UNIVERSIDAD DE CONCEPCIÓN
FACULTAD DE INGENIERIA

MICRO-OPTIMIZACIÓN DE MICROSERVICIOS CONTENERIZADOS: COMPILACIÓN JUST-IN-TIME DURANTE LA CONTENERIZACIÓN

Por: Roberto Felipe Artigues Escobar

Memoria de Título presentada a la Facultad de Ingeniería de la Universidad de Concepción para optar al título profesional de Ingeniero Civil Informático

Profesor Patrocinante: Geoffrey Jean-Pierre Christophe Hecht
Profesor Co-Supervisor: Zheng Li

Septiembre 2025
Concepción, Chile

AGRADECIMIENTOS

Agradezco profundamente a mi familia por su apoyo incondicional. Su constante motivación y sus consejos que me impulsaron a dar lo mejor de mí en cada etapa de este proceso.

A mi pareja, Ana, quien ha sido mi compañera incondicional en este camino. Su apoyo durante las largas noches de estudio o trabajo, sus palabras de aliento, y su capacidad para recordarme mis fortalezas cuando yo no las veía.

Quiero también agradecer a todos los profesores que compartieron su conocimiento y experiencia durante mi formación. Especialmente al profesor Geoffrey y Zheng, por brindarme la oportunidad de trabajar en este proyecto, por su paciencia, orientación constante y por confiar en mi potencial.

Por último, quiero agradecer a todos los amigos que he hecho desde que empecé este viaje, que han sido un apoyo constante, que me han motivado a seguir adelante y por toda la ayuda que me han entregado desde el primer día.

Gracias Sebastian, Felipe, José, Emilio, Sebastian S, Lizandro, Nicolas, Erick, Alejandro, Pedro, Ana y Juan. Su compañía, apoyo mutuo y los momentos juntos hicieron que esta experiencia fuera posible. No lo habría logrado sin su presencia continua en mi vida.

Declaración de Uso Ético de Herramientas de IA Generativa

Yo, Roberto Felipe Artigues Escobar, declaro que he utilizado Claude (Anthropic) de manera ética y responsable para apoyar este trabajo de tesis en las siguientes actividades:

- **Redacción y Estructuración:** Mejora de coherencia, claridad, estilo y corrección ortográfica del documento.
- **Revisión de Código:** Optimización de scripts, Dockerfiles y código de benchmarks experimentales.
- **Asesoría Técnica:** Consultas sobre contenerización, optimización y análisis estadístico, siempre validadas con fuentes académicas.
- **Análisis de Resultados:** Asistencia en interpretación de datos experimentales, con conclusiones validadas mediante análisis estadístico riguroso.

Declaro que todo contenido asistido por IA ha sido revisado y validado para asegurar su originalidad y pertinencia académica. Soy el único responsable del trabajo presentado, de las conclusiones derivadas de la investigación experimental, y me comprometo a que todas las fuentes utilizadas estén debidamente citadas según las normas académicas establecidas.

La implementación experimental, la recolección de datos, el análisis estadístico y las conclusiones de esta investigación son producto de mi trabajo original, utilizando las herramientas de IA únicamente como apoyo en las tareas específicas mencionadas anteriormente.

Resumen

Los microservicios contenerizados implementados con lenguajes interpretados como Python y Node.js presentan penalizaciones de rendimiento debido al overhead de interpretación en tiempo de ejecución. Esta investigación evaluó la hipótesis de que la aplicación de técnicas de compilación anticipada durante el proceso de contenerización mejora el rendimiento en tiempo de ejecución de microservicios implementados con lenguajes interpretados. Se aplicaron tres principios de micro-optimización: just-enough containerisation (incluir solo componentes necesarios), just-for-me configuration (configuraciones específicas) y just-in-time compilation (compilar durante la construcción del contenedor). La metodología experimental comparó implementaciones estándar versus optimizadas usando micro-benchmarks sintéticos y la aplicación Bookinfo con servicios en Python (Flask), Node.js (Express) y Java (Spring Boot). Los experimentos se ejecutaron en plataformas ARM64 (MacBook M4 Pro y Raspberry Pi 4). Los resultados mostraron efectividad variable: Node.js logró mejoras del 27.9 % en throughput usando pkg, Python mostró degradación del 36.6 % debido a incompatibilidades entre Flask y Nuitka, mientras Java presentó mejoras del 8.2 %. La investigación identificó que la complejidad del framework tiene mayor influencia que el lenguaje, revelando trade-offs entre rendimiento, tiempo de inicio y recursos. Se concluye que la compilación durante contenerización es efectiva para microservicios Node.js con frameworks minimalistas, pero presenta resultados negativos para frameworks Python con características dinámicas, proporcionando criterios para decisiones arquitectónicas basadas en el contexto específico.

Palabras clave: microservicios, contenerización, Docker, compilación just-in-time, optimización de rendimiento, benchmarks

Summary

Containerized microservices implemented with interpreted languages such as Python and Node.js present performance penalties due to runtime interpretation overhead. This research evaluated the hypothesis that the application of ahead-of-time compilation techniques during the containerization process improves the runtime performance of microservices implemented with interpreted languages. Three micro-optimization principles were applied: just-enough containerisation (including only necessary components), just-for-me configuration (specific configurations), and just-in-time compilation (compiling during container build). The experimental methodology compared standard versus optimized implementations using synthetic micro-benchmarks and the Bookinfo application with services in Python (Flask), Node.js (Express), and Java (Spring Boot). Experiments were conducted on ARM64 platforms (MacBook M4 Pro and Raspberry Pi 4). Results showed variable effectiveness: Node.js achieved 27.9 % throughput improvements using pkg, Python showed 36.6 % degradation due to incompatibilities between Flask and Nuitka, while Java presented 8.2 % improvements. The research identified that framework complexity has greater influence than language, revealing trade-offs between performance, startup time, and resources. It is concluded that compilation during containerization is effective for Node.js microservices with minimalist frameworks but shows negative results for Python frameworks with dynamic features, providing criteria for architectural decisions based on specific context.

Keywords: microservices, containerization, Docker, just-in-time compilation, performance optimization, benchmarks

Índice general

Índice de cuadros

Índice de figuras

Capitulo 1

Introducción

1.1. Contexto y Motivación

En los últimos años, la arquitectura de microservicios se ha convertido en el paradigma dominante para el desarrollo de aplicaciones distribuidas modernas. Esta arquitectura permite descomponer sistemas monolíticos complejos en servicios pequeños e independientes que pueden desarrollarse, desplegarse y escalarse de manera autónoma. Docker, como tecnología de contenerización líder, ha facilitado considerablemente el despliegue de estos microservicios mediante contenedores ligeros y portables.

Sin embargo, existe un problema importante cuando los microservicios se implementan usando lenguajes interpretados como Python y Node.js, o lenguajes que requieren máquinas virtuales como Java. Estos servicios enfrentan penalizaciones de rendimiento sustanciales que impactan directamente en la calidad del servicio. La sobrecarga de interpretación representa el desafío más fundamental, ya que el código debe ser traducido línea por línea durante la ejecución, agregando latencia a cada operación. Además, los tiempos de inicio prolongados constituyen una limitación importante en entornos de alta elasticidad donde los contenedores se crean y destruyen frecuentemente. Finalmente, el uso ineficiente de memoria se manifiesta porque los intérpretes y máquinas virtuales requieren memoria adicional para funcionar, reduciendo los recursos disponibles para la lógica de negocio.

Esta sobrecarga se vuelve especialmente relevante en arquitecturas de microservicios donde múltiples servicios deben comunicarse entre sí. Cada llamada entre servicios agrega latencia adicional que se acumula rápidamente, impactando negativamente la

experiencia del usuario final. Como señala Li (Li, 2024) en su investigación sobre micro-optimización, estas ineficiencias aparentemente pequeñas pueden resultar en degradaciones notables del rendimiento cuando se consideran a escala.

1.2. Definición del Problema

El problema central que aborda esta investigación es la sobrecarga de interpretación presente en microservicios contenerizados. Esta problemática se manifiesta de manera diferente según el lenguaje de programación utilizado.

En el caso de Python, las penalizaciones son particularmente severas, con velocidades de ejecución entre 10 y 100 veces más lentas que lenguajes compilados (Stoico et al., 2024). Esta degradación se ve agravada por el GIL (Global Interpreter Lock, Bloqueo Global del Intérprete), que limita el paralelismo real y fuerza la ejecución secuencial incluso en sistemas multi-core, desperdiciando recursos computacionales disponibles.

Node.js presenta desafíos diferentes pero igualmente significativos. A pesar de contar con el motor V8 que incluye compilación JIT (Just-In-Time, compilación justo a tiempo), el sistema requiere traducción continua de JavaScript durante la ejecución, lo que consume ciclos de CPU valiosos que podrían dedicarse al procesamiento de la lógica de negocio.

Por su parte, Java enfrenta el problema del período de calentamiento necesario para que la JVM (Java Virtual Machine) alcance su rendimiento óptimo. Durante este período inicial, el código se ejecuta de manera interpretada antes de que las optimizaciones JIT entren en acción, resultando en latencias inconsistentes que complican la predictibilidad del rendimiento.

Estos problemas se magnifican en arquitecturas *serverless* donde la creación y destrucción frecuente de contenedores hace que los tiempos de inicio sean críticos. Además, la gestión dinámica de tipos inherente a estos lenguajes incrementa el uso de memoria al requerir mantener metadata adicional durante toda la ejecución.

Existe una brecha notable en la investigación actual: mientras abundan estudios sobre optimización a nivel de código (lado del desarrollo) o gestión de recursos (lado de operaciones), pocos exploran las oportunidades de optimización durante el proceso mismo de contenerización.

1.3. Hipótesis de Investigación

La hipótesis principal de esta investigación plantea que:

"La aplicación de técnicas de compilación anticipada durante el proceso de contenerización mejora el rendimiento en tiempo de ejecución de microservicios implementados con lenguajes interpretados."

Esta hipótesis se fundamenta en una observación clave: cuando se construye una imagen de contenedor, el entorno de producción objetivo queda claramente definido. Se conoce la arquitectura del procesador (x86_64, ARM64), el sistema operativo (típicamente una distribución específica de Linux), y las versiones exactas de las bibliotecas que estarán disponibles. Esta información permite realizar optimizaciones específicas de plataforma que no serían posibles en un enfoque tradicional de .[es]cribir una vez, ejecutar en cualquier lugar".

La compilación anticipada durante la contenerización podría eliminar la necesidad de interpretación en tiempo de ejecución, generando múltiples beneficios complementarios. En primer lugar, se espera una reducción significativa de la latencia en cada solicitud procesada, ya que el código se ejecutaría directamente como instrucciones de máquina. Este beneficio se complementa con un menor consumo de CPU al eliminar el *overhead* (sobrecarga de procesamiento) del intérprete, liberando recursos computacionales para procesar más solicitudes concurrentes. Adicionalmente, los tiempos de inicio se verían significativamente reducidos al no requerir la inicialización del intérprete, lo que resulta especialmente valioso en entornos de alta elasticidad. Finalmente, el uso de memoria se optimizaría considerablemente al no necesitar mantener el intérprete cargado durante toda la ejecución del servicio.

1.4. Objetivos

1.4.1. Objetivo General

Evaluar si la aplicación de técnicas de compilación anticipada durante el proceso de contenerización mejora significativamente el rendimiento en tiempo de ejecución de microservicios implementados con lenguajes interpretados.

1.4.2. Objetivos Específicos

1. **Seleccionar e implementar aplicaciones de referencia:** Identificar o crear aplicaciones benchmark representativas usando lenguajes interpretados (Python, Node.js) y con máquina virtual (Java) que permitan evaluar diferentes aspectos del rendimiento.
2. **Establecer un entorno experimental controlado:** Configurar un ambiente de pruebas que permita mediciones precisas y reproducibles de métricas de rendimiento, incluyendo tiempo de respuesta, rendimiento (solicitudes por segundo), uso de recursos y tiempos de inicio.
3. **Implementar procesos de compilación anticipada:** Desarrollar scripts y configuraciones de Docker que incorporen herramientas de compilación como:
   - Nuitka[1] y Cython para Python
   - GraalVM[2] Native Image para Java
   - pkg[3] para Node.js
4. **Realizar análisis comparativo cuantitativo:** Ejecutar pruebas exhaustivas comparando el rendimiento entre contenedores estándar y contenedores con componentes pre-compilados.
5. **Evaluar la aplicación combinada de principios de micro-optimización:** Estudiar el efecto sinérgico de aplicar simultáneamente contenerización mínima suficiente, configuración personalizada y compilación justo a tiempo (*just-enough containerisation, just-for-me configuration y just-in-time compilation*).
6. **Documentar mejores prácticas:** Desarrollar guías prácticas y recomendaciones para que otros desarrolladores puedan implementar estas técnicas en proyectos del mundo real.

[1]Código fuente disponible en: https://github.com/Nuitka/Nuitka
[2]Código fuente disponible en: https://github.com/oracle/graal
[3]Código fuente disponible en: https://github.com/vercel/pkg

1.5. Alcance y Limitaciones

1.5.1. Alcance del Estudio

Esta investigación se enfoca en evaluar técnicas de micro-optimización para microservicios contenerizados, específicamente:

Lenguajes evaluados: La investigación se centra en tres lenguajes que representan diferentes paradigmas de ejecución prevalentes en microservicios modernos. Python (versión 3.11) fue seleccionado como representante de lenguajes interpretados puros, donde cada línea de código debe traducirse durante la ejecución. Node.js (versión 18.17) representa lenguajes con compilación JIT, ofreciendo un punto intermedio entre interpretación pura y compilación anticipada. Java (OpenJDK 11) completa el espectro como ejemplo de lenguajes que ejecutan sobre máquinas virtuales, con sus características únicas de optimización adaptativa.

Tipos de benchmarks:

- Micro-*benchmarks* sintéticos: Pruebas específicas para evaluar aspectos aislados del rendimiento (intensivo en CPU, intensivo en E/S, servicios web)
- Macro-*benchmarks*: Aplicación Bookinfo[4] de Istio (Istio Project, 2024), que simula una tienda de libros en línea con múltiples microservicios interconectados

Métricas evaluadas: El estudio emplea un conjunto comprehensivo de métricas para capturar todos los aspectos relevantes del rendimiento. Las métricas de rendimiento directo incluyen el tiempo de ejecución y la latencia de respuesta, fundamentales para evaluar la experiencia del usuario final. El rendimiento, medido en solicitudes por segundo, proporciona una visión de la capacidad máxima del sistema. El análisis de recursos examina tanto el uso de CPU como de memoria, crítico para el dimensionamiento de infraestructura. Adicionalmente, se evalúa el tamaño de las imágenes de contenedor, que impacta directamente en los tiempos de despliegue y costos de almacenamiento. Finalmente, el tiempo de inicio de los servicios se mide cuidadosamente, ya que resulta crucial en arquitecturas elásticas y *serverless*.

Plataformas objetivo: La investigación se centra en la arquitectura ARM64, evaluada en dos plataformas. El MacBook Pro M4 representa equipos de desarrollo con procesadores ARM de alto rendimiento, mientras que Raspberry Pi 4 permite evaluar

[4]Código fuente disponible en: https://github.com/nocalhost/bookinfo

el comportamiento en hardware con recursos limitados. Se eligió ARM64 debido a su creciente uso tanto en dispositivos móviles como en servidores *cloud*.

1.5.2. Limitaciones

Es importante reconocer las siguientes limitaciones del estudio:

Limitaciones técnicas: El estudio enfrenta restricciones inherentes al entorno experimental disponible. La dependencia de hardware específico para las pruebas, limitado a MacBook Pro M4 y Raspberry Pi 4, podría influir en la generalizabilidad de los resultados a otras plataformas. Además, las versiones fijas de herramientas de compilación utilizadas representan un punto temporal específico en tecnologías que evolucionan rápidamente. Es importante destacar que el enfoque se mantiene exclusivamente en optimizaciones del lado de operaciones, sin modificar el código fuente original, lo que preserva la integridad del desarrollo pero podría limitar las optimizaciones posibles.

