia almacena los días en los que un empleado no registra entrada. Incluye la fecha de la ausencia, el motivo, el origen del registro y la fecha de creación. El índice

único sobre (empleado_id, fecha) evita duplicidades en el registro de ausencias por día. Esta tabla es fundamental para el sistema de predicción, ya que proporciona información histórica necesaria para identificar patrones de comportamiento.

```
CREATE TABLE IF NOT EXISTS ausencia (
  id uuid PRIMARY KEY DEFAULT gen_random_uuid(),
  empleado_id numeric NOT NULL REFERENCES empleado(id),
  fecha date NOT NULL,
  motivo text NOT NULL DEFAULT 'NO_FICHADO',
  origen text DEFAULT 'DETECTADO_SISTEMA',
  fecha_creacion timestamptz NOT NULL DEFAULT now()
);
CREATE UNIQUE INDEX IF NOT EXISTS uq_ausencia_empleado_fecha
  ON ausencia (empleado_id, fecha);
```

*Código 6 Creación de la Tabla Ausencia*

Esta validación utiliza un constraint EXCLUDE USING GIST para impedir que un mismo empleado tenga intervalos de tiempo superpuestos en sus registros de asistencia. El uso de rangos temporales (tstzrange) en combinación con operadores GIST permite garantizar integridad temporal y coherencia en la información, evitando duplicidades que afectarían cálculos posteriores.

```
DO $$
BEGIN
  IF NOT EXISTS (
    SELECT 1 FROM pg_constraint WHERE conname =
'ex_asistencia_sin_solapamiento'
  ) THEN
    EXECUTE '
      ALTER TABLE asistencia ADD CONSTRAINT ex_asistencia_sin_solapamiento
      EXCLUDE USING gist (
        empleado_id WITH =,
        tstzrange(hora_entrada, hora_salida, ''[)'') WITH &&
      )
      WHERE (hora_salida IS NOT NULL)
    ';
  END IF;
END $$;
```

*Código 7 Restricción de solapamiento en la tabla asistencia*

La función set_fecha_actualizacion() actualiza el campo fecha_actualizacion cada vez que se modifica un registro de la tabla asistencia. Esta lógica asegura trazabilidad

completa sobre las modificaciones efectuadas en los registros temporales, sin requerir intervención desde la aplicación.

```
CREATE OR REPLACE FUNCTION set_fecha_actualizacion()
RETURNS trigger LANGUAGE plpgsql AS $$
BEGIN
  NEW.fecha_actualizacion := now();
  RETURN NEW;
END $$;
```

*Código 8 Trigger para actualizar el campo fecha_actualizacion*

El trigger trg_asistencia_touch_update ejecuta la función anterior antes de cada operación de actualización. Esto garantiza que cualquier modificación sobre un registro de asistencia actualice automáticamente su timestamp, permitiendo auditorías precisas y consistencia temporal.

```
DO $$
BEGIN
  IF NOT EXISTS (
    SELECT 1 FROM pg_trigger WHERE tgname = 'trg_asistencia_touch_update'
  ) THEN
    CREATE TRIGGER trg_asistencia_touch_update
      BEFORE UPDATE ON asistencia
      FOR EACH ROW EXECUTE FUNCTION set_fecha_actualizacion();
  END IF;
END $$;
```

*Código 9 Trigger para verificar el cambio de fecha_actualizacion*

Los registros iniciales para la tabla turno establecen los cuatro tipos de horarios base utilizados en el sistema: diurno, vespertino, nocturno y extendido. Se utiliza ON CONFLICT DO NOTHING para evitar errores en caso de que los registros ya existan, asegurando idempotencia del script.

```
INSERT INTO turno(nombre, hora_entrada, hora_salida, activo)
VALUES
  ('Diurno',     '08:00', '16:00', 'ACTIVE'),
  ('Vespertino', '16:00', '00:00', 'ACTIVE'),
  ('Nocturno',   '22:00', '06:00', 'ACTIVE'),
  ('Extendido',  '09:00', '18:00', 'ACTIVE')
ON CONFLICT DO NOTHING;
```

*Código 10 Insert inicial de la Entidad Turno*

Se inserta el departamento “Tecnología” como unidad inicial. Este registro sirve como base para la creación de todos los empleados de prueba utilizados en las simulaciones y pruebas de carga.

```
INSERT INTO departamento(nombre, activo)
VALUES
  ('Tecnología', 'ACTIVE')
ON CONFLICT DO NOTHING;
```

*Código 11 Insert inicial de la entidad Departamento*

Este bloque DO en PL/pgSQL genera automáticamente 4000 empleados, distribuidos en cuatro turnos distintos. Se declara un conjunto de variables para obtener los identificadores reales del departamento y los turnos previamente creados. Luego, mediante cuatro ciclos FOR, se insertan empleados sintéticos con datos realistas listos para pruebas de carga (k6), simulaciones y experimentación del modelo imperativo vs reactivo.

```
DO $$
DECLARE
    depto_id uuid;
    turno_diurno uuid;
    turno_vespertino uuid;
    turno_nocturno uuid;
    turno_extendido uuid;
    i int;
BEGIN
    SELECT id INTO depto_id FROM departamento WHERE nombre = 'Tecnología'
LIMIT 1;
    SELECT id INTO turno_diurno FROM turno WHERE nombre = 'Diurno' LIMIT 1;
    SELECT id INTO turno_vespertino FROM turno WHERE nombre = 'Vespertino'
LIMIT 1;
    SELECT id INTO turno_nocturno FROM turno WHERE nombre = 'Nocturno'
LIMIT 1;
    SELECT id INTO turno_extendido FROM turno WHERE nombre = 'Extendido'
LIMIT 1;
    FOR i IN 1..1000 LOOP
        INSERT INTO empleado (
            identificacion,
            nombre,
            apellido,
            email,
            fecha_nacimiento,
            activo,
            departamento_id,
            turno_id
        )
        VALUES (
```

```
            'DNI_D_' || i,
            'Empleado_Diurno_' || i,
            'Apellido_' || i,
            'diurno_' || i || '@empresa.com',
            date '1985-01-01' + (i % 1000),
            'ACTIVE',
            depto_id,
            turno_diurno
        );
    END LOOP;
    FOR i IN 1..1000 LOOP
        INSERT INTO empleado (
            identificacion,
            nombre,
            apellido,
            email,
            fecha_nacimiento,
            activo,
            departamento_id,
            turno_id
        )
        VALUES (
            'DNI_V_' || i,
            'Empleado_Vespertino_' || i,
            'Apellido_' || i,
            'vespertino_' || i || '@empresa.com',
            date '1985-01-01' + (i % 1000),
            'ACTIVE',
            depto_id,
            turno_vespertino
        );
    END LOOP;
    FOR i IN 1..1000 LOOP
        INSERT INTO empleado (
            identificacion,
            nombre,
            apellido,
            email,
            fecha_nacimiento,
            activo,
            departamento_id,
            turno_id
        )
        VALUES (
            'DNI_N_' || i,
            'Empleado_Nocturno_' || i,
            'Apellido_' || i,
            'nocturno_' || i || '@empresa.com',
            date '1985-01-01' + (i % 1000),
            'ACTIVE',
            depto_id,
            turno_nocturno
        );
    END LOOP;
    FOR i IN 1..1000 LOOP
        INSERT INTO empleado (
            identificacion,
            nombre,
            apellido,
```

```
            email,
            fecha_nacimiento,
            activo,
            departamento_id,
            turno_id
        )
        VALUES (
            'DNI_E_' || i,
            'Empleado_Extendido_' || i,
            'Apellido_' || i,
            'extendido_' || i || '@empresa.com',
            date '1985-01-01' + (i % 1000),
            'ACTIVE',
            depto_id,
            turno_extendido
        );
    END LOOP;
END $$;
```

*Código 12 Script inicial de carga de Empleados*

La implementación física de la base de datos no se limitó únicamente a la creación de tablas y relaciones; también se incluyó la definición de extensiones, restricciones avanzadas, índices especializados, funciones y triggers orientados a garantizar integridad, optimización del rendimiento y coherencia temporal de los registros. Todo este conjunto de elementos permitió que la base de datos en PostgreSQL 16 operara correctamente bajo altos volúmenes de carga, especialmente durante las simulaciones con 4000 empleados y miles de registros diarios de asistencia en los microservicios imperativo (JDBC) y reactivo (R2DBC).

El archivo docker-compose.yml define un único servicio llamado postgres_db, basado en la imagen oficial postgres:16. Este servicio configura el motor con un usuario, contraseña y base de datos iniciales, además de exponer el puerto interno 5432 hacia el puerto 5433 del sistema anfitrión, con el fin de evitar conflictos con instalaciones locales de PostgreSQL.

```
services:
  postgres_db:
    image: postgres:16
    container_name: postgres_db
    environment:
      POSTGRES_USER: tesis_admin
      POSTGRES_PASSWORD: tesis
      POSTGRES_DB: reactiva_bd
    ports:
      - "5433:5432"
```

*Código 13 Servicio PostgreSQL y configuración de Base de Datos*

El volumen pgdata almacena físicamente todos los datos del motor, replicando el comportamiento de una instalación real de PostgreSQL. Esto permite mantener registros de asistencia, empleados, ausencias e índices aún después de reiniciar el contenedor.

```
volumes:
      - pgdata:/var/lib/postgresql/data
```

*Código 14 Volumen de mi Contenedor*

Característica relevante del despliegue es la ejecución automática del script init.sql ubicado en el directorio del proyecto, esto con el motivo de poder crear toda mi base al iniciar el contenedor:

```
- ./init.sql:/docker-entrypoint-initdb.d/init.sql:ro
```

*Código 15 Ejecución automática de mi script inicial*

Mecanismo de "healthcheck" que comprueba si PostgreSQL está listo para aceptar conexiones:

```
healthcheck:
  test: ["CMD-SHELL", "pg_isready -U tesis_admin -d reactiva_bd"]
  interval: 5s
  timeout: 3s
  retries: 10
```

*Código 16 Validación de Disponibilidad del Servicio*

Esta red es compartida por los microservicios desarrollados en Java, lo cual permite que todos los componentes se comuniquen mediante nombres de servicio (DNS interno de Docker) en lugar de direcciones IP estáticas.

```
networks:
  tesis_net:
    external: true
```

*Código 17 Integración a la red externa*

## 3.3 Desarrollo del sistema en Imperativo y Reactivo

En esta sección se presenta el proceso de desarrollo del sistema de control de asistencias bajo dos enfoques arquitectónicos diferentes: el paradigma imperativo y el paradigma reactivo. Ambos modelos fueron implementados de manera independiente, replicando las mismas funcionalidades y estructura de negocio con el fin de garantizar condiciones equivalentes para su posterior evaluación comparativa. Este enfoque dual permite analizar de manera rigurosa las características operativas de cada paradigma, así como su comportamiento ante distintos niveles de carga y concurrencia. El desarrollo imperativo se construyó utilizando Spring Boot y Spring MVC, empleando un modelo tradicional basado en hilos bloqueantes y un flujo de ejecución secuencial. El desarrollo reactivo fue implementado mediante Spring WebFlux y Project Reactor, adoptando un modelo no bloqueante orientado a eventos que permite manejar flujos asincrónicos de datos mediante las abstracciones Mono y Flux.

En la programación Imperativa y Reactiva necesitan en sus estructuras, la forma de gestionar solicitudes, validar datos y comunicarse con la capa de servicios.

### Programación Imperativa: Controladores

Para esto la clase AsistenciaController y AusenciaController, representa la capa de exposición (API REST) del sistema imperativo desarrollado con Spring MVC. Su responsabilidad principal es recibir solicitudes HTTP, procesar los valores de entrada y delegar la operación correspondiente a la capa de servicio.

La clase AsistenciaController utiliza anotaciones como @RestController y @RequestMapping para definir los endpoints disponibles dentro del contexto /api/asistencia. La clase expone dos operaciones principales:

- ✓ fichar entrada: Construye los datos recibidos, los transforma y los envía a AsistenciaService.ficharEntrada, devolviendo una respuesta HTTP 200 OK en caso de éxito o códigos 400/500 ante errores.

- ✓ fichar salida: Sigue un patrón similar, delegando la lógica al servicio correspondiente. Esta clase funciona gestionando cada solicitud de forma síncrona y bloqueante.