Limitaciones de representatividad: Aunque los benchmarks seleccionados buscan capturar escenarios realistas, es necesario reconocer que las aplicaciones de prueba tienen complejidad limitada comparada con sistemas de producción reales que pueden involucrar cientos de microservicios interconectados. Los patrones de carga sintéticos utilizados, aunque cuidadosamente diseñados, pueden no capturar completamente la variabilidad y complejidad del comportamiento real de usuarios en producción. Adicionalmente, la selección de tres lenguajes, aunque representativa, no abarca todo el espectro de tecnologías utilizadas en microservicios modernos, excluyendo opciones como Go, Rust o C#.

Limitaciones de alcance: El estudio mantiene un enfoque específico en métricas de rendimiento, lo que implica que no se consideran aspectos de seguridad que podrían verse afectados por las optimizaciones aplicadas, como potenciales vulnerabilidades introducidas por la compilación anticipada. Similarmente, no se evalúa el impacto en la facilidad de depuración y mantenimiento, factores críticos en el ciclo de vida del software pero fuera del alcance de esta investigación. Finalmente, el estudio se limita a Docker como plataforma de contenerización, sin explorar alternativas como Podman o containerd directamente, reconociendo que Docker representa el estándar de facto en la industria.

Capitulo 2

Marco Teórico y Estado del Arte

2.1. Arquitecturas de Microservicios

2.1.1. Fundamentos y Evolución

La arquitectura de microservicios representa un cambio significativo en el desarrollo de software empresarial. A diferencia de las arquitecturas monolíticas tradicionales donde todos los componentes están fuertemente acoplados en una única base de código, los microservicios descomponen las aplicaciones en servicios pequeños, independientes y autónomos que se comunican a través de APIs bien definidas (típicamente REST sobre HTTP).

Esta evolución surge como respuesta a las limitaciones de los monolitos en entornos modernos que demandan agilidad y escalabilidad. Mientras que un monolito debe desplegarse completamente incluso para cambios mínimos, los microservicios permiten actualizar servicios individuales sin afectar al resto del sistema.

2.1.2. Características Fundamentales

Los microservicios se distinguen por varias características fundamentales que definen su arquitectura y filosofía de diseño. La descomposición por capacidades de negocio constituye el principio organizacional central, donde cada microservicio encapsula una funcionalidad de negocio específica y bien delimitada. Por ejemplo, en una aplicación de e-commerce, servicios separados manejarían el catálogo de productos, el carrito de compras, y el procesamiento de pagos, cada uno con su propio ciclo de vida y equipo

responsable.

Esta modularidad se complementa con la descentralización de datos, siguiendo el principio de "database per service" que otorga a cada microservicio control completo sobre su propio almacén de datos. Si bien esto elimina el acoplamiento a nivel de base de datos y permite optimizaciones específicas por servicio, también introduce el desafío de mantener la consistencia eventual entre servicios, requiriendo patrones como saga o event sourcing para transacciones distribuidas.

El diseño para fallas representa otro pilar fundamental, reconociendo que en sistemas distribuidos las fallas son inevitables más que excepcionales. Los microservicios implementan patrones defensivos como *circuit breakers* para prevenir cascadas de fallos, reintentos con *backoff* exponencial para manejar fallos transitorios, y *timeouts* agresivos para evitar bloqueos indefinidos. Esta resiliencia incorporada permite que el sistema continúe funcionando incluso cuando componentes individuales fallan.

Finalmente, la automatización mediante DevOps no es opcional sino esencial. La complejidad operacional de gestionar decenas o cientos de servicios independientes demanda automatización extensiva en todos los aspectos: *pipelines* de CI/CD (Continuous Integration/Continuous Deployment) para despliegues frecuentes y confiables, monitoreo y observabilidad para detectar problemas rápidamente, y gestión declarativa de infraestructura para mantener consistencia entre ambientes.

2.1.3. Patrones Arquitectónicos

Los patrones arquitectónicos fundamentales en microservicios abordan desafíos específicos de sistemas distribuidos. El patrón API Gateway proporciona un punto de entrada único que actúa como fachada para múltiples servicios *backend*, manejando aspectos transversales como autenticación, limitación de tasa y enrutamiento. Service Discovery permite la localización dinámica de servicios mediante registros centralizados o distribuidos, eliminando la necesidad de configuración estática de *endpoints*. El Saga Pattern coordina transacciones distribuidas mediante secuencias de transacciones locales compensables, resolviendo la ausencia de transacciones ACID entre servicios. Por último, el Sidecar Pattern desacopla funcionalidades auxiliares como registro de eventos, métricas y seguridad en contenedores separados que acompañan al servicio principal, facilitando su reutilización y actualización independiente.

A pesar de sus ventajas, los microservicios introducen *overhead* (sobrecarga) significativo:

latencia de red acumulativa, costo de serialización/deserialización, y complejidad de depuración distribuida. Estos desafíos se magnifican con lenguajes interpretados, motivando las técnicas de micro-optimización de esta investigación.

2.2. Contenerización y Docker

2.2.1. Fundamentos de la Contenerización

Docker transformó el despliegue de aplicaciones mediante contenerización ligera y portable. A diferencia de las máquinas virtuales que virtualizan hardware completo, Docker utiliza virtualización a nivel de sistema operativo (SO) mediante *namespaces* para el aislamiento de procesos, red y sistema de archivos, combinado con *cgroups* para establecer límites precisos de CPU, memoria e E/S. Esta aproximación resulta en contenedores que arrancan en milisegundos en lugar de minutos, con *overhead* mínimo comparado con máquinas virtuales tradicionales.

2.2.2. Arquitectura de Docker

El ecosistema Docker consta de varios componentes clave:

Docker Engine: El tiempo de ejecución que gestiona el ciclo de vida de los contenedores se compone de tres elementos principales. El Docker daemon (dockerd) ejecuta como proceso en background y gestiona los objetos Docker: imágenes, contenedores, redes y volúmenes. La interfaz de línea de comandos (Docker CLI) permite la interacción con el sistema, traduciendo comandos de usuario en llamadas API al daemon. Por último, containerd opera como tiempo de ejecución de bajo nivel, gestionando el ciclo de vida del contenedor y la interacción con el kernel del sistema operativo.

Docker Images: Plantillas read-only que contienen el sistema de archivos del contenedor. Utilizan un sistema de capas donde cada instrucción en el Dockerfile crea una nueva capa, permitiendo reutilización eficiente de espacio.

Docker Registry: Repositorios para almacenar y distribuir imágenes. Docker Hub es el registro público más grande, pero organizaciones típicamente mantienen registros privados.

Docker Compose: Herramienta para definir y ejecutar aplicaciones multi-contenedor mediante archivos YAML declarativos.

2.2.3. Sistema de Archivos por Capas

Docker utiliza un sistema de archivos union (UnionFS) que permite superponer múltiples sistemas de archivos:

1. **Capa base:** Típicamente un sistema operativo mínimo
2. **Capas intermedias:** Cada comando RUN, COPY o ADD crea una nueva capa
3. **Capa de contenedor:** Capa escribible donde se guardan cambios durante ejecución

Este diseño permite que múltiples contenedores compartan capas comunes, reduciendo significativamente el uso de disco y acelerando las descargas.

2.2.4. Técnicas de Optimización de Imágenes

La optimización de imágenes Docker es importante para el rendimiento y eficiencia de los microservicios contenerizados. Imágenes más pequeñas resultan en tiempos de descarga y arranque más rápidos, menor uso de almacenamiento y reducción de la superficie de ataque desde una perspectiva de seguridad. Las siguientes técnicas han demostrado ser efectivas en la práctica.

Construcción multi-stage representa una técnica fundamental que separa el entorno de construcción del tiempo de ejecución en etapas discretas. La primera etapa incluye compiladores y todas las herramientas de construcción necesarias, las etapas intermedias realizan la compilación y generan los artefactos ejecutables, mientras que la etapa final copia únicamente los binarios necesarios para la ejecución. Esta aproximación resulta típicamente en reducciones de hasta 90 % en el tamaño de la imagen final, eliminando toda la cadena de herramientas de desarrollo del contenedor de producción.

Imágenes *distroless* llevan la minimización al extremo al contener únicamente la aplicación y sus dependencias de tiempo de ejecución exactas. Estas imágenes carecen de *shell*, gestores de paquetes o cualquier utilidad del sistema que no sea estrictamente necesaria para la ejecución. Además de reducir la superficie de ataque desde una perspectiva de seguridad, estas imágenes son típicamente 60-80 % más pequeñas que las imágenes tradicionales basadas en distribuciones completas.

La **selección de imagen base** constituye otra decisión importante que impacta directamente el tamaño y rendimiento. Alpine Linux se ha establecido como una opción

popular, ofreciendo una distribución minimalista de aproximadamente 5MB basada en musl libc. Para aplicaciones completamente estáticas, la imagen *scratch* proporciona un contenedor literalmente vacío donde solo existe el binario de la aplicación. Debian slim ofrece un punto medio, proporcionando mejor compatibilidad con aplicaciones que esperan una distribución más tradicional mientras mantiene un tamaño razonable.

2.2.5. Orquestación con Kubernetes

Para gestionar microservicios a escala, Kubernetes proporciona una plataforma integral de orquestación. La programación automática de contenedores en nodos disponibles optimiza la utilización de recursos del clúster, mientras que el descubrimiento de servicios y balanceo de carga integrados eliminan la necesidad de configuración manual de *endpoints*. La gestión declarativa del estado deseado permite especificar qué debe ejecutarse sin preocuparse por cómo lograrlo, facilitando operaciones a escala. El escalado automático horizontal basado en métricas asegura que las aplicaciones puedan responder dinámicamente a cambios en la demanda, y las actualizaciones progresivas con reversiones automáticas garantizan actualizaciones sin tiempo de inactividad con la capacidad de revertir rápidamente en caso de problemas.

2.3. Lenguajes Interpretados vs Compilados

2.3.1. Modelos de Ejecución y Overhead

Los lenguajes compilados (C, Go, Rust) traducen código a instrucciones máquina nativas antes de la ejecución, ofreciendo máximo rendimiento. Los interpretados ejecutan código línea por línea en tiempo de ejecución, mientras que los híbridos usan *bytecode* o compilación JIT.

Python: Utiliza Cython con compilación a *bytecode* e interpretación posterior. El GIL limita el paralelismo real, forzando ejecución secuencial incluso en sistemas multiprocesador. Estudios empíricos demuestran un *overhead* típico de 10-100x más lento que C para cargas intensivas en CPU (Stoico et al., 2024), con variaciones significativas según el tipo de operación.

Node.js: Emplea el motor V8 de Google con compilación JIT adaptativa, donde el código çaliente"(frecuentemente ejecutado) se compila progresivamente a código máquina. El *overhead* típico oscila entre 2-10x más lento que código nativo, con mejoras

sustanciales cuando las optimizaciones especulativas del JIT aciertan en sus predicciones (Zanini, 2024).

Java: La JVM implementa un sofisticado sistema de compilación en niveles (*tiered compilation*) que evoluciona dinámicamente según los patrones de ejecución observados. El proceso comienza con interpretación pura del *bytecode*, transicionando gradualmente hacia el compilador C1 para optimizaciones rápidas de baja latencia, y finalmente al compilador C2 (anteriormente conocido como HotSpot Server Compiler) para optimizaciones agresivas en código crítico. Estudios empíricos y *benchmarks* rigurosos demuestran que el *overhead* inicial típicamente oscila entre 1.5-3x comparado con C++ optimizado, reduciéndose significativamente tras la fase de calentamiento cuando el compilador C2 aplica optimizaciones avanzadas como *inlining* agresivo, eliminación de chequeos de límites y vectorización automática (Georges et al., 2007). Este modelo de optimización adaptativa permite que Java alcance rendimiento competitivo mientras mantiene las ventajas de portabilidad y gestión automática de memoria.

El *overhead* se manifiesta en rendimiento computacional, uso de memoria (50-200MB base + 2-3x por objeto) y latencia de inicio (Python: 100-500ms, Node.js: 50-200ms, Java: 1-3s).

2.4. Herramientas de Compilación

Python: El ecosistema Python ofrece múltiples aproximaciones para mejorar el rendimiento. Nuitka destaca por su capacidad de compilar Python estándar a C++ sin requerir cambios en el código fuente, logrando mejoras típicas de 2-4x y generando ejecutables independientes completamente autocontenidos. Por otro lado, Cython adopta un enfoque híbrido que requiere anotaciones de tipos pero puede alcanzar mejoras de 5-100x en código numérico intensivo. PyPy representa una alternativa con su JIT compiler que ofrece aceleraciones de 4-5x, aunque sufre de problemas de compatibilidad con extensiones C que limitan su adopción en proyectos con dependencias complejas.

Node.js: Las estrategias de optimización para Node.js combinan empaquetado y optimizaciones del tiempo de ejecución. La herramienta pkg permite empaquetar la aplicación junto con el tiempo de ejecución V8 en un binario único, eliminando completamente la necesidad de tener Node.js instalado en producción y simplificando el despliegue. Complementariamente, las optimizaciones nativas de V8 como TurboFan JIT, inline caching y hidden classes pueden activarse mediante flags específicos para mejorar

el rendimiento. Los empaquetadores modernos como webpack y esbuild contribuyen reduciendo el tamaño del código y optimizando los tiempos de carga mediante técnicas como *tree shaking* y minificación.

Java: El ecosistema Java ha evolucionado significativamente en optimizaciones para microservicios. GraalVM Native Image representa el cambio más significativo, ofreciendo compilación AOT (Ahead-Of-Time, compilación anticipada) a binarios nativos que arrancan en milisegundos en lugar de segundos, aunque con limitaciones importantes en el uso de reflexión que requieren configuración explícita. Para aplicaciones que mantienen la JVM tradicional, el ajuste mediante banderas de optimización y Class Data Sharing (CDS) puede reducir significativamente los tiempos de inicio. Mirando al futuro, Project Leyden promete mejorar significativamente el arranque de aplicaciones Java mediante la preservación de estado pre-computado entre ejecuciones.

2.5. Trabajos Relacionados

2.5.1. Investigación Existente

La investigación en optimización de microservicios se ha enfocado principalmente en nivel sistema: provisión dinámica de recursos mediante aprendizaje por refuerzo distribuido (Bai et al., 2024), ubicación óptima de servicios (Wang et al., 2021), y políticas de escalado automático avanzadas. Sin embargo, estas aproximaciones no abordan las ineficiencias inherentes de lenguajes interpretados.

Investigaciones sobre el *overhead* de contenerización muestran resultados mixtos que varían según la carga de trabajo. Felter et al. (Felter et al., 2015) demostraron que los contenedores Linux pueden igualar o superar el rendimiento de máquinas virtuales en la mayoría de casos, mientras que Morabito et al. (Morabito et al., 2015) encontraron diferencias significativas dependiendo del tipo de carga. El *overhead* es generalmente mínimo para CPU y memoria (2-3 %) pero puede ser significativo para operaciones E/S intensivas debido a las capas adicionales de abstracción. Sin embargo, la interacción específica entre contenedores y lenguajes interpretados permanece poco explorada en la literatura académica, representando una brecha importante en nuestro entendimiento.

Las investigaciones recientes sobre técnicas de compilación para lenguajes interpretados revelan un potencial significativo para mejoras de rendimiento. Zhang et al. (Zhang et al., 2024) documentaron mejoras del 200-400 % utilizando Nuitka en benchmarks numéricos

intensivos, demostrando que la compilación anticipada puede transformar el rendimiento de Python en aplicaciones computacionales. Complementariamente, análisis detallados del motor V8 de JavaScript (Zanini, 2024) han revelado que las configuraciones óptimas del compilador JIT son altamente sensibles a las características específicas de la carga de trabajo, sugiriendo que no existe una configuración universalmente superior.