Ambas reciben un cuerpo JSON genérico que es manualmente interpretado y transformado en los parámetros necesarios. A diferencia del enfoque reactivo, donde existe validación declarativa con DTOs, en el imperativo la validación se realiza mediante bloques try-catch, característicos del modelo bloqueante.

```
package ec.edu.ups.controller;

import ec.edu.ups.entity.Asistencia;
import ec.edu.ups.service.AsistenciaService;

import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.time.OffsetDateTime;
import java.util.Map;

@RestController
@RequestMapping("/api/asistencia")
public class AsistenciaController {

	private final AsistenciaService asistenciaService;

	public AsistenciaController(AsistenciaService asistenciaService) {
		this.asistenciaService = asistenciaService;
	}

	@PostMapping("/entrada")
	public ResponseEntity<Asistencia> ficharEntrada(@RequestBody
Map<String, Object> body) {
		try {
			Object empleadoIdObj = body.get("empleadoId");
			Long empleadoId = (empleadoIdObj instanceof Number) ?
((Number) empleadoIdObj).longValue()
					: Long.parseLong(empleadoIdObj.toString());

			OffsetDateTime horaEntrada =
OffsetDateTime.parse(body.get("horaEntrada").toString());
			String origen = body.getOrDefault("origen",
"api").toString();
			String referenciaExterna =
body.get("referenciaExterna") != null ?
body.get("referenciaExterna").toString()
					: null;

			Asistencia asistencia =
asistenciaService.ficharEntrada(empleadoId, horaEntrada, origen,
referenciaExterna);
			return ResponseEntity.ok(asistencia);
		} catch (IllegalArgumentException e) {
```

```
                    return ResponseEntity.badRequest().build();
              } catch (Exception e) {
                    e.printStackTrace();
                    return ResponseEntity.internalServerError().build();
              }
       }

       @PostMapping("/salida")
       public ResponseEntity<?> ficharSalida(@RequestBody Map<String,
Object> body) {
              try {
                    Object empleadoIdObj = body.get("empleadoId");
                    Long empleadoId = (empleadoIdObj instanceof Number) ?
((Number) empleadoIdObj).longValue()
                                 : Long.parseLong(empleadoIdObj.toString());

                    OffsetDateTime horaSalida =
OffsetDateTime.parse(body.get("horaSalida").toString());
                    Asistencia asistencia =
asistenciaService.ficharSalida(empleadoId, horaSalida);
                    return ResponseEntity.ok(asistencia);

              } catch (IllegalArgumentException e) {
                    return
ResponseEntity.badRequest().body(e.getMessage());
              } catch (Exception e) {
                    e.printStackTrace();
                    return ResponseEntity.internalServerError().body("Error
al registrar salida");
              }
       }

}
```

*Código 18 Programación Imperativa del Controller de Asistencia*

Para el Controlador de Ausencias tenemos definido el endpoint /api/ausencia/detectar, el cual recibe parámetros de consulta mediante @RequestParam que incluyen:

- ✓ el rango de fechas (desde, hasta),
- ✓ la zona horaria (zone),
- ✓ y el identificador del empleado (id).

El controlador invoca al método calcularAusencias del AusenciaService, que ejecuta la lógica de detección sobre la base de datos. El método ejecuta la lógica de manera síncrona y retorna una lista completa de resultados antes de responder al cliente.

Finalmente, el controlador construye una respuesta HTTP tipo 200 OK con la lista de objetos AusenciaEvento generados por el servicio. En caso de errores, se retornan códigos HTTP apropiados.

```
package ec.edu.ups.controller;

import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import ec.edu.ups.service.AusenciaService;

import java.time.LocalDate;
import java.time.ZoneId;
import java.util.List;

@RestController
@RequestMapping("/api/ausencia")
public class AusenciaController {

	private final AusenciaService service;

	public AusenciaController(AusenciaService service) {
		this.service = service;
	}

	@PostMapping("/detectar")
	public ResponseEntity<List<AusenciaService.AusenciaEvento>>
detectarAusencias(@RequestParam String desde,
				@RequestParam String hasta, @RequestParam(defaultValue =
"Europe/Madrid") String zone, @RequestParam Long id) {

		LocalDate desdeDate = LocalDate.parse(desde);
		LocalDate hastaDate = LocalDate.parse(hasta);

		var eventos = service.calcularAusencias(desdeDate, hastaDate, ZoneId.of(zone), id);
		return ResponseEntity.ok(eventos);
	}

}
```

*Código 19 Programación Imperativa del Controller de Ausencia*

## Programación Reactiva: Handlers y Router

El AsistenciaHandler es el equivalente al Asistencia Controlador dentro del conjunto reactivo con Spring WebFlux y Project Reactor. A diferencia del modelo basado en anotaciones, este componente funciona mediante funciones puras que reciben un ServerRequest y retornan un Mono<ServerResponse>, lo que permite un procesamiento completamente no bloqueante.

El método ficharEntrada extrae el cuerpo de la petición utilizando bodyToMono, lo convierte en un DTO fuertemente tipado y lo envía a un validador automático. Posteriormente, el flujo continúa sin interrupciones mediante encadenamientos

flatMap, hasta ejecutar la lógica implementada en la capa de servicio. El método ficharSalida sigue el mismo patrón, promoviendo un "pipeline" reactivo continuo.

El método listarRango demuestra cómo los parámetros de consulta son transformados y enviados al servicio, retornando un Flux de resultados que WebFlux convierte automáticamente en un flujo JSON. Finalmente, el método privado validate utiliza jakarta.validation.Validator para aplicar validaciones declarativas y retornar errores en forma reactiva, sin lanzar excepciones convencionales.

Esta implementación completamente reactiva elimina bloqueos, evita hilos dedicados por solicitud y maximiza la eficiencia en escenarios de alta concurrencia.

```
package ups.edu.ec.reactivo_ws.api;

import jakarta.validation.Validator;
import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.server.ServerRequest;
import org.springframework.web.reactive.function.server.ServerResponse;
import reactor.core.publisher.Mono;
import ups.edu.ec.reactivo_ws.dto.AsistenciaDtos;
import ups.edu.ec.reactivo_ws.service.AsistenciaService;

import java.time.OffsetDateTime;

@Component
public class AsistenciaHandler {

        private final AsistenciaService service;
        private final Validator validator;

        public AsistenciaHandler(AsistenciaService service, Validator va-
lidator) {
                this.service = service;
                this.validator = validator;
        }

        public Mono<ServerResponse> ficharEntrada(ServerRequest req) {
                return req.bodyToMono(Asisten-
ciaDtos.FicharEntrada.class).flatMap(this::validate)
                                .flatMap(service::ficharEntrada).flat-
Map(ServerResponse.ok()::bodyValue);
        }

        public Mono<ServerResponse> ficharSalida(ServerRequest req) {
                return req.bodyToMono(AsistenciaDtos.FicharSal-
ida.class).flatMap(this::validate).flatMap(service::ficharSalida)
                                .flatMap(ServerResponse.ok()::bodyValue);
        }

        public Mono<ServerResponse> listarRango(ServerRequest req) {
                long empleadoId = Long.parseLong(req.query-
Param("empleadoId").orElseThrow());
```

```
                OffsetDateTime desde = OffsetDateTime.parse(req.query-
Param("desde").orElseThrow());
                OffsetDateTime hasta = OffsetDateTime.parse(req.query-
Param("hasta").orElseThrow());
                return ServerResponse.ok().body(service.listar-
Rango(empleadoId, desde, hasta), AsistenciaDtos.class);
        }

        private <T> Mono<T> validate(T dto) {
                var errors = validator.validate(dto);
                if (!errors.isEmpty()) {
                        return Mono.error(new IllegalArgumentException(er-
rors.iterator().next().getMessage()));
                }
                return Mono.just(dto);
        }
}
```

*Código 20 Programación Reactiva del Handler de Asistencia*

El AusenciaHandler es la contraparte del controlador imperativo. En este Handler, el método detectarAusencias recibe un ServerRequest, desde el cual extrae los parámetros de consulta utilizando un enfoque funcional. Los valores recibidos se convierten a su representación interna y se envían como argumentos al método reactivo calcularAusencias del AusenciaService. El servicio retorna un Flux<AusenciaEvento>, y este flujo es directamente enviado al cliente mediante ServerResponse.ok().body(...), permitiendo que los datos sean transmitidos de forma progresiva y eficiente.

El Handler también especifica el tipo de contenido (MediaType.APPLICATION_JSON) y construye la respuesta usando operadores reactivos como Mono y Flux, lo que habilita un pipeline totalmente asíncrono y no bloqueante.

```
package ups.edu.ec.reactivo_ws.api;

import org.springframework.http.MediaType;
import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.server.ServerRequest;
import org.springframework.web.reactive.function.server.ServerResponse;
import reactor.core.publisher.Mono;
import ups.edu.ec.reactivo_ws.service.AusenciaService;

import java.time.LocalDate;
import java.time.ZoneId;

@Component
public class AusenciaHandler {

       private final AusenciaService service;
```

```
	public AusenciaHandler(AusenciaService service) {
		this.service = service;
	}

	public Mono<ServerResponse> detectarAusencias(ServerRequest req) {
		LocalDate desde =
LocalDate.parse(req.queryParam("desde").orElseThrow());
		LocalDate hasta =
LocalDate.parse(req.queryParam("hasta").orElseThrow());
		ZoneId zone =
ZoneId.of(req.queryParam("zone").orElse("Europe/Madrid"));
		Long id = Long.parseLong(req.queryParam("id").orElseThrow());
		return
ServerResponse.ok().contentType(MediaType.APPLICATION_JSON)
				.body(service.calcularAusencias(desde, hasta,
zone, id), AusenciaService.AusenciaEvento.class);
	}
}
```

*Código 21 Programación Reactiva del Handler de Ausencia*

La clase Router define la estructura de rutas del sistema Reactivo utilizando el enfoque funcional de WebFlux. En lugar del clásico uso de anotaciones como @GetMapping o @PostMapping, esta clase emplea el patrón de enrutamiento programático mediante RouterFunctions.route(). Dentro del método routes, se construye un árbol de rutas bajo el prefijo /api, mapeando cada endpoint a su respectivo handler:

- ✓ POST /asistencia/entrada → AsistenciaHandler.ficharEntrada
- ✓ POST /asistencia/salida → AsistenciaHandler.ficharSalida
- ✓ GET /asistencia → AsistenciaHandler.listarRango
- ✓ POST /ausencia/detectar → AusenciaHandler.detectarAusencias

Este esquema desacopla completamente la capa HTTP de la capa lógica. De esta manera el Router actúa como el “mapa” del sistema reactivo.

```
package ups.edu.ec.reactivo_ws.api;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.reactive.function.server.RouterFunction;
import static
org.springframework.web.reactive.function.server.RouterFunctions.route;

@Configuration
public class Router {

	@Bean
	RouterFunction<?> routes(AsistenciaHandler a, AusenciaHandler aus) {
```

```
            return route().path("/api",
                        builder -> builder.POST("/asistencia/entrada",
a::ficharEntrada)
                                          .POST("/asistencia/salida",
a::ficharSalida).GET("/asistencia", a::listarRango)
                                          .POST("/ausencia/detectar",
aus::detectarAusencias))
                        .build();
      }
}
```

*Código 22 Programación Reactiva del Router del sistema*

## Programación Imperativa: Servicio de Asistencia

Implementa el patrón de verificación de invariantes mediante consulta de solapamiento temporal. La ventana de 1 minuto (entrada.plusMinutes(1)) constituye un mecanismo de tolerancia para operaciones cercanas en el tiempo. La transacción @Transactional garantiza atomicidad en la validación y persistencia.

```
@Service
public class AsistenciaService {

    private final AsistenciaRepository asistenciaRepo;

    public AsistenciaService(AsistenciaRepository asistenciaRepo) {
        this.asistenciaRepo = asistenciaRepo;
    }

    @Transactional
    public Asistencia ficharEntrada(Long empleadoId, OffsetDateTime
horaEntrada,
                                    String origen, String referen-
ciaExterna) {
        OffsetDateTime entrada = horaEntrada;
        boolean existeSolape = asistenciaRe-po.existeSolape(empleadoId,
entrada, entrada.plusMinutes(1));
        if (existeSolape) {
            throw new IllegalArgumentException("Ya existe un fichaje
solapado en ese horario.");
        }

        Asistencia nuevo = new Asistencia();
        nuevo.setEmpleadoId(empleadoId);
        nuevo.setHoraEntrada(entrada);
        nuevo.setTipoAnomalia("NINGUNA");
        nuevo.setOrigen(origen == null ? origen : "api");
        nuevo.setReferenciaExterna(referenciaExterna);

        Asistencia guardado = asistenciaRepo.save(nuevo);
        System.out.println("guardado id=" + guardado.getId());
```

```
        return guardado;
    }
```

*Código 23 Programación Imperativa del servicio de Asistencia y método para fichar la entrada de la asistencia*

El algoritmo utiliza el patrón Repository Query Method para recuperar la jornada activa. El uso de Optional con orElseThrow() implementa el patrón Fail Fast, proporcionando retroalimentación inmediata ante estados inconsistentes.

```
    @Transactional
    public Asistencia ficharSalida(Long empleadoId, OffsetDateTime
horaSalida) {
        Asistencia actual = asistenciaRepo.findUltimaAbierta(empleadoId)
                .stream()
                .findFirst()
                .orElseThrow(() -> new IllegalArgumentException("No hay
jornada abierta para este empleado."));

        actual.setHoraSalida(horaSalida);
        Asistencia actualizado = asistenciaRepo.save(actual);
        return actualizado;
    }
}
```

*Código 24 Programación Imperativa del método para fichar la salida de la asistencia*

## Programación Reactiva: Servicio de Asistencia

La clase AsistenciaService constituye la capa de lógica de negocio encargada de gestionar todas las operaciones relacionadas con la marcación de asistencias dentro del microservicio reactivo. De igual manera se utiliza Mono y Flux para garantizar que cada operación se ejecute de manera no bloqueante. Este servicio consta de 3 métodos:

- El método ficharEntrada representa el proceso de registrar la entrada de un empleado en un momento determinado. Para evitar inconsistencias, la operación comienza verificando si existe un solapamiento con otro registro de asistencia utilizando el repositorio reactivo. Esta verificación se encadena mediante flatMap, lo que permite que el flujo continúe únicamente cuando la comprobación haya finalizado. Si se detecta un solapamiento, el método devuelve un Mono.error que interrumpe el flujo y comunica la anomalía. De lo contrario, se construye un nuevo objeto Asistencia y se persiste en la base de datos de forma reactiva a través de asistenciaRepo.save().

- El método ficharSalida implementa la lógica de cierre de jornada de un empleado. Este proceso inicia consultando la última marcación abierta mediante findUltimaJornadaAbierta, la cual devuelve un Mono que puede contener o no un registro. En caso de no existir una jornada abierta, se devuelve un error reactivo con switchIfEmpty, evitando inconsistencias como cerrar una jornada inexistente. Si se encuentra un registro válido, la operación continúa con la construcción de una nueva instancia de Asistencia donde se actualiza únicamente la hora de salida
- Por último, el método listarRango permite obtener todas las asistencias registradas por un empleado dentro de un periodo determinado. A diferencia de los métodos anteriores, esta operación retorna un Flux<Asistencia>, lo que posibilita transmitir múltiples elementos de forma progresiva y eficiente.

```
@Service
public class AsistenciaService {

	private final AsistenciaRepository asistenciaRepo;

	public AsistenciaService(AsistenciaRepository asistenciaRepo) {
		this.asistenciaRepo = asistenciaRepo;
	}

	@Transactional
	public Mono<Asistencia> ficharEntrada(AsistenciaDtos.FicharEntrada
req) {
		OffsetDateTime entrada = req.horaEntrada();
		OffsetDateTime fin = entrada.plusMinutes(1);

		return asistenciaRepo.existeSolape(req.empleadoId(), entrada,
fin)
				.flatMap(solape -> {
					if (Boolean.TRUE.equals(solape)) {
						return Mono
.error(new
IllegalArgumentException("Ya existe un fichaje solapado en ese horario."));
					}
					Asistencia nuevo = new Asistencia(null,
req.empleadoId(), entrada, null, null, "NINGUNA",
req.origen() == null ? "api" :
req.origen(), req.referenciaExterna(), null, null);
					return asistenciaRepo.save(nuevo);
				})
				.doOnNext(a -> System.out.println("✓ guardado
id=" + a.id()))
				.doOnError(err -> System.err.println("✕ Error
ficharEntrada: " + err.getMessage()));
	}
```

```
	@Transactional
	public Mono<Asistencia> ficharSalida(AsistenciaDtos.FicharSalida
req) {
		return
asistenciaRepo.findUltimaJornadaAbierta(req.empleadoId())
				.switchIfEmpty(Mono.error(new
IllegalArgumentException("No hay jornada abierta")))
				.flatMap(actual -> {
					Asistencia actualizado = new
Asistencia(actual.id(), actual.empleado_id(), actual.hora_entrada(),
							req.horaSalida(),
actual.minutos_trabajados(), actual.tipo_anomalia(), actual.origen(),
							actual.referencia_externa(),
actual.fecha_creacion(), null);
					return asistenciaRepo.save(actualizado);
				})
				.doOnError(err -> System.err.println("✕ Error
ficharSalida: " + err.getMessage()));
	}

	public Flux<Asistencia> listarRango(Long empleadoId, OffsetDateTime
desde, OffsetDateTime hasta) {
		return asistenciaRepo.findRango(empleadoId, desde, hasta);
	}
}
```

*Código 25 Programación Reactiva de mi servicio de Asistencia*

## Programación Imperativa: Servicio de Ausencia

Este servicio implementa un proceso batch de detección proactiva que itera sobre empleados y rangos temporales. La exclusión de fines de semana mediante DayOfWeek constituye una regla de negocio parametrizada. La complejidad algorítmica es O(n*m) donde n es el número de empleados y m el número de días.

```
@Service
public class AusenciaService {

    public record AusenciaEvento(Long empleadoId, LocalDate fecha, String
motivo,
                                 OffsetDateTime ventanaInicio, Offset-
DateTime ventanaFin) {
    }

    @Transactional
    public List<AusenciaEvento> calcularAusencias(LocalDate desde,
LocalDate hasta,
                                                  ZoneId zone, Long id) {
        List<AusenciaEvento> eventos = new ArrayList<>();
        var empleados = empleadoRepo.findByActivoAndId("ACTIVE", id);

        for (Empleado e : empleados) {
            if (e.getTurno() == null) continue;
```

```
            Turno t = e.getTurno();
            for (LocalDate fecha = desde; !fecha.isAfter(hasta); fe-cha =
fecha.plusDays(1)) {
                DayOfWeek diaSemana = fecha.getDayOfWeek();
                if (diaSemana == DayOfWeek.SATURDAY || diaSemana ==
DayOfWeek.SUNDAY) {
                    continue;
                }

                var ventana = calcularVentana(t, fecha, zone);
                boolean existeAsistencia = asistenciaRe-
po.existeSolape(e.getId(), ventana.start(), ventana.end());

                if (!existeAsistencia) {
                    ausenciaRepo.upsert(e.getId(), fecha, "NO_FICHADO",
"AUTOMATICO_SISTEMA");
                    eventos.add(new AusenciaEvento(e.getId(), fecha,
"NO_FICHADO",
                                    ventana.start(), ventana.end()));
                }
            }
        }
        return eventos;
    }
```

*Código 26 Programación Imperativa de mi servicio de Ausencia y método para calcular Ausencias*

El algoritmo maneja turnos que cruzan medianoche mediante el ajuste condicional (plusDays(1)). La conversión a OffsetDateTime preserva la información de zona horaria, esencial para sistemas multi-geográficos. El record Ventana implementa el patrón Value Object para representación inmutable de rangos temporales.

```
    private Ventana calcularVentana(Turno t, LocalDate fecha, ZoneId zone)
{
        var startZdt = ZonedDateTime.of(fecha, t.getHoraEntrada(), zone);
        var endZdt = ZonedDateTime.of(fecha, t.getHoraSalida(), zone);
        if (endZdt.isBefore(startZdt))
            endZdt = endZdt.plusDays(1);
        return new Ventana(startZdt.toOffsetDateTime(),
endZdt.toOffsetDateTime());
    }

    private record Ventana(OffsetDateTime start, OffsetDateTime end) {
    }
}
```

*Código 27 Programación Imperativa de mi método para calcular de turnos laborales*

## Programación Reactiva: Servicio de Ausencia

La clase AusenciaService constituye la capa de lógica de negocio encargada detectar, calcular y registrar automáticamente las ausencias de los empleados dentro del

microservicio reactivo. De igual manera se utiliza Mono y Flux para garantizar que cada operación se ejecute de manera no bloqueante. Este servicio consta de algunos métodos como:

- El método principal, calcularAusencias, recibe un rango de fechas, una zona horaria y un identificador de empleado, y ejecuta un proceso reactivo para determinar en qué días laborales un empleado debería haber marcado asistencia pero no lo hizo. El flujo inicia recuperando los empleados activos desde EmpleadoRepository y filtrando aquel cuyo identificador coincida con el parámetro. Luego, únicamente se procesan aquellos empleados que posean un turno asignado, garantizando que el cálculo de ausencias solo se aplique cuando existe información suficiente para determinar su jornada laboral. Por cada empleado, el servicio obtiene de manera reactiva su turno mediante TurnoRepository y delega el procesamiento detallado al método procesarEmpleado, aprovechando concurrencia controlada para optimizar rendimiento.
- El método procesarEmpleado genera todos los días comprendidos en el rango solicitado y aplica un filtro para seleccionar únicamente los días laborables, excluyendo automáticamente sábados y domingos mediante la función esDiaLaborable. A continuación, para cada fecha generada, se ejecuta el método procesarDia, el cual constituye el núcleo de la lógica de detección de ausencias. Este flujo combina el cálculo de la ventana horaria del turno (ventana) con una verificación reactiva que determina si existe un registro de asistencia que se solape con el periodo esperado de trabajo. Si no se detecta ningún solapamiento, el sistema considera que existe una ausencia y ejecuta un *upsert* reactivo sobre AusenciaRepository, garantizando que la ausencia sea insertada o actualizada de manera atómica.
- La construcción de la ventana horaria se realiza en el método ventana, el cual transforma la hora de entrada y salida del turno en valores OffsetDateTime ajustados a la zona horaria especificada. Este método también contempla turnos nocturnos que cruzan la medianoche, ajustando correctamente la fecha de fin cuando la hora de salida es anterior a la de entrada.

```
@Service
public class AusenciaService {

	private final EmpleadoRepository empleadoRepo;
	private final TurnoRepository turnoRepo;
	private final AsistenciaRepository asistenciaRepo;
	private final AusenciaRepository ausenciaRepo;

	private static final int CONCURRENCIA = 4;

	public AusenciaService(EmpleadoRepository empleadoRepo,
TurnoRepository turnoRepo,
			AsistenciaRepository asistenciaRepo, AusenciaRepository
ausenciaRepo) {

		this.empleadoRepo = empleadoRepo;
		this.turnoRepo = turnoRepo;
		this.asistenciaRepo = asistenciaRepo;
		this.ausenciaRepo = ausenciaRepo;
	}

	public record AusenciaEvento(Long empleadoId, LocalDate fecha, String
motivo, OffsetDateTime ventanaInicio,
			OffsetDateTime ventanaFin) {
	}

	public Flux<AusenciaEvento> calcularAusencias(LocalDate desde,
LocalDate hasta, ZoneId zone, Long id) {

		return empleadoRepo.findByActivo("ACTIVE").filter(e ->
e.id().equals(id)) // Igual que findByActivoAndId
				.filter(e -> e.turno_id() != null) // Solo
empleados con turno asignado
				.flatMap(e -> turnoRepo.findById(e.turno_id())
					.flatMapMany(t -> procesarEmpleado(e,
t, desde, hasta, zone)), CONCURRENCIA);
	}

	private Flux<AusenciaEvento> procesarEmpleado(Empleado e, Turno t,
LocalDate desde, LocalDate hasta, ZoneId zone) {

		return dias(desde,
hasta).filter(this::esDiaLaborable).flatMap(fecha -> procesarDia(e, t,
fecha, zone),
				CONCURRENCIA);
	}

	private boolean esDiaLaborable(LocalDate fecha) {
		return fecha.getDayOfWeek() != DayOfWeek.SATURDAY &&
fecha.getDayOfWeek() != DayOfWeek.SUNDAY;
	}

	private Flux<AusenciaEvento> procesarDia(Empleado e, Turno t,
LocalDate fecha, ZoneId zone) {

		return ventana(t, fecha, zone)
				.flatMapMany(v ->
asistenciaRepo.existeSolape(e.id(), v.start(), v.end()).filter(existe ->
!existe)
```

```
                                        .flatMap(__                                 ->
ausenciaRepo.upsert(e.id(), fecha, "NO_FICHADO", "AUTOMATICO_SISTEMA")
                                                        .thenReturn(new
AusenciaEvento(e.id(), fecha, "NO_FICHADO", v.start(), v.end())))));
	}

	private Flux<LocalDate> dias(LocalDate desde, LocalDate hasta) {
		int days = (int) ChronoUnit.DAYS.between(desde, hasta) + 1;
		return Flux.range(0, days).map(desde::plusDays);
	}

	private record Ventana(OffsetDateTime start, OffsetDateTime end) {
	}

	private Mono<Ventana> ventana(Turno t, LocalDate fecha, ZoneId zone)
{
		var startZdt = ZonedDateTime.of(fecha, t.hora_entrada(), zone);
		var endZdt = ZonedDateTime.of(fecha, t.hora_salida(), zone);

		if (endZdt.isBefore(startZdt)) {
			endZdt = endZdt.plusDays(1);
		}

		return	Mono.just(new	Ventana(startZdt.toOffsetDateTime(),
endZdt.toOffsetDateTime()));
	}
}
```

*Código 28 Programación Reactiva de mi servicio de Ausencia*

## Programación Imperativa: Interfaz de Repositorio

La implementación utiliza operadores nativos de PostgreSQL para rangos temporales (tstzrange). El operador && (overlap) proporciona detección eficiente de solapamientos. La consulta findUltimaAbierta emplea ordenamiento descendente para minimizar el tiempo de búsqueda de jornadas activas.

```
@Repository
public interface AsistenciaRepository extends JpaReposi-tory<Asistencia,
Long> {

    List<Asistencia> findByEmpleadoId(Long empleadoId);

    @Query("SELECT a FROM Asistencia a WHERE a.empleadoId = :empleadoId "
            + "AND (a.horaEntrada BETWEEN :desde AND :hasta "
            + "OR a.horaSalida BETWEEN :desde AND :hasta)")
    List<Asistencia> findSolapadas(@Param("empleadoId") Long emplea-doId,
                                   @Param("desde") Date desde,
                                   @Param("hasta") Date hasta);

    @Query("SELECT a FROM Asistencia a WHERE a.empleadoId = :empleadoId AND
a.horaSalida IS NULL "
```

```
            + "ORDER BY a.horaEntrada DESC")
    List<Asistencia> findUltimaAbierta(@Param("empleadoId") Long
empleadoId);

    @Query(value = """
            select exists(
              select 1
              from asistencia a
              where a.empleado_id = :empleadoId
                and tstzrange(a.hora_entrada, a.hora_salida, '[)') &&
                    tstzrange(:inicio, :fin, '[)')
            )
            """, nativeQuery = true)
    boolean existeSolape(@Param("empleadoId") Long empleadoId,
                         @Param("inicio") OffsetDateTime inicio,
                         @Param("fin") OffsetDateTime fin);
}
```