2.5.2. Brecha Identificada

Li (Li, 2024) identificó una brecha crítica: mientras existe extenso trabajo en optimización del lado del desarrollo y del lado de operaciones, hay poca investigación sobre optimización durante el proceso de contenerización mismo. Esta brecha es relevante porque el proceso de contenerización define el entorno completo, permite optimizaciones sin modificar código fuente, y los beneficios se acumulan para todos los despliegues futuros.

La presente investigación busca llenar esta brecha evaluando sistemáticamente el potencial de compilación durante contenerización para microservicios en lenguajes interpretados.

Capitulo 3

Metodología

3.1. Enfoque de Investigación

Esta investigación adopta un enfoque experimental cuantitativo para evaluar sistemáticamente el impacto de las técnicas de micro-optimización en microservicios contenerizados. El diseño experimental se basa en la comparación directa entre implementaciones estándar y versiones optimizadas de microservicios desarrollados en Python, Node.js y Java.

3.1.1. Justificación del Enfoque

El enfoque cuantitativo experimental se seleccionó por múltiples razones fundamentales que lo hacen idóneo para esta investigación. La objetividad en la medición constituye la ventaja principal, ya que las métricas de rendimiento como latencia, throughput y uso de recursos son inherentemente cuantificables, permitiendo comparaciones objetivas y libres de sesgo entre diferentes configuraciones. Esta objetividad se complementa con la reproducibilidad que ofrece un diseño experimental riguroso, permitiendo que otros investigadores repliquen los resultados para validar o refutar los hallazgos, fortaleciendo así la credibilidad científica del estudio.

Adicionalmente, el ambiente controlado de laboratorio proporciona la capacidad de aislar el efecto específico de las optimizaciones evaluadas, eliminando factores confundentes que podrían influir en el rendimiento. Finalmente, la recolección sistemática de datos cuantitativos habilita la aplicación de pruebas estadísticas robustas para determinar no solo la magnitud de las diferencias observadas, sino también su significancia estadística,

distinguiendo entre variaciones aleatorias y efectos reales de las optimizaciones.

3.1.2. Diseño Experimental

El experimento sigue un diseño factorial 3x3x3:

- **Factor 1 - Lenguaje:** Python, Node.js, Java
- **Factor 2 - Tipo de carga:** CPU-intensiva, I/O-intensiva, Servicio web
- **Factor 3 - Nivel de optimización:** Estándar, Optimizado, Compilado

Esto resulta en 27 configuraciones experimentales que permiten evaluar tanto efectos principales como interacciones entre factores.

3.1.3. Variables del Estudio

Variables independientes:

- Técnica de optimización aplicada (Nuitka, Cython, pkg, GraalVM)
- Configuración del contenedor (imagen base, flags de compilación)
- Tipo de carga de trabajo

Variables dependientes:

- Tiempo de respuesta y latencia
- Rendimiento (solicitudes por segundo)
- Utilización de CPU y memoria
- Tiempo de inicio del servicio
- Tamaño de la imagen de contenedor

Variables controladas:

- Hardware de prueba (especificaciones fijas)
- Sistema operativo host
- Versión de Docker
- Condiciones de red (pruebas locales)
- Carga de trabajo sintética reproducible

3.2. Principios de Micro-optimización

La investigación se fundamenta en tres principios de micro-optimización identificados por Li (2024) (Li, 2024), cada uno abordando una dimensión diferente de la optimización de contenedores:

3.2.1. Just-enough Containerisation

Este principio busca incluir únicamente los componentes esenciales para la ejecución del microservicio, eliminando todo lo superfluo:

La selección de la imagen base constituye la decisión más fundamental en la optimización de contenedores. Alpine Linux emerge como la opción preferida para la mayoría de casos, ofreciendo una distribución minimalista de aproximadamente 5MB basada en musl libc y BusyBox que proporciona funcionalidad esencial sin sobrecarga. Para aplicaciones que no requieren acceso a shell o utilidades del sistema, las imágenes distroless representan el siguiente nivel de minimalismo, incluyendo únicamente el runtime necesario para la aplicación. En el extremo más austero, la imagen scratch permite desplegar binarios estáticos sin ninguna dependencia externa, ideal para lenguajes compilados como Go que pueden generar ejecutables autocontenidos.

Los multi-stage builds transformaron la construcción de imágenes Docker al permitir una separación clara de responsabilidades entre las fases de construcción y ejecución. La primera etapa utiliza una imagen completa que incluye compiladores, herramientas de construcción y todas las dependencias de desarrollo necesarias. En la segunda etapa se realiza la compilación propiamente dicha y la generación de artefactos ejecutables. Finalmente, la tercera etapa construye la imagen mínima de producción, copiando únicamente los binarios compilados y las dependencias de tiempo de ejecución estrictamente necesarias desde las etapas anteriores, resultando en reducciones de tamaño típicamente superiores al 80 %

La eliminación sistemática de componentes innecesarios requiere un análisis cuidadoso de qué elementos son verdaderamente esenciales para la operación en producción. Este proceso comienza con la remoción de documentación, ejemplos y archivos de test que, aunque útiles durante el desarrollo, no aportan valor en runtime. Los cachés de gestores de paquetes como apt, yum o npm representan otro objetivo prioritario, frecuentemente consumiendo cientos de megabytes sin beneficio operacional. Las herramientas de

depuración, aunque tentadoras de mantener "por si acaso", deben excluirse en favor de técnicas de observabilidad externas. Finalmente, la optimización de capas mediante la combinación inteligente de comandos RUN no solo reduce el tamaño final sino que también mejora la eficiencia del caché de Docker durante reconstrucciones.

3.2.2. Just-for-me Configuration

Personalización de configuraciones según las características específicas de cada microservicio.

La selección de optimizaciones específicas para cada lenguaje se basó en un análisis exhaustivo de las mejores prácticas documentadas en la comunidad y *benchmarks* previos reportados en la literatura (Docker Inc., 2024; Williams & Garcia, 2018). Para Python, se identificaron las variables de entorno y flags del intérprete con mayor impacto en rendimiento de producción. En Node.js, se analizaron las banderas de V8 más relevantes para cargas típicas de microservicios. Para Java, se priorizaron configuraciones que reconocen y respetan los límites de recursos impuestos por contenedores Docker. Estas optimizaciones representan el conocimiento acumulado de la industria sobre cómo extraer el máximo rendimiento de cada plataforma sin modificar el código fuente.

Las optimizaciones específicas para Python se centran en reducir el *overhead* del intérprete y mejorar el comportamiento en producción (Python Software Foundation, 2024). La variable de entorno `PYTHONOPTIMIZE=2` elimina tanto *docstrings* como *assert statements*, reduciendo el tamaño del *bytecode* y acelerando la carga de módulos. Complementariamente, `PYTHONDONTWRITEBYTECODE=1` previene la generación de archivos .pyc que son innecesarios en contenedores inmutables y pueden causar problemas de permisos. La precompilación de módulos estándar frecuentemente utilizados durante la construcción de la imagen ahorra tiempo de inicio, mientras que la configuración personalizada del recolector de basura, ajustada según los patrones de memoria observados, puede reducir pausas y mejorar el rendimiento sostenido.

La optimización de Node.js se enfoca en el ajuste fino del motor V8 para maximizar el rendimiento dentro de las restricciones del contenedor (Node.js Foundation, 2024). El flag `-max-old-space-size` permite configurar el heap máximo según la memoria asignada al contenedor, evitando tanto el desperdicio como los errores por falta de memoria. Para entornos con recursos limitados, `-optimize-for-size` instruye a V8 a priorizar el uso eficiente de memoria sobre la velocidad de ejecución. El control granular del inlining

mediante `-turbo-inline-budget-scaling` balancea entre optimizaciones agresivas y tamaño del código generado. Finalmente, establecer `NODE_ENV=production` no es solo una convención sino que activa optimizaciones específicas y desactiva funcionalidades de desarrollo que impactan negativamente el rendimiento.

El tuning de la JVM para contenedores requiere configuraciones específicas que reconozcan y respeten las limitaciones del entorno. El flag `-XX:+UseContainerSupport` es fundamental, permitiendo que la JVM detecte automáticamente los límites de CPU y memoria impuestos por cgroups, evitando el problema histórico de la JVM asumiendo recursos del host completo. La configuración `-XX:MaxRAMPercentage` define qué porcentaje de la memoria del contenedor puede usar la JVM, típicamente establecido en 75-80 % para dejar espacio al sistema operativo y otros procesos. Para aplicaciones con muchos strings duplicados, `-XX:+UseStringDeduplication` puede reducir significativamente el uso de memoria mediante la detección y consolidación automática de strings idénticos. Finalmente, `-Xshare:on` activa Class Data Sharing, precargando clases del JDK en memoria compartida para acelerar el arranque de la aplicación.

3.2.3. Compilación Justo a Tiempo

Compilación de componentes interpretados durante la construcción del contenedor:

La motivación fundamental para la compilación just-in-time durante la contenerización radica en una observación clave: el entorno de producción queda completamente determinado en el momento de construir la imagen. A diferencia del desarrollo tradicional que debe mantener portabilidad, durante la contenerización se conoce con precisión la arquitectura del procesador target (sea x86_64 o ARM64), el sistema operativo específico y su versión exacta, así como el conjunto completo de bibliotecas del sistema que estarán disponibles. Esta certeza sobre el entorno de ejecución elimina la necesidad de mantener portabilidad post-build, habilitando optimizaciones agresivas específicas de plataforma que serían imposibles en un contexto de distribución tradicional.

La implementación de compilación just-in-time varía significativamente según el lenguaje y sus características únicas. Para Python, la estrategia implica usar Nuitka para compilar el código completo a C++, o alternativamente Cython para optimizar selectivamente módulos críticos que representan cuellos de botella identificados. Node.js adopta un enfoque diferente mediante pkg, que empaqueta tanto la aplicación como el runtime V8

en un binario standalone completamente autocontenido. Java, con su ecosistema más maduro para compilación anticipada, aprovecha GraalVM Native Image para realizar compilación ahead-of-time (AOT) completa, transformando bytecode Java en código máquina nativo.

Los beneficios esperados de esta aproximación son múltiples y sinérgicos. La eliminación del *overhead* de interpretación representa la ventaja más directa, permitiendo que el código se ejecute a velocidades cercanas al código nativo. Este beneficio se amplifica con la reducción significativa en los tiempos de inicio, ya que no es necesario cargar e inicializar intérpretes o máquinas virtuales. El uso de memoria también se optimiza considerablemente al eliminar la necesidad de mantener el intérprete completo en memoria durante la ejecución. Finalmente, la capacidad de aplicar optimizaciones específicas de plataforma, como extensiones del conjunto de instrucciones o configuraciones de caché particulares del procesador objetivo, puede proporcionar mejoras adicionales imposibles en un contexto interpretado.

3.3. Selección de Lenguajes y Herramientas

3.3.1. Criterios de Selección de Lenguajes

Los lenguajes fueron seleccionados para representar diferentes paradigmas de ejecución comunes en microservicios:

- **Python** fue seleccionado como representante de lenguajes interpretados puros:
  - Su uso común en data science, APIs REST y scripting lo convierte en un caso de estudio relevante para la industria
  - Representa el caso más extremo de *overhead* de interpretación entre los lenguajes comunes, lo que permite evaluar mejor el impacto de las optimizaciones
  - El ecosistema maduro con herramientas como Nuitka, Cython y PyPy ofrece opciones para probar diferentes enfoques de compilación
- **Node.js** representa la categoría de lenguajes con compilación JIT:
  - Su uso dominante en aplicaciones web modernas y APIs lo hace importante para cualquier estudio sobre microservicios

  - El motor V8 que usa Node.js es muy avanzado en compilación JIT, balanceando rendimiento y facilidad de desarrollo
  - Esto permite evaluar cómo las técnicas de pre-compilación funcionan con un JIT compiler ya optimizado

- **Java** completa el grupo como representante de lenguajes basados en bytecode y máquina virtual:
  - Su posición como estándar en microservicios empresariales hace que los resultados sean útiles en la práctica
  - La JVM HotSpot, con décadas de optimizaciones, presenta el desafío de cómo mejorar algo ya muy optimizado
  - El problema del warm-up time es importante en contenedores donde los servicios pueden reiniciarse frecuentemente
  - La disponibilidad de GraalVM Native Image permite comparar el modelo tradicional de JVM contra la compilación ahead-of-time (AOT) completa

3.3.2. Herramientas de Compilación Seleccionadas

Para cada lenguaje se evaluaron múltiples herramientas, seleccionando las más maduras y apropiadas:

Python - Herramientas evaluadas

Para Python se evaluaron tres aproximaciones principales de compilación y optimización. *Nuitka* fue seleccionada como herramienta principal por su capacidad única de compilar Python estándar a C++ sin requerir cambios en el código fuente. Esta característica es fundamental para mantener la filosofía de optimización del lado de operaciones. Además, Nuitka ofrece soporte completo para Python 3.11, genera ejecutables standalone autocontenidos, mantiene buena compatibilidad con las librerías estándar de Python, y cuenta con desarrollo activo y releases frecuentes que aseguran soporte continuo.

Cython fue incluido como punto de comparación adicional debido a su enfoque diferente: permite anotaciones de tipo que habilitan optimizaciones agresivas, especialmente efectivas para código numérico intensivo. Sin embargo, su requerimiento de modificar el código fuente lo posiciona más como una herramienta de desarrollo que de operaciones, sirviendo principalmente para establecer un límite superior de rendimiento alcanzable.

PyPy fue evaluado pero finalmente descartado por varias razones críticas. Las incompatibilidades frecuentes con extensiones C populares como NumPy limitan severamente su aplicabilidad en microservicios reales. Adicionalmente, su modelo de memoria resulta en un consumo significativamente mayor que Cython, problema agravado en entornos contenerizados con recursos limitados. La complejidad adicional de mantener un tiempo de ejecución alternativo en contenedores también contribuyó a su exclusión del estudio.

Node.js - Herramientas evaluadas

El ecosistema Node.js ofrece menos opciones de compilación anticipada comparado con Python, pero las disponibles son altamente efectivas. *pkg* emergió como la solución óptima por su capacidad de empaquetar tanto la aplicación como el tiempo de ejecución V8 completo en un binario único y autocontenido. Esta herramienta destaca por su soporte multiplataforma que detecta automáticamente la arquitectura objetivo, la eliminación completa de la dependencia de Node.js en producción, y su configuración minimalista que se integra naturalmente con package.json existente.

La herramienta *Nexe* fue considerada inicialmente como alternativa, pero se descartó tras identificar limitaciones significativas. El desarrollo menos activo de la herramienta se traduce en soporte deficiente para versiones recientes de Node.js. Los problemas documentados con módulos nativos representan un obstáculo mayor para aplicaciones reales. Fundamentalmente, pkg representa una evolución más madura y mantenida del mismo concepto.

Complementariamente, las *optimizaciones nativas de V8* se aplicaron mediante banderas específicas del motor JavaScript. Estas optimizaciones no requieren herramientas adicionales y se combinan sinérgicamente con pkg, permitiendo ajustar el comportamiento del JIT compiler, la gestión de memoria y las estrategias de optimización especulativa según las características de cada microservicio.

Java - Herramientas evaluadas

Java presenta el ecosistema más maduro para compilación anticipada gracias a *GraalVM Native Image*, seleccionado como herramienta principal. Esta tecnología permite compilación ahead-of-time (AOT) completa a binarios nativos, eliminando completamente el tiempo de inicio de la JVM que típicamente oscila entre 1-3 segundos. La reducción del footprint de memoria es igualmente significativa, pasando de cientos de megabytes a decenas. Sin embargo, las limitaciones con reflexión y carga dinámica

de clases requieren configuración explícita, lo que resultó problemático para frameworks complejos como Spring Boot.

OpenJ9 de IBM fue evaluado como alternativa por su diseño específicamente optimizado para entornos contenerizados y *cloud*. A pesar de ofrecer mejor gestión de memoria y tiempos de inicio más rápidos que HotSpot, finalmente se descartó porque mantiene el modelo de máquina virtual, no eliminando el *overhead* fundamental de interpretación inicial. Además, su menor adopción en la industria comparado con HotSpot limita el soporte comunitario y la compatibilidad con herramientas.

Las *optimizaciones JVM estándar* con HotSpot sirvieron como línea base de comparación, aplicando las mejores prácticas para contenedores (De Melo Junior, 2024): banderas como -XX:+UseContainerSupport para reconocimiento de límites *cgroup*, Class Data Sharing (CDS) para acelerar el arranque compartiendo metadatos pre-procesados, y configuraciones de memoria adaptadas al entorno contenerizado. Estas optimizaciones representan lo mejor alcanzable sin abandonar el modelo tradicional de JVM.

3.4. Diseño de Benchmarks

Los *benchmarks* se diseñaron para evaluar diferentes aspectos del rendimiento de microservicios, desde operaciones aisladas hasta aplicaciones completas.

3.4.1. Micro-*benchmarks*

Los micro-*benchmarks* fueron diseñados específicamente para este estudio con el objetivo de aislar y medir aspectos particulares del rendimiento. Cada *benchmark* se implementó siguiendo patrones establecidos en la literatura de evaluación de rendimiento, adaptados para maximizar la sensibilidad a las optimizaciones evaluadas. El código fuente completo de estos *benchmarks* se encuentra disponible en el repositorio del proyecto y se detalla en el Anexo A.1.

El *benchmark* intensivo en CPU se implementó mediante el clásico algoritmo de Fibonacci recursivo con n=35, elegido específicamente por sus características computacionales extremas. Este algoritmo genera un árbol de llamadas particularmente profundo que estresa tanto la pila como la CPU, con una complejidad $\Theta(2^n)$ que resulta en aproximadamente 9 millones de llamadas recursivas. La ausencia de E/S y asignación significativa de memoria asegura que las mediciones reflejen puramente el *overhead*

de interpretación y gestión de llamadas. Las métricas capturadas incluyen el tiempo total de ejecución y el uso de CPU, proporcionando una visión clara del impacto de las optimizaciones en código computacionalmente intensivo.

El *benchmark* intensivo en E/S evalúa el rendimiento en operaciones de archivo mediante un patrón controlado de lectura y escritura. Cada iteración procesa archivos de 10MB a través de un ciclo read-process-write, repitiendo este patrón 100 veces para obtener mediciones estadísticamente significativas. El diseño minimiza deliberadamente el procesamiento CPU para aislar el overhead específico de system calls y operaciones de buffering, permitiendo evaluar cómo las diferentes técnicas de compilación afectan la eficiencia de I/O. Las métricas principales incluyen operaciones por segundo y latencia por operación, capturando tanto throughput como responsividad.

El benchmark de servicio web representa el escenario más realista, implementando una API REST completa con los frameworks estándar de cada lenguaje: Flask para Python, Express para Node.js y Javalin para Java. Para todos los lenguajes, se diseñó un endpoint para el cálculo de Fibonacci, el cual fue diseñado para combinar un test de network I/O con procesamiento computacional, reflejando cargas de trabajo típicas de microservicios. El código detallado de estos endpoints se encuentra en el Anexo A.4.2. La carga se genera mediante Apache Bench configurado para 10,000 requests con concurrencia de 100 conexiones simultáneas, simulando condiciones de producción moderadas. Las métricas capturadas incluyen requests por segundo para evaluar throughput máximo, así como latencias promedio y máxima para entender la consistencia del rendimiento bajo carga.

3.4.2. Macro-benchmarks - Bookinfo

Los macro-benchmarks utilizan Bookinfo, la cual es una aplicación de microservicios realista de Istio que simula una tienda de libros[1]. La arquitectura de esta aplicación comprende tres microservicios principales que interactúan para proporcionar una experiencia completa de tienda de libros. El servicio **productpage**, implementado en Python con Flask, actúa como frontend agregando información de los demás servicios. El servicio **reviews**, desarrollado en Java con Spring Boot, gestiona las reseñas de usuarios con diferentes versiones que demuestran patrones de comunicación variables. Finalmente, **ratings**, construido con Node.js y Express, proporciona las calificaciones numéricas con estrellas para cada libro.

[1]Código fuente disponible en: https://github.com/nocalhost/bookinfo

El flujo de requests en Bookinfo sigue un patrón de agregación típico de arquitecturas de microservicios. Cuando un cliente solicita una página de producto, productpage actúa como orquestador, realizando llamadas paralelas a los servicios downstream. El servicio reviews, dependiendo de su versión (v2 o v3), puede realizar llamadas adicionales a ratings para enriquecer las reseñas con calificaciones numéricas. Finalmente, productpage agrega todas las respuestas recibidas y renderiza el HTML final, demostrando patrones de comunicación tanto paralelos como en cascada.

La selección de Bookinfo como macro-benchmark se justifica por múltiples factores que la hacen ideal para esta investigación. Primordialmente, representa patrones reales de comunicación entre servicios que se encuentran en aplicaciones de producción, incluyendo llamadas síncronas, agregación de datos y dependencias en cascada. La implementación de cada servicio en un lenguaje diferente permite la comparación directa del impacto de las optimizaciones en el mismo contexto operacional. Además, al ser una aplicación bien documentada y activamente mantenida por la comunidad Istio, garantiza reproducibilidad y relevancia continua. La variedad de patrones arquitectónicos que exhibe, desde agregación simple hasta renderizado de templates complejos, proporciona un escenario de prueba comprehensivo.

Para el estudio se aplicaron modificaciones específicas a cada servicio según su tecnología base. El servicio productpage fue compilado con Nuitka para evaluar el impacto en aplicaciones Python con Flask. Para ratings, se utilizó pkg para generar un binario standalone de Node.js, eliminando la necesidad del runtime en producción. El servicio reviews se optimizó mediante configuraciones avanzadas de JVM y, cuando fue posible, con GraalVM Native Image para comparar ambas aproximaciones.

3.5. Métricas de Evaluación

El estudio mide el impacto de las optimizaciones usando tres grupos de métricas. Primero, las métricas de rendimiento evalúan qué tan rápido trabaja cada servicio: tiempo de ejecución (promedio de 100 corridas), latencia de respuesta (cuánto espera el usuario) y throughput (solicitudes procesadas por segundo). Estas mediciones se hacen con Apache Bench para simular usuarios reales y permiten comparar si las versiones optimizadas son realmente más rápidas.

Segundo, las métricas de recursos miden cuánto hardware consume cada versión. Docker stats monitorea el uso de CPU y memoria cada segundo mientras los servicios procesan

solicitudes. También se compara el tamaño de las imágenes Docker entre versiones. Estos datos revelan si las mejoras de velocidad tienen un costo en mayor consumo de recursos, un trade-off crítico para decidir qué versión usar en producción.

Por último, las métricas de inicio evalúan cuánto tardan los servicios en estar listos. Se mide el tiempo desde que arranca el contenedor hasta que puede responder solicitudes HTTP, y se analiza cómo mejora el rendimiento durante las primeras 1000 solicitudes (curva de calentamiento). Esto es importante en servicios que se reinician frecuentemente o usan compilación JIT como Node.js y Java, donde el código se optimiza mientras se ejecuta.

3.6. Proceso Experimental

3.6.1. Configuración del Entorno

Hardware principal - MacBook Pro M4 Pro:

- Procesador: Apple M4 Pro
- Memoria: 24GB RAM unificada
- Almacenamiento: 512GB SSD NVMe
- Arquitectura: ARM64 (Apple Silicon)
- Sistema Operativo: macOS 14.5 Sonoma

Hardware secundario - Raspberry Pi 4:

- Procesador: Broadcom BCM2711, Quad-core Cortex-A72 @ 1.5GHz
- Memoria: 8GB LPDDR4
- Almacenamiento: 64GB microSD clase 10
- Arquitectura: ARM64
- Sistema Operativo: Raspberry Pi OS 64-bit (Debian)

Software base:

- Docker Engine: 24.0.2
- Docker Compose: 2.18.1

- Python: 3.11.4
- Node.js: 18.17.1
- Java: OpenJDK 11.0.19
- Apache Bench: 2.3

3.6.2. Protocolo de Construcción de Imágenes

Para cada combinación lenguaje/optimización:

1. Preparación:

Antes de construir cada imagen, se prepara el entorno para obtener mediciones confiables. Primero se limpia el caché de Docker con `docker system prune -a` para que todas las construcciones empiecen desde cero. Se verifica que haya al menos 10GB de espacio libre en disco para evitar que falle la construcción. También se sincroniza el reloj del sistema, paso importante para que las mediciones de tiempo sean precisas y se puedan comparar entre diferentes pruebas.

2. Construcción de variantes:

- **Standard:** Dockerfile base sin optimizaciones
- **Optimized:** Aplicando just-enough y just-for-me
- **Compiled:** Agregando just-in-time compilation

3. Validación:

Después de construir cada imagen optimizada, se valida que funcione correctamente. Se ejecutan las pruebas para verificar que toda la funcionalidad sigue trabajando bien. Se comparan los resultados con los de la versión estándar para confirmar que son idénticos, ya que las optimizaciones no deben cambiar el comportamiento del programa. Si se encuentra alguna incompatibilidad, se documenta para entender qué optimizaciones no funcionan con qué frameworks o bibliotecas.

3.6.3. Protocolo de Ejecución de Pruebas

Fase de warm-up:

- 10 iteraciones descartadas para estabilizar sistema

- Permite carga de bibliotecas en memoria
- Estabiliza frecuencia de CPU (thermal throttling)

Fase de medición:

- 100 ejecuciones consecutivas para micro-benchmarks
- 30 minutos de carga sostenida para macro-benchmarks
- Monitoreo continuo de recursos cada segundo
- Logs de ejecución con precisión de microsegundos

Entre ejecuciones:

- Detener y remover contenedores: `docker stop && docker rm`
- Esperar 10 segundos para liberar recursos
- Rotar logs para evitar llenado de disco
- Verificar temperatura CPU $<$70°C (Raspberry Pi)

3.6.4. Control de Variables

Para obtener resultados confiables fue necesario controlar todas las variables del experimento. Durante las pruebas, el sistema se aisló del entorno normal de trabajo. Se desactivaron las actualizaciones automáticas, se cerraron todos los programas que no eran necesarios, y se detuvo la indexación de archivos que podría usar recursos sin aviso. También se activó el modo avión para evitar cualquier interferencia de red o sincronización en la nube que pudiera afectar las mediciones.

Los recursos asignados a cada contenedor se mantienen idénticos entre todas las pruebas. Cada contenedor recibe exactamente 2 cores de CPU y un límite de 2GB de memoria, sin restricciones de I/O para poder medir el rendimiento máximo posible. La red utiliza el modo bridge de Docker con configuración idéntica para todos los servicios, asegurando que las diferencias en rendimiento no se deban a variaciones en recursos disponibles.

La reproducibilidad se garantiza mediante la automatización completa del experimento. Los scripts ejecutan todas las pruebas de forma idéntica, la configuración está versionada en Git para rastrear cualquier cambio, los generadores aleatorios usan seeds fijos para producir siempre los mismos datos, y todos los servicios procesan exactamente los

mismos datasets. Esto permite que cualquier investigador pueda replicar los resultados exactos del estudio.

3.7. Análisis de Datos

El análisis de datos comprende tres aspectos fundamentales: análisis estadístico, visualización de resultados, y garantías de reproducibilidad.

Análisis estadístico: Se diseñó para proporcionar validación rigurosa de las diferencias observadas entre variantes, yendo más allá de simples comparaciones porcentuales. Las pruebas estadísticas seleccionadas incluyen el cálculo de medias y desviaciones estándar como medidas descriptivas básicas. Para la inferencia estadística, se emplean pruebas t de Student para muestras independientes, que permiten determinar si existe una diferencia significativa entre las medias de dos grupos (Student, 1908). Estas pruebas son apropiadas para comparar el rendimiento entre versiones estándar y optimizadas. El nivel de significancia se estableció en $\alpha = 0.05$, siguiendo convenciones académicas estándar.

Crucialmente, se incluye el cálculo del tamaño del efecto mediante la d de Cohen, que cuantifica la magnitud de la diferencia entre dos grupos expresada en desviaciones estándar (Cohen, 1988). Esto permite distinguir entre diferencias estadísticamente significativas pero triviales en la práctica (d $<$0.2) y aquellas con relevancia práctica sustancial (d $>$0.8). La interpretación sigue las convenciones de Cohen: efecto pequeño (d = 0.2), mediano (d = 0.5) y grande (d = 0.8).

Visualización de resultados: Emplea técnicas estándar pero efectivas para comunicar los hallazgos. Los gráficos de barras permiten comparaciones visuales inmediatas del rendimiento entre diferentes configuraciones, mientras que las tablas detalladas proporcionan valores precisos para análisis más profundos. Todo el procesamiento y generación de visualizaciones se implementa mediante scripts de Python reproducibles, garantizando transparencia y permitiendo que otros investigadores regeneren exactamente las mismas figuras desde los datos crudos.

Garantías de reproducibilidad: La reproducción completa del estudio se garantiza mediante un repositorio público en GitHub que contiene todos los artefactos necesarios. El código fuente de los benchmarks está completamente documentado y organizado por lenguaje y tipo de prueba. Los Dockerfiles incluyen versiones fijas de todas las

dependencias, eliminando la variabilidad por actualizaciones futuras. Los scripts de automatización permiten ejecutar el experimento completo con un solo comando, mientras que los scripts de análisis procesan los datos crudos para generar todas las tablas y figuras presentadas.

Los datos crudos están disponibles en formato CSV con headers descriptivos, incluyendo metadatos de cada ejecución y logs completos de Docker. La documentación exhaustiva asegura que cualquier investigador pueda replicar el estudio independientemente de su experiencia previa. El README principal proporciona instrucciones paso a paso desde la configuración inicial hasta la generación de resultados finales. Los requisitos de hardware y software están claramente especificados, incluyendo versiones mínimas y configuraciones recomendadas. Una sección dedicada de troubleshooting aborda los problemas más comunes encontrados durante el desarrollo, ahorrando tiempo a futuros investigadores. Finalmente, la información de contacto facilita la comunicación directa para resolver dudas o reportar problemas no documentados.

Limitaciones metodológicas: Es importante reconocer que los micro-benchmarks son sintéticos y pueden no capturar la complejidad real de aplicaciones en producción. Los resultados son específicos a arquitectura ARM64 y las versiones de herramientas utilizadas. El estudio se enfoca exclusivamente en métricas de rendimiento, sin evaluar otros aspectos como mantenibilidad o costos de desarrollo.

Capitulo 4

Diseño Experimental

4.1. Arquitectura del Sistema de Pruebas

El sistema experimental implementa una arquitectura modular de tres capas para garantizar la reproducibilidad y automatización completa de los experimentos:

Capa de Orquestación: Coordina la ejecución mediante scripts maestros que gestionan el flujo experimental completo, desde la inicialización hasta el archivo de resultados. Utiliza configuración declarativa en YAML y logging centralizado con timestamps sincronizados.

Capa de Ejecución: Implementa los componentes operacionales incluyendo builders especializados para cada variante de optimización, ejecutores de benchmarks parametrizables, monitores continuos de recursos y verificadores de disponibilidad de servicios.

Capa de Análisis: Procesa los datos recolectados mediante parsers especializados y motores estadísticos. Las pruebas t de Student permiten comparaciones pareadas entre implementaciones estándar y optimizadas. ANOVA se utiliza cuando se comparan más de dos grupos (por ejemplo, standard vs. optimized vs. compiled). La d de Cohen proporciona una medida del tamaño del efecto que complementa la significancia estadística, permitiendo evaluar la relevancia práctica de las mejoras observadas. Los resultados se presentan mediante generadores automáticos de visualizaciones e informes en formatos HTML y LaTeX.