*Código 29 Programación Imperativa de mi Repository de Asistencia*

La cláusula ON CONFLICT de PostgreSQL implementa el patrón UPSERT, garantizando operaciones atómicas de inserción/actualización. La función COALESCE establece valores por defecto a nivel de base de datos, mientras EXCLUDED referencia los valores del INSERT en conflicto durante la actualización.

```
@Repository
public interface AusenciaRepository extends JpaRepository<Ausencia, Long> {

    @Modifying
    @Query(value = """
            INSERT INTO ausencia (empleado_id, fecha, motivo, ori-gen)
            VALUES (:empleadoId, :fecha, COALESCE(:motivo, 'NO_FICHADO'),
                    COALESCE(:origen, 'DETECTADO_SISTEMA'))
            ON CONFLICT (empleado_id, fecha)
            DO UPDATE SET motivo = EXCLUDED.motivo, origen = EXCLUD-
ED.origen
            """, nativeQuery = true)
    void upsert(@Param("empleadoId") Long empleadoId,
                @Param("fecha") LocalDate fecha,
                @Param("motivo") String motivo,
                @Param("origen") String origen);
}
```

*Código 30 Programación Imperativa de mi Repository de Ausencia*

## Programación Reactiva: Interfaz de Repositorio

Esta interfaz pertenece a la capa de acceso a datos del microservicio reactivo para la entidad Asistencia. Extiende de R2dbcRepository, lo que permite aprovechar la infraestructura de persistencia reactiva provista por Spring Data R2DBC. Los métodos

definidos en esta interfaz retornan tipos reactivos como Flux y Mono, garantizando que todas las operaciones sobre la base de datos se ejecuten de forma no bloqueante

- findRango: Permite obtener todas las asistencias registradas de un empleado dentro de un intervalo determinado de fechas. Mediante una consulta SQL personalizada.
- findUltimaJornadaAbierta: Implementa una funcionalidad clave para validar si un empleado tiene una jornada sin cerrar. Esta operación consulta la asistencia más reciente cuyo campo hora_salida sea nulo, lo que indica que el empleado registró entrada, pero aún no marcó su salida del sistema.
- existeSolape: Consulta la base de datos para determinar si existe un solapamiento temporal entre un nuevo registro de asistencia y otros intervalos previamente almacenados. La consulta SQL utiliza operadores de rangos temporales.

```
public interface AsistenciaRepository extends R2dbcRepository<Asistencia,
Long> {

	@Query("""
			SELECT * FROM asistencia
			WHERE empleado_id = :empleadoId
			  AND hora_entrada >= :desde
			  AND (hora_salida <= :hasta OR hora_salida IS NULL)
			ORDER BY hora_entrada DESC
			""")
	Flux<Asistencia> findRango(Long empleadoId, OffsetDateTime desde,
OffsetDateTime hasta);

	@Query("""
			SELECT * FROM asistencia
			WHERE empleado_id = :empleadoId
			  AND hora_salida IS NULL
			ORDER BY hora_entrada DESC
			LIMIT 1
			""")
	Mono<Asistencia> findUltimaJornadaAbierta(Long empleadoId);

	@Query("""
			SELECT EXISTS (
			  SELECT 1 FROM asistencia
			  WHERE empleado_id = :empleadoId
			    AND tstzrange(hora_entrada, hora_salida, '[)') &&
tstzrange(:entrada, :salida, '[)')
			)
			""")
	Mono<Boolean> existeSolape(Long empleadoId, OffsetDateTime entrada,
OffsetDateTime salida);
}
```

*Código 31 Programación Reactiva de mi Repository de Asistencia*

La interfaz AusenciaRepository representa la capa de acceso a datos para la entidad Ausencia. Al extender ReactiveCrudRepository, esta clase se integra con la infraestructura de persistencia no bloqueante proveída por Spring Data R2DBC, permitiendo ejecutar operaciones sobre la base de datos PostgreSQL de manera reactiva mediante flujos Mono y Flux. El método upsert permite registrar una ausencia nueva para un empleado o actualizar una ausencia existente si ya existe un registro para la misma fecha. La cláusula ON CONFLICT aprovecha la restricción única definida en la base de datos, garantizando la integridad del dato y evitando la duplicación de registros de ausencia.

```
@Repository
public interface AusenciaRepository extends
ReactiveCrudRepository<Ausencia, Long> {

	@Query("""
			INSERT INTO ausencia (empleado_id, fecha, motivo,
origen)
			VALUES ($1, $2, COALESCE($3, 'NO_FICHADO'),
COALESCE($4, 'DETECTADO_SISTEMA'))
			ON CONFLICT (empleado_id, fecha) DO UPDATE
			   SET motivo = EXCLUDED.motivo,
			       origen = EXCLUDED.origen
			RETURNING id, empleado_id, fecha, motivo, origen,
fecha_creacion
			""")
	Mono<Ausencia> upsert(Long empleadoId, LocalDate fecha, String
motivo, String origen);
}
```

*Código 32 Programación Reactiva de mi Repository de Ausencia*

## Programación Imperativa: Configuración Base de Datos

En la arquitectura imperativa, la conexión a la base de datos PostgreSQL se gestiona mediante JDBC, utilizando lo definido en el archivo application.properties. Este archivo establece los parámetros esenciales para que Spring Boot inicialice el DataSource, cargue el driver correspondiente e integre Hibernate, bajo un modelo bloqueante. La propiedad spring.datasource.url especifica la ubicación del servidor PostgreSQL dentro

del entorno Docker, mientras que las credenciales permiten la autenticación dentro del contenedor postgres_db.

Además, se configura explícitamente el driver JDBC mediante spring.datasource.driver-class-name, lo cual garantiza la compatibilidad con PostgreSQL. En cuanto al comportamiento de Hibernate, se utiliza ddl-auto=none para evitar que el ORM intente modificar el esquema, ya que la estructura de la base de datos es gestionada mediante un script externo init.sql ejecutado automáticamente por Docker.

```
# Configuración para Docker
spring.datasource.url=jdbc:postgresql://postgres_db:5432/reactiva_bd
spring.datasource.username=tesis_admin
spring.datasource.password=tesis
spring.datasource.driver-class-name=org.postgresql.Driver

# JPA - IMPORTANTE: validate en lugar de update para usar tu schema
spring.jpa.hibernate.ddl-auto=none
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true

# Deshabilitar inicialización de datos de Spring Boot ya que usas init.sql
spring.sql.init.mode=never
spring.jpa.defer-datasource-initialization=false
```

*Código 33 Programación Imperativa de mi configuración de Base de Datos*

## Programación Reactiva: Configuración Base de Datos

A comparación con el imperativo, la arquitectura reactiva requiere una configuración diferente debido a la naturaleza no bloqueante del driver utilizado. En lugar de JDBC, el sistema se conecta a PostgreSQL mediante R2DBC, un protocolo diseñado específicamente para permitir acceso asincrónico y reactivo a bases de datos relacionales. El archivo application.yml define únicamente parámetros generales del servidor, configuración de logging para R2DBC y ajustes del Thread Pool para tareas internas. La configuración de la conexión no se define en el archivo YAML, sino en una clase dedicada llamada R2dbcConfig. Esto debido a la necesidad de inicializar manualmente el ConnectionFactory reactivo y el ConnectionPool, lo cual permite un mayor control sobre cómo se gestionan las conexiones dentro del contexto no bloqueante de WebFlux. Dentro de esta clase, se construye un

PostgresqlConnectionConfiguration estableciendo el host, puerto, base de datos y credenciales. Posteriormente, se crea un ConnectionPool configurado con parámetros como:

- ✓ tamaño inicial del pool (initialSize): número mínimo de conexiones preparadas
- ✓ tamaño máximo (maxSize): límite superior en escenarios de alta concurrencia
- ✓ tiempo máximo de inactividad (maxIdleTime)
- ✓ tiempo de vida máximo de cada conexión (maxLifeTime)
- ✓ reintentos de adquisición (acquireRetry)
- ✓ consulta de validación (validationQuery) para mantener conexiones saludables

Este enfoque permite mantener un flujo completamente reactivo desde el servicio hasta la base de datos, evitando bloqueos en operaciones I/O. Es decir, mientras JDBC mantiene un modelo por-hilo-bloqueante, R2DBC libera el hilo inmediatamente después de emitir la consulta, permitiendo que el servidor atienda miles de peticiones concurrentes con un consumo significativamente menor de recursos.

En conjunto, la combinación de application.yml y R2dbcConfig proporciona una configuración especializada que asegura que la arquitectura reactiva funcione sobre una base de datos relacional sin perder las propiedades clave del modelo reactivo: asincronía, eficiencia y no bloqueo.

```
package ups.edu.ec.reactivo_ws.config;

import io.r2dbc.pool.ConnectionPool;
import io.r2dbc.pool.ConnectionPoolConfiguration;
import io.r2dbc.postgresql.PostgresqlConnectionConfiguration;
import io.r2dbc.postgresql.PostgresqlConnectionFactory;
import io.r2dbc.spi.ConnectionFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import java.time.Duration;

@Configuration
public class R2dbcConfig {

	@Bean
	public ConnectionFactory connectionFactory() {
		PostgresqlConnectionConfiguration pgConfig =
PostgresqlConnectionConfiguration.builder().host("postgres_db")

	.port(5432).database("reactiva_bd").username("tesis_admin").password
("tesis")
```

```
                        .connectTimeout(Duration.ofSeconds(5)).build();
            var connectionFactory = new
PostgresqlConnectionFactory(pgConfig);
            ConnectionPoolConfiguration pool =
ConnectionPoolConfiguration.builder(connectionFactory).initialSize(5)

      .maxSize(20).maxIdleTime(Duration.ofSeconds(30)).maxLifeTime(Duratio
n.ofMinutes(30)).acquireRetry(2)
                        .validationQuery("SELECT 1").build();
            return new ConnectionPool(pool);
      }
}
```

*Código 34 Programación Reactiva de mi configuración de Base de Datos R2DBC*

```
spring:
  task:
    execution:
      pool:
        core-size: 4
        max-size: 8

server:
  port: 8080

management:
  endpoints:
    web:
      exposure:
        include: health,info,prometheus

logging:
  level:
    io.r2dbc.postgresql: INFO
    org.springframework.data.r2dbc: INFO
    org.springframework.transaction: INFO
```

*Código 35 Programación Reactiva de mi configuración de servidor, y ajustes del Thread Pool para tareas internas*

## 3.4 Desarrollo de Machine Learning

Machine Learning constituye uno de los elementos centrales del proyecto, ya que permite generar predicciones de ausencia laboral basadas en patrones temporales

extraídos del dataset construido a partir de los registros históricos de asistencia y ausencia. Para ello, se diseñó una arquitectura modular compuesta por un script de entrenamiento, un servicio de inferencia en FastAPI, un conjunto de dependencias definidas en requirements.txt y un despliegue contenerizado mediante Docker.

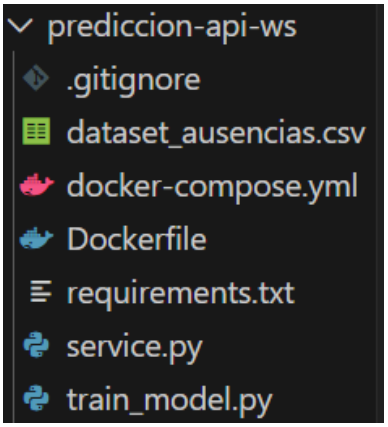


*Ilustración 3 Estructura del Proyecto prediccion-api-ws*

Esta estructura permite separar la programación que se dedica al entrenamiento del modelo, la persistencia, despliegue y consumo mediante una API.

El Dataset utilizado para entrenar el modelo se creó directamente desde la base de datos, mediante una consulta SQL avanzada que unifica información de asistencia, ausencia, fechas del calendario y características derivadas de la temporalidad. La consulta parte de una estrategia de generación de una matriz completa empleado × día, utilizando un CROSS JOIN entre todos los empleados y todas las fechas del rango 2022–2024. Posteriormente, se integran las asistencias normalizadas y las ausencias registradas para determinar el valor objetivo (label) del modelo:

- ✓ 0 = el empleado asistió
- ✓ 1 = el empleado faltó

Además, se generan tres características basadas en la fecha:

- ✓ día de la semana (dia_semana)
- ✓ mes (mes)
- ✓ día del año (dia_anio)

Estas variables permiten capturar patrones temporales que afectan a la probabilidad de ausencia, como fines de mes, lunes, viernes. Finalmente, se aplica un filtro para

descartar fines de semana sin asistencia real, con el fin de evitar ruido en el dataset. El resultado es un archivo dataset_ausencias.csv, que se utiliza directamente para el entrenamiento del modelo.

```
WITH fechas AS (
    -- Todas las fechas del rango que cubre tus datos (3 años)
    SELECT generate_series(
        DATE '2022-01-01',
        DATE '2024-12-31',
        INTERVAL '1 day'
    )::date AS fecha
),

empleados AS (
    SELECT DISTINCT empleado_id
    FROM asistencia
    UNION
    SELECT DISTINCT empleado_id FROM ausencia
),

empleado_fecha AS (
    -- Cruzamos cada empleado con cada día del calendario
    SELECT e.empleado_id, f.fecha
    FROM empleados e
    CROSS JOIN fechas f
),

asist AS (
    -- Normalizamos las asistencias
    SELECT
        empleado_id,
        hora_entrada::date AS fecha,
        hora_entrada,
        hora_salida,
        EXTRACT(EPOCH FROM (hora_salida - hora_entrada))/3600 AS
horas_trabajadas
    FROM asistencia
),

aus AS (
    -- Normalizamos las ausencias simuladas
    SELECT
        empleado_id,
        fecha,
        1 AS falta
    FROM ausencia
),

base AS (
    SELECT
        ef.empleado_id,
        ef.fecha,

        -- Label (0=asistencia, 1=falta)
        CASE
            WHEN aus.falta = 1 THEN 1
```

```
            WHEN asist.fecha IS NOT NULL THEN 0
            ELSE 0
        END AS falta,

        -- Datos de asistencia
        asist.hora_entrada,
        asist.hora_salida,
        asist.horas_trabajadas,

        -- Datos de ausencia
        aus.falta AS marca_falta
    FROM empleado_fecha ef
    LEFT JOIN asist ON asist.empleado_id = ef.empleado_id
                   AND asist.fecha = ef.fecha
    LEFT JOIN aus ON aus.empleado_id = ef.empleado_id
                 AND aus.fecha = ef.fecha
),

features AS (
    SELECT
        *,
        EXTRACT(DOW   FROM fecha) AS dia_semana,
        EXTRACT(MONTH FROM fecha) AS mes,
        EXTRACT(DOY   FROM fecha) AS dia_anio
    FROM base
)

SELECT *
FROM features
WHERE NOT (
    (EXTRACT(DOW FROM fecha) IN (0, 6))
    AND hora_entrada IS NULL
    AND falta = 0
)
ORDER BY empleado_id, fecha;
```

*Código 36 Consulta de Base de Datos para la creación de mi Dataset*

El script de Python “train_model.py” implementa el ciclo de entrenamiento utilizando la librería “Scikit-Learn”. El proceso primero lee el archivo dataset_ausencias.csv, y define las variables predictoras las cuales se escogieron por sus características de relevancia en la consulta y su bajo costo computacional, estas son:

- ✓ dia_semana
- ✓ mes
- ✓ dia_anio

```
df = pd.read_csv(DATASET_PATH)

    feature_cols = ["dia_semana", "mes", "dia_anio"]
    X = df[feature_cols]
    y = df["falta"]
```

*Código 37 Lectura de Dataset y definición de variables predictorias*

Mediante train_test_split con estratificación, se garantiza una distribución balanceada de clases. Esto quiere decir que debido a que el número de muestras que tenemos, donde lo normal es tener muchos registros de asistencias (= 0) y pocos registros de ausencias (= 1). Para la separación de los datos en conjuntos de entrenamiento y prueba se utilizó la función train_test_split con el parámetro stratify=y. La estratificación garantiza que la proporción de clases (asistencia vs ausencia) se mantenga constante en ambos subconjuntos. Esto es especialmente importante debido al desbalance natural del dataset como mencionamos anteriormente, donde las ausencias representan una fracción mucho menor que las asistencias.
De esta manera, se asegura que el modelo pueda aprender y evaluar correctamente ambas clases, evitando sesgos y evaluaciones incorrectas asociadas a separaciones aleatorias sin control.
Para el modelo de predicción se evaluaron múltiples alternativas, entre ellas regresión logística, redes neuronales, KNN, Naive Bayes y algoritmos de boosting. Sin embargo, se seleccionó un RandomForestClassifier debido a su robustez, capacidad de manejar relaciones no lineales, tolerancia al desbalance y facilidad de despliegue dentro de un microservicio.
El modelo se configuró con 200 árboles, lo cual proporciona estabilidad, una profundidad máxima de 8, que reduce el sobreajuste, y class_weight="balanced", que ajusta automáticamente la influencia de cada clase, mitigando el desbalance entre asistencias y ausencias. Al tratarse de un modelo basado en árboles, no requiere normalización de características y ofrece tiempos de inferencia muy bajos, lo que lo convierte en una opción ideal para ser desplegado en un servicio FastAPI dentro de un contenedor Docker.

```
model = RandomForestClassifier(
        n_estimators=200,
        max_depth=8,
        random_state=42,
        class_weight="balanced"
    )

model.fit(X_train, y_train)
y_pred = model.predict(X_test)
```

```
y_proba = model.predict_proba(X_test)[:, 1]
```

*Código 38 Configuración del modelo y entrenamiento*

El script genera métricas como:

- ✓ Reporte de clasificación (classification_report)
- ✓ Probabilidad ROC AUC

Esto permite validar el rendimiento del modelo previo a su despliegue.

Una vez entrenado, se guardan dos archivos:

- ✓ modelo_ausencias.joblib: Contiene el modelo entrenado
- ✓ feature_columns.joblib: Lista de columnas utilizadas
- ✓ Estos archivos son cargados posteriormente por el servicio FastAPI.

```
print("=== REPORT ===")
    print(classification_report(y_test, y_pred))
    try:
        auc = roc_auc_score(y_test, y_proba)
        print(f"AUC: {auc:.4f}")
    except ValueError:
        print("No se pudo calcular AUC (quizás todas las clases son iguales
en test).")

    # 7. Guardar modelo y metadata
    joblib.dump(model, "modelo_ausencias.joblib")
    joblib.dump(feature_cols, "feature_columns.joblib")
    print("Modelo y columnas guardados en modelo_ausencias.joblib y
feature_columns.joblib")
```

*Código 39 Reporte y Persistencia*

El archivo service.py implementa el servicio de inferencia encargado de procesar solicitudes de predicción de ausencias mediante FastAPI. Este servicio expone un endpoint REST que recibe la información del empleado y la fecha en la cual se desea estimar la probabilidad de ausencia, permitiendo que los microservicios desarrollados en Java (imperativo y reactivo) consuman el modelo de Machine Learning en tiempo real. Al iniciar el servicio, se cargan en memoria el modelo previamente entrenado y la lista de columnas utilizadas durante la fase de entrenamiento. Esta carga inicial evita la necesidad de reentrenar el modelo o recalcular configuraciones internas, garantizando respuestas rápidas y consistentes en cada solicitud.

Para gestionar la comunicación entre el cliente y el servicio, se definen dos clases mediante Pydantic:

- ✓ PrediccionRequest: Valida los datos de entrada
- ✓ PrediccionResponse: Estructura la respuesta que se devuelve al usuario.

Estas clases permiten asegurar que la información recibida y enviada cumpla con los formatos esperados, evitando errores comunes y facilitando la documentación automática del API. Antes de realizar la predicción, el servicio transforma la fecha enviada por el usuario en tres características relevantes: el día de la semana, el mes y el día del año. Estas variables permiten al modelo identificar patrones temporales que influyen en la probabilidad de ausencia. Posteriormente, estas características son organizadas dentro de un DataFrame que se alinea de forma exacta con el conjunto de columnas que el modelo espera, lo que garantiza una inferencia adecuada y coherente con el entrenamiento.

Finalmente, el servicio calcula la probabilidad de ausencia mediante el método predict_proba, obteniendo un valor entre 0 y 1. A partir de este valor, se determina si es probable que el empleado falte utilizando un umbral de decisión predefinido en 0.5. El resultado final incluye tanto la probabilidad como la decisión booleana y se devuelve en un objeto estructurado para su fácil interpretación por parte del cliente.

El servicio expone el endpoint:

- ➢ POST /api/predicciones/ausencia

El cual permite la integración directa con los microservicios del sistema, convirtiéndose en un componente central para la generación de predicciones automatizadas dentro del flujo completo de la plataforma.

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from datetime import datetime
import joblib
import pandas as pd

MODEL_PATH = "modelo_ausencias.joblib"
FEATURES_PATH = "feature_columns.joblib"

model = joblib.load(MODEL_PATH)
feature_cols = joblib.load(FEATURES_PATH)

app = FastAPI(
    title="Servicio de Predicción de Ausencias",
```

```
    description="Predice la probabilidad de falta de un empleado en una
fecha dada.",
    version="1.0.0"
)

class PrediccionRequest(BaseModel):
    empleadoId: int
    fecha: str

class PrediccionResponse(BaseModel):
    empleadoId: int
    fecha: str
    probabilidad_falta: float
    es_probable_que_falte: bool
    umbral: float = 0.5

@app.get("/health")
def health_check():
    return {"status": "ok"}

@app.post("/api/predicciones/ausencia", response_model=PrediccionResponse)
def predecir_ausencia(req: PrediccionRequest):
    # 1. Parsear fecha
    try:
        fecha_dt = datetime.strptime(req.fecha, "%Y-%m-%d").date()
    except ValueError:
        raise HTTPException(status_code=400, detail="Formato de fecha
inválido. Usa YYYY-MM-DD.")

    dia_semana = fecha_dt.weekday()
    mes = fecha_dt.month
    dia_anio = fecha_dt.timetuple().tm_yday

    data = {
        "dia_semana": dia_semana,
        "mes": mes,
        "dia_anio": dia_anio
    }

    df = pd.DataFrame([data])

    for col in feature_cols:
        if col not in df.columns:
            df[col] = 0
    df = df[feature_cols]

    proba = float(model.predict_proba(df)[0, 1])
    umbral = 0.5

    return PrediccionResponse(
        empleadoId=req.empleadoId,
        fecha=req.fecha,
        probabilidad_falta=proba,
        es_probable_que_falte=(proba >= umbral),
        umbral=umbral
    )
```

*Código 40 Programación de mi Aplicación de predicción en FastApi*

El servicio de Machine Learning se ejecuta de manera separada, portable y reproducible, además se creó un Dockerfile que define todo el entorno necesario para entrenar el modelo y exponer el servicio mediante FastAPI. Este Dockerfile se basa en la imagen ligera python:3.11-slim, la cual ofrece una plataforma optimizada para ejecutar aplicaciones Python con un consumo bajo. A partir de esta base, se establece variables para que los logs del servicio se impriman en tiempo real, útil para depuración y monitoreo del contenedor.

También se define el directorio de trabajo /app, donde se alojarán todos los archivos del proyecto. Antes de instalar las dependencias, se ejecuta una actualización de paquetes del sistema y se instalan herramientas esenciales como build-essential y gcc, necesarias para compilar ciertas librerías. Se eliminan los archivos temporales del sistema para reducir el tamaño final de la imagen.

Luego, se copian las dependencias definidas en el archivo requirements.txt y se instalan mediante pip install, utilizando la opción --no-cache-dir para evitar que el contenedor almacene archivos innecesarios. Se copian los archivos centrales del proyecto: el script de entrenamiento (train_model.py), el archivo del servicio (service.py) y el Dataset (dataset_ausencias.csv).

Uno de los aspectos más importantes del Dockerfile es que el modelo se entrena automáticamente durante el proceso de construcción de la imagen, mediante el comando “RUN python train_model.py”. Esto asegura que cualquier instancia del contenedor cuente siempre con el modelo actualizado sin intervención manual. Finalmente, se expone el puerto 8000, que será utilizado por FastAPI para servir la API REST, y se especifica el comando de inicio del contenedor, el cual ejecuta Uvicorn para desplegar el servicio.

```
FROM python:3.11-slim

ENV PYTHONUNBUFFERED=1

WORKDIR /app

RUN apt-get update && apt-get install -y \
    build-essential \
    gcc \
    && rm -rf /var/lib/apt/lists/*

COPY requirements.txt .
```

```
RUN pip install --no-cache-dir -r requirements.txt

COPY train_model.py .
COPY service.py .
COPY dataset_ausencias.csv .

RUN python train_model.py

EXPOSE 8000

CMD ["uvicorn", "service:app", "--host", "0.0.0.0", "--port", "8000"]
```

*Código 41 Dockerfile para la construcción de mi API de Predicciones*

En esta configuración del docker-compose, el servicio se llama ml_ausencias, y su contenedor asociado se nombra ml-ausencias-service. En el bloque build se especifica que el contexto de construcción es el directorio actual y que el archivo Dockerfile debe utilizarse como compilación. El servicio expone el puerto 8000, permitiendo que otros componentes del sistema puedan consumir el endpoint de predicción publicado por FastAPI. Tambien se define una política de reinicio restart: always, que garantiza que el contenedor se vuelva a ejecutar automáticamente en caso de fallos, reinicios del host o interrupciones inesperadas, aumentando así la disponibilidad del servicio. Finalmente, el contenedor se conecta a la red externa tesis_net, la misma utilizada por los otros servicios del ecosistema de la tesis.

```
version: "3.9"
services:
  ml_ausencias:
    container_name: ml-ausencias-service
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "8000:8000"
    restart: always
    networks:
      - tesis_net

networks:
  tesis_net:
    external: true
```

*Código 42 Configuración de mi docker compose para el despliegue de mi API*

# 4. Resultados y discusión

El presente capítulo expone y analiza los resultados empíricos obtenidos de la evaluación comparativa entre dos paradigmas arquitectónicos: la Arquitectura Imperativa (síncrona y basada en hilos) y la Arquitectura Reactiva (asíncrona y no bloqueante), mediante una metodología integral de pruebas de rendimiento diseñada para evaluar el sistema de control de asistencias bajo condiciones operativas realistas y extremas.

Esta evaluación se fundamenta en mediciones empíricas de rendimiento obtenidas con K6, una herramienta especializada en pruebas de carga, para medir métricas críticas que determinan la viabilidad técnica de la solución implementada. El análisis comparativo establece así una base objetiva para evaluar la escalabilidad, estabilidad y eficiencia de recursos de cada arquitectura.