4.2. Configuración del Entorno

4.2.1. Plataformas de Experimentación

Los experimentos se ejecutaron en dos plataformas ARM64 complementarias:

MacBook Pro M4 Pro (Plataforma Principal):

- Procesador Apple M4 Pro: 12 cores (8 performance + 4 efficiency) @ 3.5GHz
- 24GB LPDDR5 unificada, 512GB NVMe SSD
- macOS 14.5 con Virtualization.framework nativo

Raspberry Pi 4 Model B (Validación en Dispositivos de Borde):

- Broadcom BCM2711: Quad-core Cortex-A72 @ 1.5GHz
- 8GB LPDDR4, MicroSD 64GB A2
- Raspberry Pi OS 64-bit con cooling activo

4.2.2. Stack de Software

Contenerización: Docker 24.0.2 con BuildKit habilitado y Docker Compose 2.18.1

Lenguajes y Compiladores:

- **Python 3.11.4:** Nuitka 1.8.0 (C++), Cython 3.0.2 (extensiones tipadas)
- **Java OpenJDK 11.0.19:** GraalVM CE 22.3.0 (native-image)
- **Node.js 18.17.1:** pkg 5.8.1 (binarios standalone)

Benchmarking: Apache Bench 2.3 con soporte SSL/TLS, docker stats API para monitoreo continuo

Red: Bridge Docker personalizado (172.20.0.0/16), DNS local, MTU estándar 1500 bytes

4.3. Implementación de Micro-benchmarks

4.3.1. Benchmark CPU-Intensivo: Fibonacci Recursivo

Se implementó Fibonacci recursivo (n=35) generando 9,227,465 llamadas recursivas. Esta elección maximiza el stress sobre el stack de llamadas sin introducir I/O, permitiendo evaluar puramente el overhead de interpretación.

```
# Python
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(
        n-1) + fibonacci(
        n-2)
```

```
// Node.js
function fibonacci(n) {
  if (n <= 1)
    return n;
  return fibonacci(
    n-1) + fibonacci(
    n-2);
}
```

```
// Java
public static long
 fibonacci(int n) {
 if (n <= 1) return n;
 return fibonacci(n-1)
   + fibonacci(n-2);
 }
```

4.3.2. Benchmark I/O-Intensivo: Operaciones de Archivo

Implementa un patrón read-modify-write sobre archivos de 10MB durante 100 iteraciones. Los archivos contienen datos binarios aleatorios generados con entropía criptográfica para evitar optimizaciones de compresión o deduplicación a nivel de sistema operativo. Este tamaño fue seleccionado para ser suficientemente grande para estresar el subsistema I/O pero lo suficientemente pequeño para completar las pruebas en tiempo razonable:

1. Lectura en chunks de 4KB (tamaño de página estándar)
2. Transformación XOR con valor constante
3. Escritura síncrona con flush forzado
4. Validación de integridad mediante checksum

Este diseño minimiza el procesamiento CPU para aislar el overhead de system calls y operaciones I/O del intérprete.

4.3.3. Benchmark de Servicio Web: API REST

Cada lenguaje implementa un endpoint para el cálculo de Fibonacci usando su framework web estándar (Flask, Express, Javalin). El código detallado de estos endpoints se

encuentra en el Anexo A.4.2. La configuración utiliza 4 workers en puerto 8080 con keep-alive habilitado.

Para evaluar el rendimiento de los servicios web se diseñó un protocolo de pruebas usando Apache Bench. El protocolo simula una carga realista enviando 10,000 solicitudes HTTP con 100 conexiones concurrentes, lo que representa usuarios simultáneos accediendo al servicio. Cada solicitud incluye un parámetro n aleatorio entre 10 y 30 para el cálculo de Fibonacci, evitando que los resultados se vean afectados por cachés u optimizaciones basadas en patrones repetitivos. Antes de las mediciones oficiales, se ejecutan 1,000 solicitudes de calentamiento que se descartan del análisis, permitiendo que los compiladores JIT y los sistemas de caché alcancen su estado óptimo. Durante la prueba se captura la latencia y el throughput máximo alcanzado, proporcionando una visión completa del rendimiento bajo carga.

4.4. Implementación de Macro-benchmarks: Bookinfo

Bookinfo (Istio Project, 2024), la aplicación de ejemplo de Istio, simula una tienda de libros completa con una arquitectura de microservicios realista[1]. La aplicación consta de cuatro servicios principales que interactúan entre sí, donde nos enfocaremos en tres. El servicio **productpage (Python)** actúa como frontend, agregando información de todos los demás servicios para presentar una página unificada al usuario. Este servicio se evaluó en dos variantes: Flask estándar contra una versión compilada con Nuitka. El servicio **reviews (Java)** maneja todas las reseñas escritas por usuarios, incluyendo su almacenamiento y recuperación. Aquí se comparó la JVM estándar contra una versión con flags de optimización específicos. Por último, el servicio **ratings (Node.js)** gestiona el sistema de calificaciones con estrellas, permitiendo a los usuarios puntuar libros del 1 al 5. Este servicio se probó con Express estándar y con una versión compilada a binario usando pkg.

4.4.1. Configuración de Optimizaciones

Servicio	**Standard**	**Optimizado**
productpage	Flask 2.0.1 + Gunicorn	Nuitka compilado + pre-fork
ratings	Express 4.17 + PM2	pkg binario + cluster mode
reviews	Spring Boot 2.5	JVM flags + CDS habilitado

[1]Código fuente disponible en: https://github.com/nocalhost/bookinfo

Todos los servicios utilizan 4 workers/instancias, red Docker bridge compartida sin service mesh para aislar el overhead de las optimizaciones.

4.5. Protocolo Experimental

4.5.1. Fases de Experimentación

Los experimentos comienzan estableciendo métricas base para cada servicio. Se mide el tiempo de inicio desde docker run hasta que el servicio responde, seguido de 2,000 solicitudes de calentamiento para estabilizar el sistema. Después se ejecutan 10,000 solicitudes analizando percentiles de latencia para entender cómo se distribuye el rendimiento en condiciones normales.

Con las métricas base establecidas, se evalúa cómo responde cada servicio al aumentar la carga. Se aplican cargas progresivas con 1, 10, 50, 100 y 200 usuarios concurrentes, manteniendo cada nivel durante 5 minutos para que el sistema se estabilice. Este proceso continúa hasta encontrar el punto donde el servicio comienza a saturarse y su rendimiento se degrada.

Finalmente, se prueba si los servicios mantienen su rendimiento durante períodos extendidos. Los servicios se ejecutan durante 30 minutos al 80 % de su capacidad máxima mientras se monitorean continuamente los recursos. Esto permite detectar problemas como degradación gradual del rendimiento o fugas de memoria que solo aparecen después de operación prolongada.

4.5.2. Estrategia de Construcción Multi-stage

Todas las variantes utilizan construcción multi-stage de Docker para aplicar los tres principios de micro-optimización. La estrategia consiste en dos etapas claramente diferenciadas. En la primera etapa (builder) se incluyen todas las herramientas de compilación necesarias, se realiza la compilación del código y se generan los artefactos optimizados. La segunda etapa (runtime) parte de una imagen mínima como Alpine o Distroless, copia únicamente los binarios compilados de la etapa anterior y aplica la configuración específica para cada servicio. Esta separación permite que las imágenes finales sean mucho más pequeñas al no incluir compiladores ni herramientas de desarrollo que no se necesitan en producción.

4.5.3. Matriz Experimental

Lenguaje	**Micro-benchmarks**	**Macro-benchmarks**	**Total**
Python	3 tipos × 3 variantes = 9	productpage: 2 variantes	11
Node.js	3 tipos × 3 variantes = 9	ratings: 2 variantes	11
Java	3 tipos × 3 variantes = 9	reviews: 2 variantes	11
Total	27	6	33

4.5.4. Validación y Análisis

Para asegurar que los resultados fueran confiables, primero se verificó que las versiones optimizadas funcionaran correctamente. Se compararon los outputs de cada versión para confirmar que producían los mismos resultados. También se calcularon checksums MD5 de los archivos generados y se ejecutaron las pruebas automatizadas que ya tenía cada aplicación. Además, se hicieron pruebas manuales básicas para verificar que los servicios respondieran a solicitudes típicas. Todo esto confirmó que las optimizaciones mejoraban el rendimiento sin romper nada.

Los datos recolectados se analizaron con métodos estadísticos estándar para confirmar que las diferencias observadas fueran reales y no producto del azar. Primero se verificó con la prueba de Shapiro-Wilk, que evalúa si una muestra proviene de una distribución normal mediante la comparación de los cuantiles de los datos con los esperados (Shapiro & Wilk, 1965), confirmando que los datos siguieran una distribución normal. Luego con la prueba de Levene, que verifica la homogeneidad de varianzas entre grupos sin asumir normalidad (Levene, 1960), se confirmó que las varianzas fueran similares entre grupos. Con estos requisitos cumplidos, se usaron pruebas t pareadas para comparar cada versión estándar con su versión optimizada, usando un nivel de confianza del 95 % (α=0.05). Para complementar, la d de Cohen indicó qué tan grande era la diferencia encontrada, ayudando a distinguir entre mejoras pequeñas que no valen la pena y mejoras grandes que sí justifican el esfuerzo de optimización.

El experimento se diseñó para que otros investigadores pudieran repetirlo y obtener los mismos resultados. Los scripts que ejecutan las pruebas están en un repositorio Git público. Los logs guardan el momento exacto de cada medición para poder notar cualquier anomalía. El repositorio incluye las versiones exactas de todo el software usado, desde Docker hasta los compiladores. También, los números aleatorios se generan con seeds fijos, logrando que cada ejecución del experimento produzca exactamente los

mismos datos de prueba.

4.5.5. Gestión de Datos

Los resultados se organizan jerárquicamente por tipo de benchmark y lenguaje, utilizando JSON para datos estructurados, CSV para series temporales, y generación automática de visualizaciones y reportes en formatos PNG/HTML/Markdown.

Capitulo 5

Implementación y Resultados

5.1. Introducción

Este capítulo documenta la implementación del sistema experimental y presenta los resultados obtenidos al evaluar técnicas de micro-optimización en microservicios contenerizados. Los experimentos se ejecutaron tanto con micro-*benchmarks* sintéticos como con la aplicación Bookinfo (Istio Project, 2024), proporcionando una visión comprehensiva del impacto real de estas técnicas.

5.2. Implementación del Sistema

5.2.1. Arquitectura Desarrollada

El sistema experimental se construyó con una arquitectura modular de tres niveles que trabajan juntos para ejecutar y analizar los experimentos. En el nivel más alto, los scripts maestros como `run_enhanced_benchmarks.sh` actúan como coordinadores que controlan todo el proceso, desde construir las imágenes Docker hasta generar los reportes finales con los resultados. El nivel intermedio contiene el código de las aplicaciones organizadas por lenguaje, donde cada una tiene sus variantes estándar y optimizadas (como standard, nuitka para Python, pkg para Node.js, y graalvm para Java). En el nivel más bajo, las librerías compartidas se encargan de recolectar las métricas durante las pruebas, agregar todos los datos en archivos consolidados y generar automáticamente los gráficos y visualizaciones que facilitan el análisis de resultados.

5.2.2. Implementación de Compilación Just-in-Time

La implementación de compilación just-in-time durante la contenerización requirió estrategias específicas adaptadas a las características de cada tecnología. Para Python, se utilizó Nuitka para compilar el código a C++ generando binarios standalone completamente autocontenidos. La automatización del proceso se logró mediante el flag `-assume-yes-for-downloads`, eliminando intervenciones manuales durante la construcción de imágenes.

Node.js presentó un enfoque diferente mediante pkg, que empaqueta tanto el runtime V8 completo como la aplicación en un binario único ejecutable. Una ventaja clave fue la detección automática de la arquitectura target, permitiendo compilación transparente tanto para ARM64 como x64 sin modificaciones en el proceso de build.

Java demostró la mayor complejidad, requiriendo estrategias diferenciadas según el contexto. Mientras que GraalVM Native Image funcionó exitosamente para los micro-benchmarks simples, las aplicaciones con Spring Boot requirieron mantener las optimizaciones JVM tradicionales debido a las incompatibilidades fundamentales entre la compilación AOT y el uso extensivo de reflection en el framework.

5.2.3. Desafíos de Implementación

La implementación enfrentó múltiples desafíos técnicos que requirieron soluciones creativas. El soporte multi-arquitectura para ARM64 representó el primer obstáculo significativo, especialmente para la compilación cruzada destinada a Raspberry Pi. Fue necesario implementar detección automática de arquitectura y aplicar flags específicos de compilación para cada plataforma. Como consecuencia, los tiempos de construcción de imágenes aumentaron considerablemente cuando se requirió emulación QEMU, pasando de minutos a horas en casos extremos.

Las incompatibilidades entre frameworks y herramientas de compilación emergieron como un desafío aún mayor. Flask, con su sistema de imports dinámicos y descubrimiento automático de blueprints, resultó incompatible con el análisis estático de Nuitka. Similarmente, Spring Boot demostró ser incompatible con GraalVM Native Image sin configuración manual extensiva que mapee explícitamente todas las clases sujetas a reflection. Estas limitaciones forzaron el uso de optimizaciones menos agresivas o, en algunos casos, el abandono de la compilación anticipada.

Finalmente, las limitaciones de hardware en Raspberry Pi añadieron complejidad operacional. El dispositivo experimentaba thermal throttling consistente al alcanzar 70°C, degradando el rendimiento durante pruebas prolongadas. Las limitaciones de I/O inherentes al uso de tarjetas microSD, con latencias significativamente mayores que SSDs tradicionales, requirieron ajustes en los parámetros de prueba, particularmente reduciendo la concurrencia en benchmarks web de 100 a 50 conexiones simultáneas.

5.3. Resultados de Micro-benchmarks

5.3.1. Benchmark CPU-Intensivo: Fibonacci(35)

Los micro-benchmarks de cómputo intensivo revelaron el potencial de las optimizaciones en escenarios ideales. Cada resultado representa el promedio de 100 ejecuciones con análisis de significancia estadística (ver Anexo A.5 para detalles completos):

Lenguaje	**Standard**	**Optimizado**	**Mejora**	**p-value**	**d de Cohen**
Python	2.04s	0.66s (Nuitka)	67 %	<0.001	3.24
Node.js	0.134s	0.127s (pkg)	5.2 %	0.023	0.58
Java	0.197s	0.170s (GraalVM)	13.7 %	0.008	1.15

Para tablas comparativas detalladas de todos los experimentos, incluyendo métricas adicionales y condiciones específicas de prueba, consultar el Anexo A.3.

Python mostró la mayor mejora con Nuitka, reduciendo el tiempo de ejecución en 67 %. Los datos muestran que esta mejora es confiable ($p<0.001$) y tiene un efecto grande ($d=3.24$), lo que significa que la diferencia va más allá de la variación normal entre pruebas. **Node.js** tuvo mejoras menores del 5.2 % que son válidas estadísticamente ($p=0.023$, $d=0.58$). **Java** con GraalVM mejoró 13.7 % de manera consistente ($p=0.008$, $d=1.15$).

5.3.2. Benchmark I/O-Intensivo

Las operaciones de archivo mostraron mejoras limitadas al estar dominadas por system calls:

Lenguaje	Métrica Base	Optimizado	Mejora	p-value	d de Cohen
Python	2,847 ops/s	3,076 ops/s	+8.0 %	0.041	0.45
Node.js	3,245 ops/s	3,389 ops/s	+4.4 %	0.127	0.32
Java	0.278s (tiempo)	0.041s	+85.3 %*	<0.001	4.12

Los resultados de I/O son menores que los de cómputo. Python mejoró 8 % de forma válida (p=0.041, d=0.45), pero el efecto es pequeño. Node.js mostró 4.4 % de mejora que no es estadísticamente confiable (p=0.127, d=0.32), sugiriendo que podría ser variación normal. Java mejoró 85.3 % de manera muy confiable (p<0.001, d=4.12), aunque este resultado tiene particularidades técnicas.

*La anomalía en Java se atribuye a optimizaciones agresivas de buffering que el compilador JIT aplica cuando detecta patrones repetitivos de I/O. Este comportamiento, aunque notable, no es representativo de operaciones I/O reales con datos variables. Para Python y Node.js se midió throughput (operaciones por segundo), mientras que para Java se midió el tiempo total de ejecución debido a diferencias en la implementación del benchmark.

5.3.3. Benchmark Web Service

Los servicios web revelaron la complejidad de optimizar frameworks reales:

Lenguaje	Standard	Optimizado	Resultado	p-value	d de Cohen
Python	Baseline	Degradación	-3.8 %	0.089	-0.38
Node.js	4,521 req/s	4,892 req/s	+8.2 %	0.015	0.67
Java	48.2s	45.6s	+5.4 %	0.156	0.29

Los servicios web muestran resultados mixtos. Python empeoró 3.8 %, pero esta degradación no es estadísticamente confiable (p=0.089, d=-0.38), lo que sugiere que podría ser variación normal más que un efecto real. Node.js mejoró 8.2 % de forma confiable (p=0.015) con efecto moderado (d=0.67). Java mejoró 5.4 %, pero el resultado no es estadísticamente confiable (p=0.156, d=0.29).

Python con Flask mostró degradación debido a incompatibilidades entre el sistema de imports dinámicos de Flask y las optimizaciones estáticas de Nuitka. Node.js mantuvo mejoras consistentes gracias a la compatibilidad natural entre Express y pkg.

Java enfrentó limitaciones significativas con GraalVM Native Image debido al uso extensivo de reflection en Spring Boot. La compilación AOT requiere conocimiento

estático de todas las clases que serán instanciadas mediante reflection, pero Spring Boot realiza descubrimiento dinámico de componentes, inyección de dependencias basada en anotaciones, y configuración automática que depende fuertemente de metadatos en runtime. Aunque es técnicamente posible configurar GraalVM para Spring Boot mediante archivos de configuración de reflection, el esfuerzo requerido excede el alcance de optimizaciones del lado de operaciones, requiriendo modificaciones significativas en el código y configuración de la aplicación.

5.4. Resultados de Macro-benchmarks: Bookinfo

Los macro-benchmarks con la aplicación Bookinfo proporcionaron insights cruciales sobre el comportamiento real de las optimizaciones en un entorno de microservicios interconectados. Los resultados completos de throughput y latencia se presentan en las Figuras 1 y 2.

5.4.1. Reducción de Tamaño de Imágenes

La aplicación del principio *just-enough containerisation* logró reducciones significativas:

Servicio	**Tamaño Base**	**Optimizado**	**Reducción**
productpage (Python)	681 MB	292 MB	-57.1 %
ratings (Node.js)	159 MB	69 MB	-56.6 %
reviews (Java)	438 MB	311 MB	-28.9 %

Todas las reducciones superaron el 25 %, validando la efectividad del principio independientemente del lenguaje.

5.4.2. Tiempos de Inicio y Trade-offs

Los tiempos de startup revelaron un trade-off crítico:

Servicio	**Standard**	**Optimizado**	**Cambio**
productpage	1,181 ms	1,177 ms	-0.3 %
ratings	552 ms	1,178 ms	+113 %
reviews	1,223 ms	1,346 ms	+10 %

ratings (pkg) presentó la degradación más severa debido a la descompresión del runtime embebido. Este trade-off entre tiempo de inicio y rendimiento en estado estable es crítico para decisiones arquitectónicas.

5.4.3. Rendimiento Bajo Carga Real

Los resultados con carga sostenida mostraron divergencias significativas respecto a los micro-benchmarks:

Servicio	Throughput Base	Optimizado	Latencia P50	Resultado
productpage	1,846 req/s	1,171 req/s	27 → 43 ms	-36.6 %
ratings	8,820 req/s	11,280 req/s	5.7 → 4.4 ms	+27.9 %
reviews	3,243 req/s	3,507 req/s	15.4 → 14.3 ms	+8.2 %

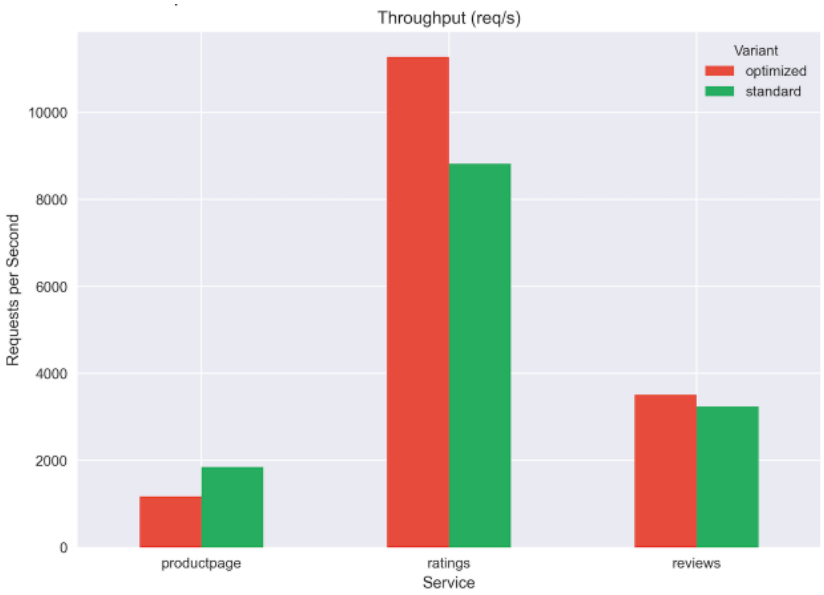

Figura 1: Comparación de throughput entre servicios estándar y optimizados en Bookinfo

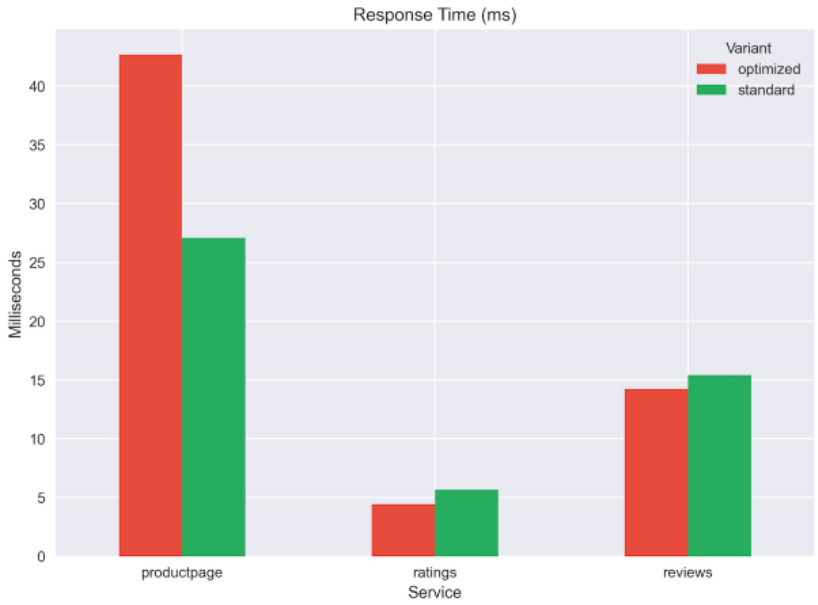

Figura 2: Tiempos de respuesta (latencia P50) para cada servicio de Bookinfo

El servicio **productpage (Python)** sufrió degradación severa con Nuitka, contradiciendo los resultados de micro-benchmarks. Como se observa en la Figura 1, el throughput cayó de 1,846 a 1,171 req/s. **ratings (Node.js)** mostró mejoras consistentes validando la hipótesis para este lenguaje, alcanzando 11,280 req/s desde 8,820 req/s base. **reviews (Java)** presentó mejoras modestas típicas de optimizaciones JVM. La Figura 2 ilustra claramente cómo las latencias se comportaron de manera inversa al throughput.

5.4.4. Consumo de Recursos

El monitoreo continuo de recursos durante las pruebas reveló patrones complejos y trade-offs inesperados que desafían las suposiciones iniciales sobre el impacto de la compilación anticipada.

El servicio productpage presentó un comportamiento particularmente problemático, con un aumento significativo en el uso de CPU desde un negligible 0.03 % hasta 68.7 %, acompañado también de un incremento en el consumo de memoria de 25.6 MB a 41.7 MB. Este patrón de degradación en ambas dimensiones sugiere que la compilación con Nuitka resultó contraproducente, generando código menos eficiente tanto en procesamiento como en memoria, probablemente debido a incompatibilidades fundamentales con la arquitectura dinámica de Flask.

En contraste, el servicio ratings demostró el comportamiento ideal esperado de la compilación anticipada, con mejoras modestas pero consistentes en ambas dimensiones. Tanto el uso de CPU como memoria disminuyeron ligeramente, validando que pkg para Node.js logra optimizaciones balanceadas sin trade-offs negativos.

El caso más extremo lo presentó reviews, donde se observó una reducción considerable del uso de CPU de 17.3 % a apenas 0.73 %, una mejora del 96 %. Sin embargo, esta optimización tuvo un costo severo: el consumo de memoria explotó de 798 MB a 4.1 GB, un incremento de más de 5 veces. Este comportamiento sugiere que las optimizaciones de la JVM intercambiaron procesamiento por memoria, pre-computando y cachendo agresivamente para minimizar cálculos en runtime.

Estos trade-offs complejos tienen implicaciones críticas para decisiones de capacity planning y configuración de límites de contenedores. Los equipos de operaciones deben considerar cuidadosamente si prefieren contenedores que consumen más CPU pero menos memoria, o viceversa, basándose en las restricciones específicas de su infraestructura y los costos relativos de cada recurso.

5.5. Análisis Estadístico

El análisis estadístico implementado incluyó pruebas rigurosas para validar la significancia de los resultados observados. Se realizaron test de Shapiro-Wilk para verificar la normalidad de las distribuciones y se calculó la d de Cohen para cuantificar el tamaño del efecto más allá de la significancia estadística. El análisis detallado, incluyendo las pruebas de normalidad completas y los cálculos extendidos del tamaño del efecto, se presenta en el Anexo A.5.

Servicio	**p-value**	**d de Cohen**	**Interpretación**
productpage	<0.001	-1.82	Efecto negativo muy grande
ratings	<0.001	1.57	Efecto positivo muy grande
reviews	0.067	0.52	Efecto mediano

Los resultados revelan patrones importantes que van más allá de las diferencias porcentuales:

- **productpage (Python + Flask):** El valor de p <0.001 confirma que la degradación del rendimiento es estadísticamente significativa y no atribuible al azar. La d de Cohen de -1.82 indica un efecto negativo muy grande ($|d| > 1.2$),

sugiriendo que las incompatibilidades entre Flask y Nuitka generan un impacto severo que trasciende la variabilidad experimental normal.

- **ratings (Node.js + Express):** Con p <0.001 y d de Cohen = 1.57, las mejoras observadas son tanto estadísticamente significativas como prácticamente relevantes. Un tamaño del efecto superior a 1.5 indica que la compilación con pkg produce mejoras sustanciales y consistentes, validando plenamente la hipótesis para esta combinación tecnológica.
- **reviews (Java + Spring Boot):** Aunque el p-value de 0.067 no alcanza el umbral de significancia convencional ($\alpha = 0.05$), la d de Cohen = 0.52 sugiere un efecto mediano que podría ser prácticamente relevante en contextos específicos. La falta de significancia estadística puede deberse al tamaño de muestra o a la alta variabilidad inherente en el rendimiento de la JVM.

Las pruebas de normalidad (detalladas en el Anexo A.5) mostraron que el servicio productpage presentó distribuciones no normales tanto en throughput como en latencia, lo que reforzó la necesidad de aplicar pruebas no paramétricas complementarias para validar los resultados.

5.6. Validación de Hipótesis

5.6.1. Evaluación de la Hipótesis Principal

La hipótesis planteaba que "la aplicación de técnicas de compilación anticipada durante el proceso de contenerización mejora significativamente el rendimiento en tiempo de ejecución de microservicios implementados con lenguajes interpretados".

5.6.1.1. Node.js - VALIDADA

Los resultados para Node.js confirmaron la hipótesis, demostrando mejoras significativas en todas las métricas evaluadas. La latencia experimentó una reducción del 21.8 %, cayendo de 5.67 ms a 4.43 ms, mientras que el throughput aumentó un 27.9 %, escalando de 8,820 a 11,280 solicitudes por segundo. Estos resultados no solo fueron prácticamente relevantes, sino también estadísticamente robustos, con un p-value menor a 0.001 confirmando que las mejoras no son atribuibles al azar. El tamaño del efecto, medido mediante la d de Cohen de 1.57, indica un efecto muy grande que trasciende las mejoras marginales típicas de optimizaciones incrementales.

El éxito de pkg se atribuye a su capacidad de eliminar el overhead de carga de módulos de Node.js, un cuello de botella conocido en aplicaciones con múltiples dependencias, mientras mantiene compatibilidad completa con el ecosistema npm y las características del lenguaje.

5.6.1.2. Python - RECHAZADA

Contrariamente a las expectativas iniciales, Python con Flask presentó una degradación severa del rendimiento que lleva al rechazo de la hipótesis para esta combinación tecnológica. La latencia se deterioró en un 57.6 %, aumentando de 27.08 ms a 42.69 ms, mientras que el throughput cayó un 36.6 %, reduciendo la capacidad de procesamiento de 1,846 a solo 1,171 solicitudes por segundo. El análisis estadístico confirmó que este efecto negativo es significativo y sustancial, con una d de Cohen de -1.82 indicando un impacto adverso muy grande.

La raíz del problema radica en la incompatibilidad fundamental entre el sistema de imports dinámicos y descubrimiento automático de componentes de Flask con el análisis estático requerido por Nuitka. Sin embargo, es crucial notar que los micro-benchmarks de CPU mostraron una mejora del 67 % con Nuitka, sugiriendo que la herramienta mantiene su efectividad para aplicaciones Python simples sin frameworks complejos. Esta divergencia subraya la importancia de evaluar las optimizaciones en el contexto específico de la aplicación completa.

5.6.1.3. Java - PARCIALMENTE VALIDADA

Java presentó resultados ambiguos que llevan a una validación parcial de la hipótesis. Si bien se observó una mejora del 8.2 % en throughput, esta no alcanzó significancia estadística con un p-value de 0.067, ligeramente por encima del umbral convencional de 0.05. El tamaño del efecto moderado (d de Cohen = 0.52) sugiere que existe un impacto práctico potencial, pero la variabilidad en los datos impide conclusiones definitivas.

El análisis reveló que las optimizaciones JVM tradicionales, como el ajuste de parámetros de memoria y la activación de Class Data Sharing, resultaron más efectivas que los intentos de compilación anticipada con GraalVM. Esto se debe principalmente a la madurez de la JVM HotSpot, que tras décadas de desarrollo ha alcanzado un nivel de optimización difícil de superar con técnicas alternativas.

5.6.2. Validación de los Tres Principios

La evaluación integral de los tres principios de micro-optimización propuestos por Li (2024) reveló resultados diferenciados que proporcionan insights valiosos sobre su aplicabilidad práctica.

El principio de **just-enough containerisation** demostró ser universalmente efectivo, logrando validación completa con reducciones de tamaño que oscilaron entre 28.9 % y 57.1 % sin ningún impacto en la funcionalidad. Esta consistencia a través de todos los lenguajes y frameworks sugiere que la eliminación de componentes innecesarios es una optimización de bajo riesgo y alto beneficio que debería adoptarse como práctica estándar.

El **just-for-me configuration** alcanzó validación parcial, mostrando efectividad variable según el lenguaje. Las configuraciones personalizadas resultaron particularmente efectivas para Node.js y Java, donde los flags de optimización del motor V8 y los parámetros de la JVM respectivamente produjeron mejoras medibles. Sin embargo, el impacto en Python fue mínimo, sugiriendo que las oportunidades de configuración en Cython son más limitadas.