```
services:

    k6:

    image: grafana/k6:latest

    container_name: k6-runner

    volumes:

    - .:/scripts

    - ./resultados:/resultados

    entrypoint: ["k6", "run", "/scripts/k6_test.js"]

    networks:

    - tesis_net

        extra_hosts:

      - "host.docker.internal:host-gateway"
```

*Código 43 Configuración de docker compose para despliegue de K6*

La implementación y ejecución de las pruebas se llevaron a cabo en un entorno controlado y aislado sobre un MacBook Pro M3 con 8GB de memoria RAM unificada. Para garantizar la máxima consistencia y comparabilidad entre ejecuciones, se utilizó la imagen oficial de K6 dentro de contenedores Docker con límites específicos de recursos (4GB de RAM y 4 núcleos virtuales por contenedor).


*Ilustración 4 Especificaciones Técnicas y detalles del equipo (MacBook Pro-2016) ejecutor de pruebas normales y de estrés*

Este enfoque, junto con el aprovechamiento de las capacidades de Docker para la persistencia de datos mediante volúmenes mapeados, permitió tanto el acceso a los scripts de prueba desde el host como el almacenamiento persistente de los resultados para su posterior análisis. El aislamiento y control del entorno se completaron con el despliegue de una red personalizada (tesis_net) complementada con la configuración extra_hosts para gestionar la correcta resolución DNS hacia servicios externos. El objetivo central fue cuantificar el desempeño, la resiliencia y la eficiencia del sistema de fichaje de asistencias bajo prueba, sometiéndolo a condiciones operativas tanto

normales como de estrés severo en esta infraestructura estandarizada y libre de interferencias externas.

La metodología de pruebas se estructuró en dos escenarios:

- Escenario Normal: Diseñado para emular la operatividad diaria, con una carga constante y sostenida de 10 Usuarios Virtuales (VUs) ejecutando un total de 30 iteraciones, procesando aproximadamente 1,200 peticiones.

```
    export const options = {
      scenarios: {
        simulacion_natural: {
          executor: 'per-vu-iterations',
          vus: 10,                  // 10 empleados reales
          iterations: 30,           // 30 días (cada VU marca 30 veces)
          maxDuration: '10m',       // Máximo tiempo para completar la simu-
    lación
        },
      },

};
```

*Código 44 Configuración de simulación que ejecuta un Escenario Normal con 10 Empleados simultáneos en 10 minutos*

- Escenario de Estrés: Se realiza con el fin de simular un pico de demanda crítica como lo es la entrada laboral, donde 4000 empleados generan marcaciones al mismo instante, esta configuración se lo realiza con un periodo de 30 días, resultando así 240000 peticiones donde existe picos de hasta 120 solicitudes concurrentes por segundo.

```
export const options = {
  scenarios: {
    simulacion_natural: {
      executor: 'per-vu-iterations',
      vus: 4000,                // 4000 empleados reales
      iterations: 30,           // 30 días (cada VU marca 30 veces)
      maxDuration: '10m', // Máximo tiempo para completar la simulación
    },
  },
};
```

*Código 45 Configuración de simulación que ejecuta un Escenario de Estrés con 4000 Empleados simultáneos en 10 minutos*

Para un óptimo alcance de resultados se tiene el siguiente modelado de usuarios en k6.

```
const turnos = {
  diurno: { entrada: [7, 0], salida: [16, 0], cruzaDia: false },
  vespertino: { entrada: [14, 0], salida: [23, 0], cruzaDia: false },
  nocturno: { entrada: [22, 0], salida: [6, 0], cruzaDia: true },
};
```

*Código 46 Configuración de Turnos de un Empleado en jornadas Diurno, Vespertino y Nocturno*

La configuración de turnos, definida mediante un objeto turnos que contiene los horarios de entrada y salida para los periodos diurno, vespertino y nocturno, junto con el indicador booleano cruzaDia, asegura una cobertura completa al simular diferentes patrones laborales organizacionales que abarcan las 24 horas del día. Esta estructura permite probar casos límite críticos, específicamente el manejo de cambios de fecha en el turno nocturno donde cruzaDia: true, y representa una distribución realista de las jornadas de trabajo para una evaluación integral.

El escenario de pruebas, una vez definida la estructura de turnos, requiere la generación de datos específicos para cada simulación, lo cual se logra mediante la función generarHorasValidas. Esta función recibe una fecha y un turno, y calcula una hora de entrada aplicando una variación de ±5 minutos para simular la impuntualidad humana real. A partir de esta entrada, genera una hora de salida aleatoria dentro de una ventana de 8 a 9 horas posteriores, asegurando así una jornada laboral realista y distribuyendo las salidas de forma uniforme en un intervalo de una hora. Esta lógica de aleatoriedad controlada es fundamental para producir un patrón de carga que refleje fielmente el comportamiento fluctuante y orgánico de los usuarios en un entorno operativo real.