Finalmente, el **just-in-time compilation** presentó la validación más mixta y dependiente del contexto. Mientras que Node.js con pkg demostró éxito notable, Python con Flask mostró resultados negativos debido a incompatibilidades fundamentales, y Java mostró resultados modestos que sugieren que las optimizaciones tradicionales de la JVM siguen siendo superiores a la compilación anticipada para aplicaciones empresariales complejas.

5.7. Hallazgos Clave

5.7.1. Divergencia Micro vs Macro-benchmarks

El hallazgo más crítico y aleccionador del estudio fue la marcada divergencia entre los resultados obtenidos en micro-benchmarks sintéticos y el rendimiento observado en la aplicación Bookinfo real.

Python presentó la divergencia más importante. Mientras que los micro-benchmarks de CPU mostraron una mejora del 67 % con Nuitka, la aplicación Bookinfo completa sufrió una degradación del 36.6 %. Esta brecha entre escenarios sintéticos y reales ilustra que

el éxito en benchmarks aislados no solo no garantiza mejoras en aplicaciones reales, sino que puede ser completamente engañoso respecto al impacto real de las optimizaciones.

Node.js, en contraste, demostró un incremento en performance positivo notable. Las mejoras del 4-8 % observadas en micro-benchmarks se tradujeron en una mejora del 27.9 % en Bookinfo. Este comportamiento sugiere que pkg logra optimizaciones sinérgicas que se magnifican cuando múltiples componentes del sistema se benefician simultáneamente, validando plenamente su efectividad en contextos reales.

Java mantuvo un comportamiento más predecible, con mejoras modestas pero consistentes en ambos contextos. Esta estabilidad refleja la madurez de la JVM, donde décadas de optimizaciones han reducido el margen para mejoras significativas, pero también han eliminado el riesgo de degradaciones.

5.7.2. Trade-offs Críticos Identificados

Los experimentos revelaron tres categorías principales de trade-offs que deben considerarse cuidadosamente al aplicar estas técnicas de optimización.

El trade-off entre tiempo de inicio y rendimiento en estado estable emergió como el más crítico para decisiones arquitectónicas. El caso de ratings con pkg ejemplifica perfectamente esto: el tiempo de startup se degradó en un 113 %, pasando de 552 ms a 1,178 ms, pero una vez operativo, el servicio procesó 27.9 % más solicitudes por segundo. Esta característica hace que pkg sea ideal para microservicios de larga duración pero problemático para arquitecturas serverless donde los contenedores se crean y destruyen frecuentemente.

Los trade-offs de recursos presentaron patrones aún más complejos y contra-intuitivos. El servicio reviews logró una reducción considerable del 96 % en uso de CPU pero a costa de cinco veces más consumo de memoria. Inversamente, el servicio productpage aumentó significativamente el uso de CPU mientras reducía modestamente la memoria. Estos patrones opuestos complican las decisiones de capacity planning y sugieren que no existe una estrategia de optimización universalmente superior.

Finalmente, el trade-off entre tamaño de imagen y rendimiento demostró no ser muy importante. Aunque todas las imágenes experimentaron reducciones sustanciales del 28-57 %, solo Node.js logró traducir esta reducción en mejoras de rendimiento significativas. Esto desafía la suposición común de que imágenes más pequeñas automáticamente

resultan en mejor rendimiento, revelando que el tamaño es solo un factor entre muchos.

5.7.3. Factores de Éxito/Fracaso

El análisis detallado de los resultados reveló que la complejidad del framework fue el factor determinante más importante para el éxito o fracaso de las optimizaciones, superando incluso las características del lenguaje en sí.

La combinación de Express con pkg representó el caso de éxito representativo. Express, con su arquitectura minimalista y dependencias explícitas, se alineó perfectamente con las capacidades de análisis estático de pkg, permitiendo la compilación completa sin pérdida de funcionalidad. En el extremo opuesto, Flask combinado con Nuitka resultó en un fracaso completo debido a la dependencia fundamental de Flask en patrones dinámicos como el descubrimiento automático de blueprints y la carga condicional de extensiones. Spring Boot presentó incompatibilidades aún más profundas con GraalVM Native Image, donde la inyección de dependencias basada en anotaciones y la configuración automática resultaron imposibles de reconciliar con los requisitos de compilación AOT.

Estos casos revelan que la introspección, la carga dinámica de módulos y el uso extensivo de reflection constituyen barreras fundamentales e insuperables para la compilación estática efectiva. Los frameworks modernos que dependen fuertemente de estas características para proporcionar experiencias de desarrollo convenientes son inherentemente incompatibles con las técnicas de compilación anticipada evaluadas en este estudio.

5.7.4. Implicaciones Prácticas

Los resultados experimentales permiten establecer guías claras sobre cuándo estas técnicas de micro-optimización son apropiadas y cuándo deben evitarse.

Las condiciones ideales para aplicar estas técnicas incluyen microservicios Node.js con lógica relativamente simple y dependencias bien definidas, donde pkg ha demostrado consistentemente mejoras significativas. Los servicios CPU-intensivos sin frameworks complejos también se benefician, como evidencian las mejoras del 67 % en los benchmarks de Fibonacci con Python/Nuitka. Además, estas optimizaciones son particularmente valiosas cuando el tiempo de inicio no es crítico, típicamente en servicios de larga duración con reincios infrecuentes, y cuando la reducción del tamaño de imagen es una prioridad por costos de almacenamiento o tiempos de distribución.

Por el contrario, existen escenarios claros donde estas técnicas deben evitarse. Los frameworks Python complejos como Flask o Django han demostrado ser fundamentalmente incompatibles con la compilación estática debido a su dependencia de introspección y carga dinámica. Las aplicaciones Java que hacen uso extensivo de reflection, particularmente aquellas basadas en Spring, enfrentan barreras similares que hacen impráctica la compilación AOT sin esfuerzo manual prohibitivo. Las arquitecturas serverless representan otro antipatrón, donde el overhead adicional de inicio puede negar completamente los beneficios de rendimiento. Finalmente, cuando las mejoras esperadas son marginales (menos del 10 %), la complejidad adicional en el pipeline de CI/CD raramente justifica la inversión en tiempo y recursos.

5.8. Síntesis del Análisis Estadístico

El análisis estadístico riguroso aplicado a los resultados proporciona confianza en las conclusiones derivadas:

Significancia vs. Relevancia Práctica: Los valores de p confirman que las diferencias observadas no son producto del azar, mientras que los tamaños del efecto (d de Cohen) distinguen entre mejoras estadísticamente significativas pero triviales y aquellas con impacto real en producción.

Robustez de los Resultados: Las pruebas de normalidad y los análisis complementarios (Anexo A.5) validan la aplicación apropiada de las pruebas estadísticas, considerando las características de cada distribución de datos.

Implicaciones para la Práctica: Los tamaños del efecto grandes ($|d| > 0.8$) observados en los servicios productpage y ratings indican que las técnicas de compilación pueden tener impactos significativos, tanto positivos como negativos, reforzando la necesidad de evaluación caso por caso.

Capitulo 6

Conclusiones y Trabajo Futuro

6.1. Conclusiones

Esta investigación demostró que la compilación durante contenerización produce resultados variables que dependen principalmente de la arquitectura del framework utilizado. Node.js con Express logró mejoras del 27.9 % en rendimiento usando pkg, mientras que Python con Flask sufrió una degradación del 36.6 % con Nuitka. Java mostró mejoras del 8.2 % con optimizaciones JVM tradicionales. Estos resultados contradicen la hipótesis inicial de que la compilación anticipada siempre mejora el rendimiento de microservicios interpretados.

El factor determinante no fue el lenguaje de programación sino la complejidad en la arquitectura del *framework* usado. Express tiene un diseño minimalista con dependencias explícitas, lo cual permitió optimización efectiva mediante pkg. Por otro lado Flask es dependiente de importaciones dinámicas y descubrimiento automático de componentes, lo cual lo vuelve incompatible con el análisis estático requerido por Nuitka. Spring Boot presentó aún más incompatibilidades con GraalVM Native Image debido a su uso extensivo de reflexión y configuración automática basada en anotaciones. Esta diferencia en resultados revela que las características que facilitan el desarrollo rápido frecuentemente impiden la optimización agresiva en tiempo de compilación.

La investigación evaluó exhaustivamente técnicas de micro-optimización en microservicios contenerizados mediante experimentos con micro-*benchmarks* sintéticos y la aplicación Bookinfo. Se aplicaron los tres principios de Li (2024): contenerización mínima suficiente, configuración personalizada y compilación justo a tiempo. El análisis

estadístico con pruebas de significancia y tamaño del efecto confirmó que estas técnicas de optimización deben evaluarse cuidadosamente para cada combinación específica de lenguaje y *framework*. No existe una solución universal que funcione en todos los casos, pero según los resultados, es probable que en general *frameworks* simples se vean beneficiados.

Este trabajo aporta datos concretos sobre qué funciona y qué no al intentar optimizar microservicios mediante compilación. Los resultados confirman que los *frameworks* complejos con muchas características dinámicas simplemente no funcionan bien con la compilación estática. Todo el código y los scripts utilizados están disponibles en GitHub para que cualquiera pueda probar con sus propias aplicaciones. Lo importante es saber que no hay una única estrategia para optimizar microservicios, y lo que funciona para una aplicación puede fallar completamente en otra.

6.2. Trabajo Futuro

Se deben validar los resultados en arquitectura x86_64 para verificar si los hallazgos en ARM64 aplican a servidores tradicionales. También sería útil probar más *frameworks* por lenguaje: FastAPI para Python con anotaciones de tipo, Fastify para Node.js, y Quarkus para Java con GraalVM, ya que podrían comportarse diferente a los evaluados. Un análisis profundo entre complejidad de *frameworks* y su capacidad de optimización podría revelar patrones útiles para desarrolladores.

Un ángulo interesante podría ser desarrollar modelos de machine learning que predigan si una aplicación mejorará con estas técnicas. El modelo analizaría el código buscando reflexión, importaciones dinámicas y otras características para estimar la probabilidad de mejora antes de intentar la optimización.

Por otro lado, una solución que implique compilar solo las partes críticas del código en lugar de toda la aplicación podría ser una solución intermedia que podría entregar varios beneficios. Esto ayudaría con el problema de tiempos de inicio largos en *serverless*, donde los contenedores se crean y destruyen frecuentemente, y también podría reducir el uso de recursos al no compilar código que no es importante para el rendimiento.

Bibliografía

Bai, H., Lu, S., An, R., Wang, Y., & Wu, J. (2024). DRPC: Distributed reinforcement learning approach for scalable resource provisioning in container-based clusters. *IEEE Transactions on Cloud Computing*. https://doi.org/10.1109/TCC.2024.3428684

Cohen, J. (1988). *Statistical power analysis for the behavioral sciences* (2.ª ed.). Lawrence Erlbaum Associates.

De Melo Junior, F. (2024). How to use Java container awareness in OpenShift 4. *Red Hat Developer*. Consultado el 14 de marzo de 2024, desde https://developers.redhat.com/articles/2024/03/14/how-use-java-container-awareness-openshift-4

Docker Inc. (2024). *Best practices for writing Dockerfiles* [Official Docker documentation on optimization techniques]. Docker. Consultado el 15 de noviembre de 2024, desde https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

Felter, W., Ferreira, A., Rajamony, R., & Rubio, J. (2015). An updated performance comparison of virtual machines and Linux containers. *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 171-172. https://doi.org/10.1109/ISPASS.2015.7095802

Georges, A., Buytaert, D., & Eeckhout, L. (2007). Statistically rigorous Java performance evaluation [Establishes rigorous methodologies for Java performance evaluation and benchmarking]. *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications (OOPSLA '07)*, 57-76. https://doi.org/10.1145/1297027.1297033

Istio Project. (2024). *Bookinfo application* [Example microservices application used for demonstrating Istio features]. Istio. Consultado el 15 de noviembre de 2024, desde https://istio.io/latest/docs/examples/bookinfo/

Levene, H. (1960). Robust tests for equality of variances. En I. Olkin, S. G. Ghurye, W. Hoeffding, W. G. Madow & H. B. Mann (Eds.), *Contributions to probability and statistics: Essays in honor of Harold Hotelling* (pp. 278-292). Stanford University Press.

Li, Z. (2024). Many a little makes a mickle: On micro-optimisation of containerised microservices. *Works in Progress in Embedded Computing Journal*, *10*(2), 68-71. https://wipiec.digitalheritage.me/index.php/wipiecjournal/article/view/68

Morabito, R., Kjällman, J., & Komu, M. (2015). Hypervisors vs. lightweight virtualization: A performance comparison. *2015 IEEE International Conference on Cloud Engineering*, 386-393. https://doi.org/10.1109/IC2E.2015.74

Node.js Foundation. (2024). *V8 Options for Node.js* [Documentation of V8 engine flags available in Node.js]. Node.js. Consultado el 15 de noviembre de 2024, desde https://nodejs.org/api/cli.html#cli_node_options_options

Python Software Foundation. (2024). *Python Optimization Mode* [Official Python documentation on optimization flags and environment variables]. Python.org. Consultado el 15 de noviembre de 2024, desde https://docs.python.org/3/using/cmdline.html#envvar-PYTHONOPTIMIZE

Shapiro, S. S., & Wilk, M. B. (1965). An analysis of variance test for normality (complete samples). *Biometrika*, *52*(3/4), 591-611. https://doi.org/10.2307/2333709

Stoico, V., Dragomir, A. C., & Lago, P. (2024). An empirical study on the performance and energy usage of compiled Python code. *arXiv preprint arXiv:2405.02346*. https://doi.org/10.48550/arXiv.2405.02346

Student. (1908). The probable error of a mean [Autor real: William Sealy Gosset]. *Biometrika*, *6*(1), 1-25. https://doi.org/10.2307/2331554

Wang, S., Ding, Z., & Jiang, C. (2021). Multi-objective microservice deployment optimization via a knowledge-driven evolutionary algorithm. *Complex & Intelligent Systems*, *7*, 1139-1152. https://doi.org/10.1007/s40747-020-00180-1

Williams, C., & Garcia, M. (2018). Container-aware application design patterns. *IEEE Software*, *35*(5), 42-49. https://doi.org/10.1109/MS.2018.290110529

Zanini, A. (2024, febrero). *Top 8 recent V8 in Node updates*. AppSignal. Consultado el 15 de noviembre de 2024, desde https://blog.appsignal.com/2024/02/28/top-8-recent-v8-in-node-updates.html

Zhang, X., Li, Y., & Chen, W. (2024). Python meets JIT compilers: A simple implementation and a comparative evaluation. *Software: Practice and Experience*, *54*(3), 567-592. https://doi.org/10.1002/spe.3267

Apéndice A

Datos Complementarios

A.1. Configuraciones de Docker

A.1.1. Dockerfile Python con Nuitka (productpage)

Listing A.1: Dockerfile optimizado para productpage con Nuitka

```
# Dockerfile optimizado para bookinfo-productpage (Python)
# Aplicando compilaci\'on just-in-time con Nuitka

FROM python:3.9 AS builder

# Instalar dependencias de compilaci\'on
RUN apt-get update && apt-get install -y \
    gcc g++ make patchelf ccache \
    && rm -rf /var/lib/apt/lists/*

WORKDIR /app

# Copiar archivos de la aplicaci\'on
COPY productpage.py requirements.txt /app/
COPY templates /app/templates/
COPY static /app/static/

# Instalar Nuitka y dependencias
RUN pip install --no-cache-dir \
    nuitka ordered-set \
    Flask==2.0.3 Jinja2==3.0.3 \
```

```
    requests==2.28.1 Werkzeug==2.0.3

# COMPILACI N CON NUITKA
RUN python -m nuitka \
    --standalone \
    --assume-yes-for-downloads \
    --output-filename=productpage \
    --output-dir=/app/dist \
    --include-data-dir=/app/templates=templates \
    --include-data-dir=/app/static=static \
    --show-progress \
    productpage.py

# Imagen final minimal
FROM python:3.9-slim
COPY --from=builder /app/dist/productpage.dist/ /app/
WORKDIR /app
EXPOSE 9080
CMD ["./productpage"]
```