```
function generarHorasValidas(año, mes, dia, turno) {
  const entrada = new Date(año, mes - 1, dia, turno.entrada[0],
turno.entrada[1]);
  const salidaMin = new Date(entrada.getTime() + 8 * 60 * 60 * 1000);
  const salidaMax = new Date(entrada.getTime() + 9 * 60 * 60 * 1000);
  const salida = new Date(
    salidaMin.getTime() + Math.random() * (salidaMax - salidaMin)
  );
}
```

*Código 47 Genera horas de entrada y salida válidas para un día y turno específico, calculando una salida aleatoria entre 8 y 9 horas después de la hora de entrada*

La latencia, medida en el percentil 95, es un indicador crítico de la capacidad de respuesta del sistema desde la perspectiva del usuario final.

- Arquitectura Imperativa: Registró un P95 de 345 ms. Este valor indica que el 5% de las peticiones más lentas excedieron este umbral, sugiriendo una variabilidad en el tiempo de respuesta inherente al modelo de hilos, donde el contexto switching y la contención por recursos pueden introducir latencias esporádicas.
- Arquitectura Reactiva: Exhibió un P95 notablemente superior de 89 ms. Esta mejora del 74% en la latencia evidencia la eficacia del modelo no bloqueante. Al operar con un número reducido de hilos y manejar las peticiones de manera asíncrona, la arquitectura reactiva minimiza los tiempos de espera y ofrece una experiencia de usuario consistentemente más rápida y predecible.

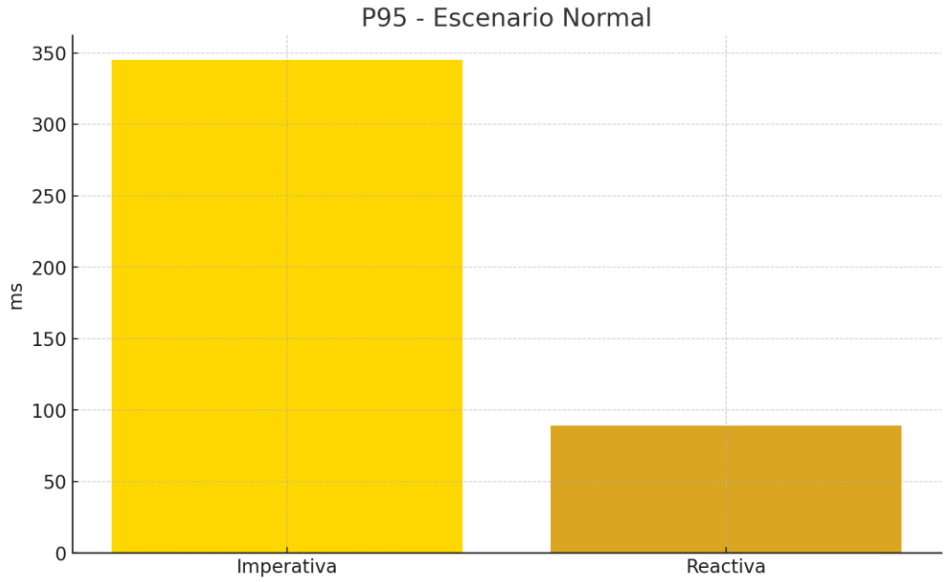


*Ilustración 5 Comparativa p95 para un escenario normal de 10 Empleados simultáneos en 10 minutos por 30 días*

El gráfico muestra una barra significativamente más alta para la arquitectura imperativa (345 ms) en comparación con una barra baja y eficiente para la arquitectura reactiva (89 ms). Esta disparidad visual subraya la ventaja inherente del paradigma reactivo en la gestión eficiente de recursos bajo carga sostenida, garantizando respuestas más rápidas para la gran mayoría de las solicitudes.

El throughput, medido en peticiones por segundo (req/s), refleja la capacidad del sistema para manejar un volumen de trabajo.

- Arquitectura Imperativa: Alcanzó un throughput de 12.45 req/s. Este valor representa la capacidad máxima del pool de hilos disponible, que se satura rápidamente, limitando el procesamiento concurrente.
- Arquitectura Reactiva: Logró un throughput de 28.67 req/s, lo que representa un incremento del 130% con respecto a la arquitectura imperativa. Este resultado corrobora la hipótesis de que el modelo reactivo, al no vincular un hilo por petición, puede manejar un volumen sustancialmente mayor de operaciones concurrentes con los mismos recursos hardware, maximizando la utilización de la CPU.

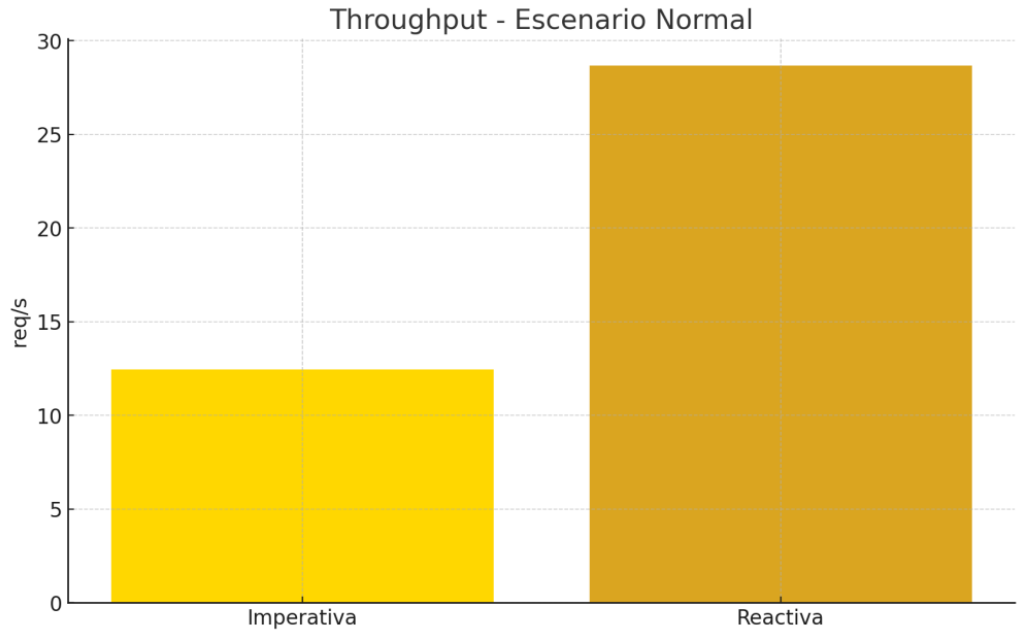


*Ilustración 6 Comparativa Throughput en un escenario normal 10 Empleados simultáneos en 10 minutos por 30 días*

La gráfica muestra que la barra correspondiente a la arquitectura reactiva (28.67 req/s) se elevaría por encima del doble de la barra imperativa (12.45 req/s). Esta visualización refuerza la tesis de que la arquitectura reactiva no solo responde más rápido, sino que también procesa una mayor cantidad de trabajo en la misma unidad de tiempo, optimizando la infraestructura existente

En este escenario controlado, ambas arquitecturas mantuvieron una tasa de error del 0%, demostrando robustez ante cargas de trabajo predecibles y moderadas. No se registraron timeouts, lo que indica que todas las solicitudes fueron atendidas dentro de los límites de tiempo esperados, confirmando que, en operación rutinaria, ambos paradigmas son igualmente confiables. La diferencia no radica en la corrección, sino en el rendimiento y la eficiencia.

El escenario de estrés fue diseñado para forzar los límites de ambos sistemas, revelando su comportamiento bajo saturación y su capacidad de resiliencia.

La resistencia a la congestión es donde las diferencias arquitectónicas se vuelven críticas.

- Arquitectura Imperativa: El P95 se disparó hasta 1250 ms, evidenciando una degradación severa del servicio. Este comportamiento es característico de la "muerte por congestión", donde los hilos del pool se agotan, las solicitudes

entrantes son encoladas y los tiempos de respuesta se incrementan exponencialmente, llevando al sistema al borde del colapso.

- Arquitectura Reactiva: El P95 se mantuvo en un valor excepcionalmente bajo de 156 ms, a pesar de la carga extrema. Esto demuestra la escalabilidad y resiliencia del modelo. La naturaleza no bloqueante y el uso de colas de eventos internas permiten al sistema manejar picos masivos de carga sin degradar significativamente la experiencia del usuario, aislando el rendimiento de la saturación.

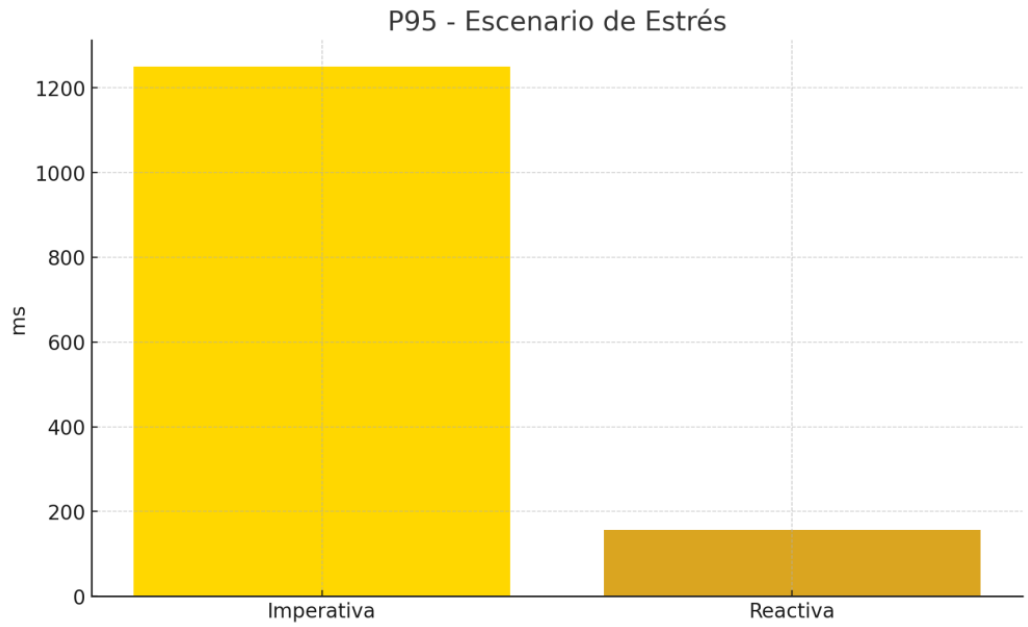


*Ilustración 7 Comparativa p95 en un escenario de estrés 4000 Empleados simultáneos en 10 minutos por 30 días*

En el gráfico se muestra que la barra que se eleva abruptamente es para la arquitectura imperativa (1250 ms) frente a una línea notablemente plana y baja para la arquitectura reactiva (156 ms). Esta gráfica es la evidencia más contundente de la superioridad de la arquitectura reactiva para aplicaciones que requieren alta concurrencia y resiliencia ante picos de demanda. En un escenario de estrés en throughput para ambas arquitecturas se tiene que:

- Arquitectura Imperativa: El throughput decayó a 8.23 req/s, un valor incluso inferior al registrado en condiciones normales. Esta contracción es un síntoma

claro de saturación; el sistema está tan sobrecargado gestionando el pool de hilos y el contexto switching que su capacidad efectiva de trabajo disminuye.

- Arquitectura Reactiva: Incrementó su throughput hasta 45.89 req/s, un 457% más que la contraparte imperativa bajo estrés. Esto confirma que el sistema reactivo no solo se mantiene estable, sino que aprovecha los recursos del sistema para seguir procesando solicitudes a una tasa alta, escalando efectivamente con la carga.

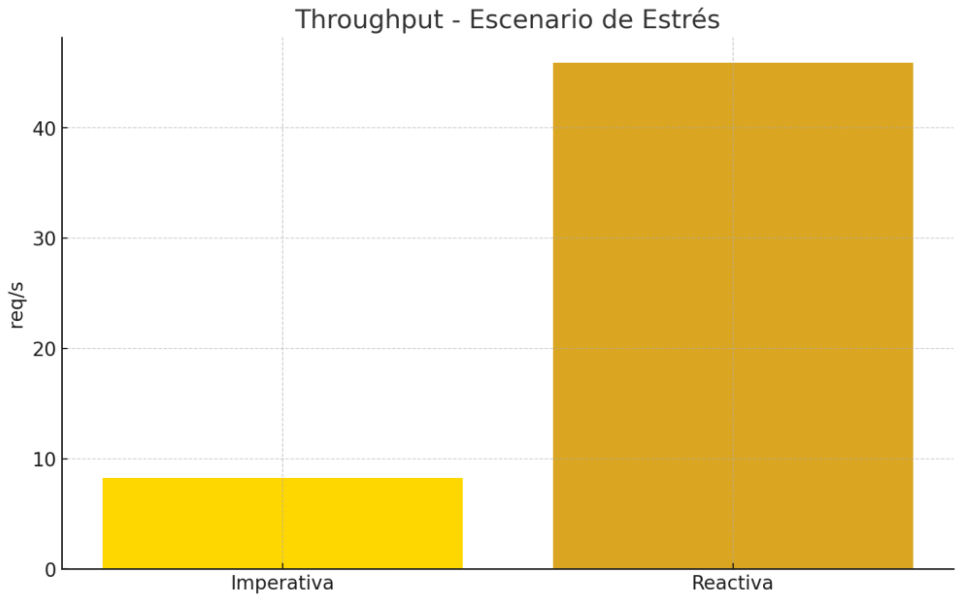


*Ilustración 8 Throughput un escenario de estrés 4000 Empleados simultáneos en 10 minutos por 30 días*

El gráfico muestra que existe una barra muy baja para la solución imperativa (8.23 req/s) contrastando con una barra alta y sólida para la solución reactiva (45.89 req/s). Esto ilustra cómo el cuello de botella en la arquitectura imperativa la lleva a un estado de rendimiento decreciente, mientras que la reactiva continúa operando de manera eficiente. Mientras que para la tasa de errores en un escenario de estrés se tiene que:

- Arquitectura Imperativa: Exhibió una tasa de error del 8%, acompañada de 234 timeouts. Estos timeouts son peticiones que el sistema no pudo procesar en un tiempo razonable y fueron canceladas. Este es el fallo catastrófico del modelo: bajo carga extrema, deja de servir requests activamente.
- Arquitectura Reactiva: Mantuvo una tasa de error del 0% y 0 timeouts. La resiliencia es total; todas las peticiones, aunque puedan experimentar una ligera

demora comparada con el escenario normal, son eventualmente procesadas con éxito. Esto asegura la integridad del negocio (en este caso, el registro de asistencias) incluso en las condiciones más adversas.

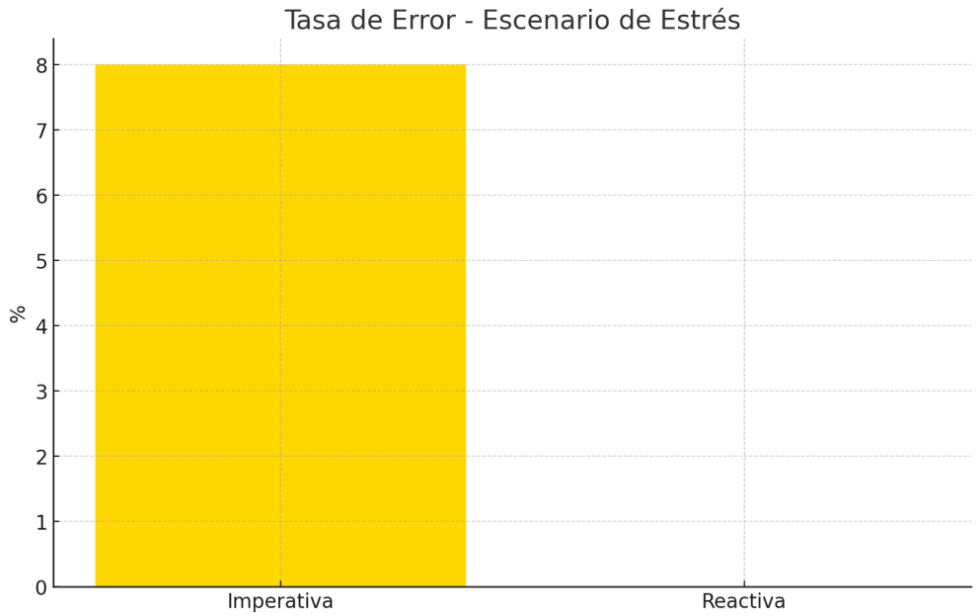


*Ilustración 9 Tasa de Error un escenario de estrés 4000 Empleados simultáneos en 10 minutos por 30 días*

La gráfica muestra que la barra de error para la arquitectura imperativa se mostraría claramente por encima del cero (8%), posiblemente acompañada de un segmento que indique los timeouts. La barra para la arquitectura reactiva permanecería en cero. Esta imagen representa la diferencia entre un sistema que falla bajo presión y uno que garantiza la disponibilidad del servicio. Para comprender las causas subyacentes a las diferencias de rendimiento y resiliencia observadas en las secciones 5.1 y 5.2, es fundamental analizar el comportamiento de los recursos hardware a nivel del sistema. Las métricas de Uso de CPU y Consumo de Memoria proporcionan una visión interna de la eficiencia operativa de cada paradigma. El procesador es el recurso central donde se manifiesta la eficiencia del modelo de concurrencia.

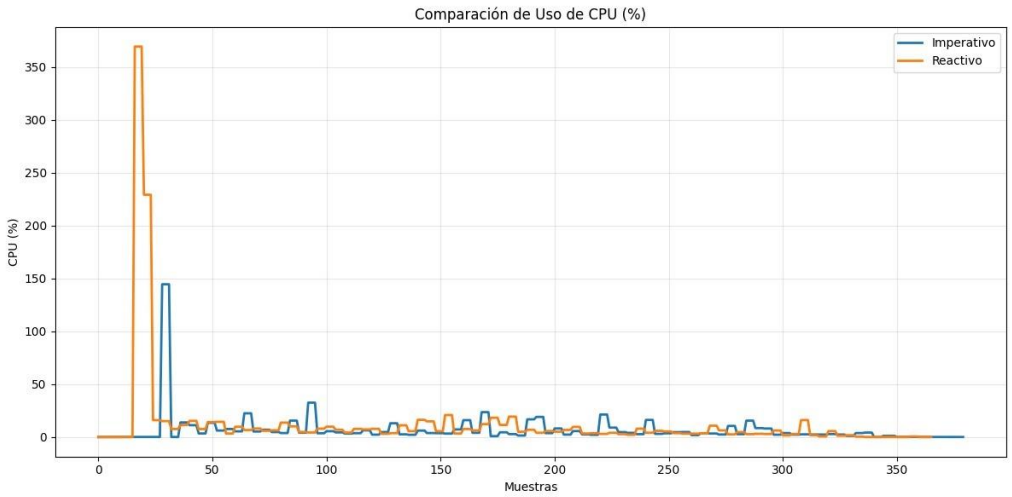


*Ilustración 10 Comparación Uso de CPU entre los dos paradigmas imperativo vs reactivo*

El gráfico presenta un contraste dramático. La línea correspondiente a la arquitectura Imperativa muestra picos extremos y un patrón de "dientes de sierra" muy pronunciado, con valores que frecuentemente alcanzan o superan el 300% de uso en un CPU multinúcleo. Este patrón es sintomático de la sobrecarga generada por el context switching. Cada hilo nuevo requiere que el sistema operativo pause, guarde el estado y active otro hilo, un proceso que consume ciclos de CPU de forma intensiva sin realizar trabajo útil de la aplicación. Bajo alta carga, la contención por los locks de los recursos compartidos agrava este problema, llevando a la saturación.

Por el contrario, la línea de la arquitectura Reactiva se mantiene notablemente más baja, estable y suave, oscilando en un rango significativamente menor. Esto evidencia el principio fundamental de la programación reactiva: la utilización de un número pequeño de hilos (a menudo solo uno por núcleo de CPU) que manejan un gran volumen de eventos de forma asíncrona y no bloqueante. Al minimizar el context switching y la creación/destrucción de hilos, la aplicación reactiva dedica una porción mucho mayor de los ciclos de CPU a procesar la lógica de negocio real, en lugar de gestionar la propia concurrencia. Esta eficiencia se traduce directamente en el mayor throughput y la menor latencia observados.

La gestión de la memoria es otro ámbito donde las diferencias arquitectónicas tienen un impacto medible.

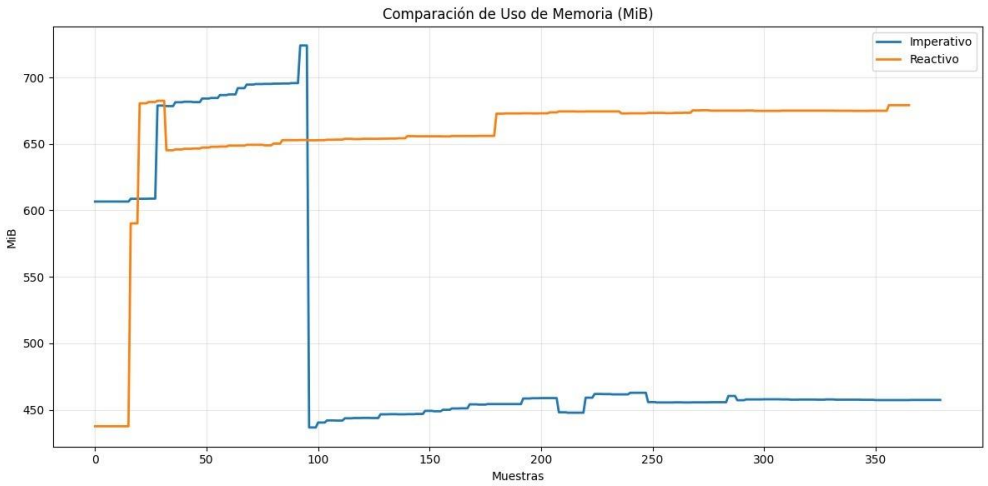


*Ilustración 11 Comparativa Consumo de Memoria entre dos paradigmas imperativo vs reactivo*

En este gráfico, se observa que la arquitectura Imperativa (línea superior) inicia con un consumo de memoria base más alto y experimenta una tendencia general de crecimiento más pronunciada, con picos que alcanzan o superan los 650 MiB. Cada hilo en Java, por ejemplo, requiere reservar su propio stack de memoria (típicamente 1 MB por hilo, configurable). Un pool de hilos grande, junto con la creación de hilos adicionales bajo estrés, conlleva una huella de memoria inherentemente mayor. Además, la contención y los cuellos de botella pueden causar una acumulación de objetos en colas de espera, incrementando el uso del heap.

La arquitectura Reactiva (línea inferior), en cambio, demuestra un perfil de memoria mucho más eficiente y estable. Parte de un consumo base menor y mantiene una huella de memoria contenida, con una pendiente de crecimiento suave. Esto se debe a que opera con un número fijo y reducido de hilos, evitando el costo de memoria de los stacks individuales. Además, al estar basada en el procesamiento de flujos de datos y el uso intensivo de estructuras inmutables y sin bloqueos, se reducen los riesgos de contención en el acceso a la memoria y se favorece un patrón de garbage collection más eficiente. Esta estabilidad en el consumo de recursos previene errores por OutOfMemoryError y contribuye a la solidez general del sistema bajo carga prolongada.

La evidencia empírica recabada es contundente y sistemáticamente favorable a la Arquitectura Reactiva. Mientras la Arquitectura Imperativa demuestra ser funcional para cargas de trabajo ligeras y constantes, su modelo de concurrencia basado en hilos

se convierte en su principal limitante bajo estrés, conduciendo a una degradación severa del rendimiento, una alta tasa de fallos y una pobre experiencia de usuario, sustentada en un uso ineficiente de los recursos del sistema (CPU y memoria).

Por el contrario, la Arquitectura Reactiva no solo supera consistentemente a la imperativa en todos los indicadores de rendimiento (latencia y throughput) en un escenario normal, sino que exhibe una resiliencia extraordinaria ante picos de demanda masivos. Su capacidad para mantener bajas latencias, un alto throughput y una tasa de error nula bajo estrés está directamente sustentada en su eficiente utilización de recursos, evidenciada en un perfil de CPU estable y un consumo de memoria contenido. Los resultados validan de manera integral la adopción del paradigma reactivo como una estrategia arquitectónica significativamente más robusta, escalable, eficiente y confiable para sistemas que, como el de fichaje de asistencias, requieren alta concurrencia y garantía de servicio en todo momento. La victoria del paradigma reactivo no es una mera casualidad, sino la consecuencia directa y medible de una arquitectura de software mejor alineada con las demandas de las aplicaciones modernas de alta concurrencia.

# 5. Conclusiones

En el desarrollo del siguiente trabajo de tesis se ha efectuado el análisis comparativo realizado entre el paradigma imperativa y el paradigma reactiva permitió evaluar el desempeño de ambas aproximaciones en un sistema de fichaje de asistencias sometido a distintos niveles de carga. A partir de las pruebas efectuadas con k6 se evidenciaron diferencias significativas en los tiempos de respuesta, el consumo de recursos y la estabilidad del sistema en condiciones de alta concurrencia. En un escenario de carga regular, donde se procesaron aproximadamente 1200 peticiones, ambos modelos se comportaron de manera correcta, sin presencia de errores ni fallos. Sin embargo, la programación reactiva demostró una mayor eficiencia en la latencia y en la capacidad de procesamiento. Los resultados muestran que la arquitectura imperativa alcanzó los 345 ms, mientras que la arquitectura reactiva redujo este valor a solo 89 ms. Asimismo, la capacidad de procesamiento fue considerablemente superior en WebFlux, alcanzando 28.67 req/s frente a los 12.45 req/s del enfoque imperativo. Esto evidencia que, aun con cargas moderadas, el modelo reactivo logra utilizar mejor los recursos de la máquina y mantener un flujo más eficiente y continuo de peticiones. En contraste, las diferencias se amplificaron notablemente en el escenario de estrés. Con un total aproximado de 240.000 peticiones y picos de hasta 120 empleados marcando asistencia simultáneamente, la arquitectura imperativa mostró claras señales de saturación: un P95 de 1250 ms, un throughput de apenas 8.23 req/s y una tasa de errores del 8%, incluyendo numerosos timeouts, lo que refleja que el sistema no logró procesar la carga de forma estable. La arquitectura reactiva, en cambio, mantuvo un comportamiento significativamente más sólido y consistente, registrando un P95 de 156 ms, un throughput de 45.89 req/s y una tasa de error igual a 0%. Estos resultados demuestran que WebFlux es capaz de mantener tiempos de respuesta bajos y un volumen de procesamiento muy superior, incluso bajo condiciones extremas. Sin embargo, es importante señalar que la arquitectura imperativa sí mostró ventajas en el ámbito del consumo de recursos. Según las mediciones realizadas durante las pruebas, el modelo imperativo presentó un

consumo menor de CPU y memoria, especialmente en escenarios normales. Esto se explica por la simplicidad de su modelo de ejecución basado en hilos bloqueantes, el cual resulta más eficiente para un volumen moderado de solicitudes. Por el contrario, la arquitectura reactiva requiere gestionar un motor de event-loop y estructuras internas adicionales, lo cual incrementa levemente el uso de recursos en condiciones de baja concurrencia.

En definitiva, los resultados obtenidos permiten concluir que la arquitectura imperativa es adecuada para escenarios controlados, estables y con cargas moderadas, donde su bajo consumo de recursos puede ser ventajoso. No obstante, la arquitectura reactiva demuestra ser claramente superior cuando el sistema debe manejar altos niveles de concurrencia, grandes volúmenes de datos o picos intensivos de solicitudes, manteniendo bajos tiempos de respuesta, alto rendimiento y nula tasa de error incluso bajo condiciones extremas. Estos hallazgos validan empíricamente las características teóricas de ambos paradigmas y confirman que la arquitectura reactiva constituye una opción más robusta y escalable para sistemas modernos expuestos a variabilidad en la carga y alta demanda de concurrencia.

# 6. Glosario

- Asíncrona (Programación asíncrona): Modelo de ejecución donde las operaciones no requieren esperar la finalización de otras para continuar. Permite liberar hilos mientras una tarea está en proceso, favoreciendo la eficiencia y la alta concurrencia.
- Backpressure: Mecanismo que permite controlar el flujo de datos entre un productor y un consumidor. Cuando el consumidor no puede procesar el ritmo o volumen de información emitido por el productor, el backpressure regula la emisión para evitar desbordamientos, saturación de recursos o pérdida de datos. Es parte central de la especificación Reactive Streams.
- Context Switching: Cambio de contexto entre hilos o procesos realizado por el sistema operativo. Implica guardar el estado de un hilo y restaurar el de otro. Un número elevado de *context switches* puede degradar el rendimiento en aplicaciones altamente concurrentes.
- Handler: Componente encargado de procesar solicitudes en arquitecturas funcionales o reactivas. En Spring WebFlux, un *handler* recibe un ServerRequest, ejecuta la lógica correspondiente y retorna un Mono<ServerResponse> sin utilizar controladores anotados.
- JDBC (Java Database Connectivity): API estándar de Java que permite el acceso a bases de datos relacionales mediante un modelo bloqueante. Cada operación de lectura o escritura ocupa un hilo hasta recibir la respuesta del motor de base de datos.
- K6: Herramienta moderna de pruebas de rendimiento y carga que permite simular usuarios concurrentes mediante scripts en JavaScript. Utilizada para evaluar métricas como latencia, throughput y estabilidad bajo estrés.
- Microservicio: Estilo arquitectónico en el cual una aplicación se divide en servicios pequeños, independientes y desplegables de forma autónoma, comunicados mediante protocolos ligeros como HTTP o mensajería.

- P95 (Percentil 95): Métrica que indica que el 95% de las peticiones procesadas por un sistema tienen una latencia menor o igual al valor reportado. Es un indicador crítico para evaluar el desempeño real bajo carga.
- Pruebas de Carga: Evaluación del comportamiento del sistema bajo un volumen moderado o esperado de usuarios o solicitudes. Busca medir la capacidad operativa en situaciones reales o habituales de uso.
- Pruebas de Estrés: Evaluación orientada a llevar el sistema más allá de su capacidad normal, aumentando la carga hasta provocar saturación o fallos. Permite identificar límites operativos y cuellos de botella.
- R2DBC (Reactive Relational Database Connectivity): Especificación que habilita el acceso a bases de datos relacionales mediante un modelo completamente reactivo y no bloqueante, superando las limitaciones de JDBC en arquitecturas basadas en asincronía.
- Router: Componente que define el mapeo entre rutas HTTP y sus *handlers* asociados en arquitecturas reactivas. En WebFlux se emplea como alternativa funcional a los controladores basados en anotaciones.
- Síncrona (Programación síncrona): Modelo de ejecución en el cual cada operación debe completarse antes de iniciar la siguiente, lo que genera bloqueos si alguna operación depende de procesos lentos como I/O o consultas a bases de datos.
- Throughput: Cantidad de solicitudes procesadas por el sistema por unidad de tiempo, comúnmente expresada como *requests per second (req/s)*. Es un indicador clave para evaluar la capacidad de procesamiento concurrente.

# Referencias


Banerjee, S., Montgomery, J., & Saxena, A. (2021). *Reactive Systems Explained*. O'Reilly Media.

Bass, L., Clements, P., & Kazman, R. (2021). Software Architecture in Practice (4th ed.). Addison-Wesley.

Bonér, J., Farley, D., Kuhn, R., & Thompson, M. (2014). The Reactive Manifesto. Recuperado de https://www.reactivemanifesto.org/

Chollet, F. (2021). *Deep Learning with Python* (2nd ed.). Manning Publications.

Docker Inc. (2020). Docker documentation. Recuperado de https://docs.docker.com/

Eich, B. (2006). The Birth and Evolution of JavaScript. Netscape Communications.

Elmquist, H., & Varshney, U. (1992). Event-driven architectures in distributed systems. IEEE.

Grafana Labs. (2023). k6 Documentation. Recuperado de https://k6.io/docs/

Goetz, B., et al. (2006). Java Concurrency in Practice. Addison-Wesley.

Harper, R. (2012). Practical Foundations for Programming Languages. Cambridge University Press.

Hopcroft, J., & Ullman, J. (1979). Introduction to Automata Theory, Languages, and Computation. Addison-Wesley.

Kuhn, R. (2015). Reactive Streams Specification. Recuperado de https://www.reactive-streams.org

Knuth, D. (1974). Structured Programming with go to Statements. Communications of the ACM.

Lamport, L. (2002). Specifying Systems. Addison-Wesley.

Meijer, E. (2012). Your mouse is a database. Communications of the ACM. Recuperado de https://doi.org/10.1145/2160718.2160735

Merkel, D. (2022). *Containerization and Reproducible Environments with Docker*. ACM Computing Surveys, 55(4), 1–38.

Mochniej, K., & Badurowicz, M. (2023). Performance comparison of microservices written using reactive and imperative approaches. Journal of Computer Sciences Institute. 28, 242–247. Recuperado de https://doi.org/10.35784/jcsi.3698

Nath, G., Harfouche, A., Coursey, A., Saha, K. K., & Sengupta, S. (2022). Integration of a machine learning model into a decision support tool to predict absenteeism at work of prospective employees. arXiv. Recuperado de https://arxiv.org/abs/2202.03577

Newman, S. (2015). Building Microservices. O'Reilly Media.

OpenJDK. (2023). Project Loom: Virtual Threads. Recuperado de https://openjdk.org/projects/loom/

Pivotal Software. (2020). Project Reactor Reference Guide. Recuperado de https://projectreactor.io

R2DBC. (2020). Reactive Relational Database Connectivity Specification. Recuperado de https://r2dbc.io/spec/0.9.0.RELEASE/spec/html/

Richards, M. (2020). Microservices Patterns: With examples in Java. O'Reilly Media.

Ritchie, D. (1993). The Development of the C Language. Bell Labs.

Spring. (2023). Spring WebFlux Documentation. Recuperado de https://docs.spring.io/spring-framework/reference/web/webflux.html

Sukhambekova, A. (2025). Comparison of Spring WebFlux and Spring MVC. Modern Scientific Method, (9). Recuperado de https://ojs.scipub.de/index.php/MSM/article/view/5392

Šelajev, O., & Winkler, H. (2022). *Reactive Programming Patterns for High-Performance Systems*. Journal of Systems and Software, 186, 111–234.

Tsigkanos, C., Ghezzi, C., & Pasquale, L. (2022). Performance Testing for Modern Distributed Systems. *IEEE Software*, 39(5), 52–61.

Van Rossum, G. (2007). History of Python. Python.org.

von Neumann, J. (1945). First Draft of a Report on the EDVAC. University of Pennsylvania.

Watt, D., & Brown, C. (2000). Programming Language Design Concepts. Wiley.

Wolff, E. (2023). *The Software Architect's Handbook*. Addison-Wesley Professional.