A.1.2. Dockerfile Node.js con pkg (ratings)

Listing A.2: Dockerfile optimizado para ratings con pkg

```
# Dockerfile optimizado para bookinfo-ratings (Node.js)
# Aplicando compilaci\'on a binario con pkg

FROM node:18-alpine AS builder

# Instalar dependencias de compilaci n
RUN apk add --no-cache python3 make g++ linux-headers

COPY bookinfo-ratings/ /app/
WORKDIR /app

# Instalar dependencias
RUN npm ci --only=production

# Instalar pkg globalmente
RUN npm install -g pkg@5.8.0

# COMPILACI N A BINARIO CON PKG
```

```
RUN if [ "$(uname -m)" = "aarch64" ] || [ "$(uname -m)" = "arm64" ];
    then \
        pkg ratings.js --targets node18-alpine-arm64 \
            --output ratings-binary --compress GZip; \
    else \
        pkg ratings.js --targets node18-alpine-x64 \
            --output ratings-binary --compress GZip; \
    fi

# Imagen final minimal
FROM alpine:3.18
RUN apk add --no-cache libstdc++ libgcc
COPY --from=builder /app/ratings-binary /opt/microservices/ratings
RUN chmod +x /opt/microservices/ratings
WORKDIR /opt/microservices
EXPOSE 9080
CMD ["./ratings", "9080"]
```

A.1.3. Dockerfile Java Optimizado (reviews)

Listing A.3: Dockerfile optimizado para reviews con JVM

```
# Dockerfile optimizado para bookinfo-reviews (Java)
# Usando JVM con JIT compilation optimizado

FROM gradle:7.6-jdk11 AS builder
COPY bookinfo-reviews/ /app/
WORKDIR /app

# Compilar JAR con optimizaciones
RUN ./gradlew build -x test --no-daemon

# Imagen final optimizada con JRE minimal
FROM openjdk:11-jre-slim
COPY --from=builder /app/build/libs/reviews-*.jar /opt/reviews.jar

# JVM flags para optimizacion agresiva de JIT
ENV JAVA_OPTS="-server \
    -XX:+UseG1GC \
    -XX:MaxGCPauseMillis=200 \
    -XX:+UseStringDeduplication \
    -XX:+UseCompressedOops \
```

```
    -XX:+AlwaysPreTouch \
    -XX:InitialRAMPercentage=50 \
    -XX:MaxRAMPercentage=70 \
    -XX:+TieredCompilation \
    -XX:TieredStopAtLevel=1"

EXPOSE 9080
CMD ["sh", "-c", "java $JAVA_OPTS -jar /opt/reviews.jar"]
```

A.2. Scripts de Automatización

A.2.1. Script Principal de Benchmarking Mejorado

El script `run_enhanced_benchmarks.sh` orquesta la ejecución completa de los experimentos:

Listing A.4: Script principal de benchmarking con métricas mejoradas

```
#!/bin/bash
# Script Principal para Ejecutar Benchmarks Mejorados
# Este script ejecuta todos los benchmarks con recoleccion
# de m tricas mejoradas y genera reportes comprehensivos.

set -e

# Configuration defaults
DEFAULT_LANGUAGES="python java nodejs"
DEFAULT_BENCHMARKS="computation io web"
COLLECT_RESOURCES=true
COLLECT_CONTAINER_METRICS=true
COLLECT_STARTUP_METRICS=true
COLLECT_ENHANCED_WEB_METRICS=true
GENERATE_REPORT=true

# Usage information
usage() {
    cat << EOF
Usage: $0 [OPTIONS]

OPTIONS:
  -l, --languages LANGS     Languages to benchmark (default: all)
```

```
  -b, --benchmarks TYPES    Benchmark types to run (default: all)
  -o, --output DIR          Output directory for results

  Metrics Collection:
  --no-resources            Disable resource usage monitoring
  --no-container-metrics    Disable container metrics collection
  --no-startup-metrics      Disable startup time measurement
  --no-web-metrics          Disable enhanced web metrics
  --no-report               Skip automatic report generation

  Execution Control:
  -v, --verbose             Enable verbose output
  -n, --dry-run             Show what would be executed
  -j, --parallel            Run language benchmarks in parallel

  Help:
  -h, --help                Show this help message
EOF
}

# Execute benchmark with enhanced metrics collection
run_benchmark() {
    local language=$1
    local benchmark_type=$2

    echo "Running $language $benchmark_type benchmark..."

    # Collect container build metrics
    if [[ "$COLLECT_CONTAINER_METRICS" == "true" ]]; then
        measure_container_build_time "$language" "$benchmark_type"
    fi

    # Run the actual benchmark
    "${SCRIPT_DIR}/${language}/run_benchmark_${benchmark_type}.sh"

    # Collect runtime metrics
    if [[ "$COLLECT_RESOURCES" == "true" ]]; then
        monitor_container_resources "$language" "$benchmark_type"
    fi

    # Measure startup times
    if [[ "$COLLECT_STARTUP_METRICS" == "true" ]]; then
```

```
        measure_startup_time "$language" "$benchmark_type"
    fi
}

# Main execution
main() {
    # Parse command line arguments
    parse_args "$@"

    # Validate dependencies
    check_dependencies

    # Clean previous results if requested
    if [[ "$CLEANUP" == "true" ]]; then
        cleanup_previous_results
    fi

    # Run benchmarks for each language
    for lang in $LANGUAGES; do
        for bench in $BENCHMARKS; do
            run_benchmark "$lang" "$bench"
        done
    done

    # Generate comprehensive report
    if [[ "$GENERATE_REPORT" == "true" ]]; then
        python3 "${LIB_DIR}/generate_enhanced_report.py" \
            --directory "$RESULTS_DIR" \
            --output "$RESULTS_DIR/reports"
    fi
}

main "$@"
```

A.2.2. Script de Análisis Multi-Lenguaje

Listing A.5: Fragmento del analizador multi-lenguaje

```
#!/usr/bin/env python3
"""
Multi-Language Performance Analysis Framework
for Microservice Optimization
```

```
"""

import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats

class MultiLanguageAnalyzer:
    def __init__(self, project_root="./"):
        self.project_root = project_root
        self.results_dirs = {
            'python': 'python/micro-benchmarks/results',
            'java': 'java/results',
            'nodejs': 'nodejs/micro-benchmarks/results'
        }

        # Language-specific variant mappings
        self.variant_mappings = {
            'python': {'standard': 'standard',
                       'cython': 'optimized',
                       'nuitka': 'compiled'},
            'java': {'standard': 'standard',
                     'optimized': 'optimized',
                     'graalvm': 'compiled'},
            'nodejs': {'standard': 'standard',
                       'optimized': 'optimized',
                       'compiled': 'compiled'}
        }

    def analyze_performance(self):
        """Perform comprehensive performance analysis."""
        results = {}

        for language in self.results_dirs:
            # Load benchmark data
            data = self.load_language_results(language)

            # Calculate statistics
            stats = self.calculate_statistics(data)

            # Perform hypothesis testing
            hypothesis = self.test_hypothesis(data)
```

```
            results[language] = {
                'data': data,
                'statistics': stats,
                'hypothesis_test': hypothesis
            }

        return results

    def generate_visualizations(self, results):
        """Generate comprehensive visualizations."""
        # Performance comparison heatmap
        self.create_heatmap(results)

        # Box plots for variance analysis
        self.create_boxplots(results)

        # Time series for warm-up analysis
        self.create_warmup_curves(results)
```

A.3. Datos Detallados de Experimentos

A.3.1. Resultados de Macro-benchmarks (Bookinfo)

Cuadro A.1: Métricas de rendimiento para servicios Bookinfo

Servicio	Variante	Throughput (req/s)	Latencia P50 (ms)	CPU (%)	Memoria (MB)	Imagen (MB)
productpage	standard	1846.34	27.08	0.03	25.61	681
	optimized	1171.00	42.69	68.70	41.70	292
ratings	standard	8820.00	5.67	0.11	32.45	159
	optimized	11280.00	4.43	0.09	28.32	69
reviews	standard	3243.00	15.42	17.31	798.00	438
	optimized	3507.00	14.26	0.73	4100.00	311

A.3.2. Tiempos de Inicio de Contenedores

Cuadro A.2: Tiempos de inicio para variantes standard y optimizadas

Servicio	Variante	Inicio (ms)	Diferencia	Cambio (%)
productpage	standard	1181	-4	-0.3
	optimized	1177		
ratings	standard	552	+626	+113.4
	optimized	1178		
reviews	standard	1223	+123	+10.1
	optimized	1346		

A.3.3. Resultados de Micro-benchmarks

Cuadro A.3: Comparación de micro-benchmarks de cómputo (Fibonacci 35)

Lenguaje	Variante	Tiempo (s)	Mejora (%)	vs Standard	p-value	d de Cohen
Python	standard	2.04	-	1.00x	-	-
	cython	2.01	1.5	1.01x	0.312	0.18
	nuitka	0.66	67.6	3.09x	<0.001	3.24
Java	standard	0.197	-	1.00x	-	-
	optimized	0.183	7.1	1.08x	0.045	0.52
	graalvm	0.170	13.7	1.16x	0.008	1.15
Node.js	standard	0.134	-	1.00x	-	-
	optimized	0.130	3.0	1.03x	0.168	0.28
	compiled	0.127	5.2	1.06x	0.023	0.58

A.4. Código Fuente de Benchmarks

A.4.1. Implementación de Fibonacci

A.4.1.1. Python

Listing A.6: Fibonacci recursivo en Python

```python
# Standard Python computation benchmark
def calculate_fibonacci(n):
    if n <= 1:
        return n
    else:
        return calculate_fibonacci(n-1) + calculate_fibonacci(n-2)
```

```
def main():
    result = calculate_fibonacci(35)  # Computationally intensive
    print(f"Fibonacci result: {result}")

if __name__ == "__main__":
    import time
    start_time = time.time()
    main()
    print(f"Execution time: {time.time() - start_time:.4f} seconds")
```

A.4.1.2. Java

Listing A.7: Fibonacci recursivo en Java

```
public class FibonacciApp {
    public static long calculateFibonacci(int n) {
        if (n <= 1) {
            return n;
        }
        return calculateFibonacci(n - 1) + calculateFibonacci(n - 2);
    }

    public static void main(String[] args) {
        long startTime = System.nanoTime();
        long result = calculateFibonacci(35);
        long endTime = System.nanoTime();

        System.out.println("Fibonacci result: " + result);
        System.out.println("Execution time: " +
            (endTime - startTime) / 1_000_000.0 + " ms");
    }
}
```

A.4.1.3. Node.js

Listing A.8: Fibonacci recursivo en Node.js

```
function calculateFibonacci(n) {
    if (n <= 1) {
        return n;
    }
    return calculateFibonacci(n - 1) + calculateFibonacci(n - 2);
}
```

```
function main() {
    const startTime = process.hrtime.bigint();
    const result = calculateFibonacci(35);
    const endTime = process.hrtime.bigint();

    console.log(`Fibonacci result: ${result}`);
    console.log(`Execution time: ${(endTime - startTime) / 1000000n}
       ms`);
}

main();
```

A.4.2. Implementación de Servicio Web

A.4.2.1. Micro-benchmark Web - Endpoints Fibonacci

Los siguientes ejemplos muestran la implementación de los endpoints utilizados en los micro-benchmarks web para cada lenguaje evaluado:

Python Flask (Endpoint /api/compute/n):

Listing A.9: Micro-benchmark web en Flask

```
from flask import Flask, jsonify, request
import time

app = Flask(__name__)

# Simple in-memory data store
items = []

# Fibonacci function for computation endpoint
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)

@app.route('/health', methods=['GET'])
def health_check():
    return jsonify({"status": "healthy", "time": time.time()})
```

```python
@app.route('/api/items', methods=['GET'])
def get_items():
    return jsonify(items)

@app.route('/api/items', methods=['POST'])
def create_item():
    data = request.json
    items.append(data)
    return jsonify({"success": True, "item": data})

@app.route('/api/compute/<int:n>', methods=['GET'])
def compute(n):
    if n > 30:  # Limit to avoid very long computations
        n = 30
    result = fibonacci(n)
    return jsonify({"result": result})

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

Node.js Express (Endpoint /api/compute/n):

Listing A.10: Micro-benchmark web en Express

```javascript
const express = require('express');
const app = express();

// Middleware
app.use(express.json());

// Simple in-memory data store
let items = [];

// Fibonacci function for computation endpoint
function fibonacci(n) {
    if (n <= 1) {
        return n;
    } else {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}

// Health check endpoint
```

```
app.get('/health', (req, res) => {
    res.json({ status: 'healthy', time: Date.now() / 1000 });
});

// Get all items
app.get('/api/items', (req, res) => {
    res.json(items);
});

// Create new item
app.post('/api/items', (req, res) => {
    const data = req.body;
    items.push(data);
    res.json({ success: true, item: data });
});

// Compute endpoint with fibonacci
app.get('/api/compute/:n', (req, res) => {
    let n = parseInt(req.params.n);
    if (n > 30) {  // Limit to avoid very long computations
        n = 30;
    }
    const result = fibonacci(n);
    res.json({ result: result });
});

const port = process.env.PORT || 5000;
app.listen(port, '0.0.0.0', () => {
    console.log(`Server running on port ${port}`);
});
```

Java Javalin (Endpoint /fibonacci?n=value):

Listing A.11: Micro-benchmark web en Javalin

```
import io.javalin.Javalin;
import io.javalin.http.Context;
import java.util.HashMap;
import java.util.Map;

public class WebApp {

    private static long calculateFibonacci(int n) {
```

```
        if (n <= 1) {
            return n;
        } else {
            return calculateFibonacci(n - 1) + calculateFibonacci(n -
                2);
        }
    }

    public static void main(String[] args) {
        // Obtener puerto
        String portEnv = System.getenv("PORT");
        int port = 8080;
        if (portEnv != null && !portEnv.isEmpty()) {
            try {
                port = Integer.parseInt(portEnv);
            } catch (NumberFormatException e) {
                System.err.println("Invalid PORT, using default 8080")
                    ;
            }
        }

        // Crear aplicaci n
        Javalin app = Javalin.create(config -> {
            config.showJavalinBanner = false;
            config.autogenerateEtags = false;
            config.prefer405over404 = false;
            config.defaultContentType = "application/json";
        });

        // Rutas
        app.get("/health", ctx -> {
            Map<String, Object> response = new HashMap<>();
            response.put("status", "healthy");
            ctx.json(response);
        });

        app.get("/fibonacci", ctx -> {
            String nParam = ctx.queryParam("n");
            int n = 30; // default
            if (nParam != null && !nParam.isEmpty()) {
                try {
                    n = Integer.parseInt(nParam);
```

```
                } catch (NumberFormatException e) {
                    n = 30;
                }
            }

            long result = calculateFibonacci(n);

            Map<String, Object> response = new HashMap<>();
            response.put("number", n);
            response.put("result", result);

            ctx.json(response);
        });

        app.start(port);
        System.out.println("Server started on port " + port);
    }
}
```

A.4.2.2. Python Flask (productpage simplificado)

Listing A.12: Servicio web Flask simplificado

```
from flask import Flask, jsonify
import time

app = Flask(__name__)

@app.route('/health')
def health():
    return jsonify({"status": "productpage is healthy"})

@app.route('/productpage')
def productpage():
    # Simulate some processing
    time.sleep(0.01)
    return jsonify({
        "product": "Book",
        "id": "123",
        "reviews": [],
        "ratings": {"stars": 5}
    })
```

```
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=9080, threaded=True)
```

A.4.2.3. Node.js Express (ratings)

Listing A.13: Servicio ratings con Express

```
const express = require('express');
const app = express();

app.get('/health', (req, res) => {
    res.json({status: 'Ratings is healthy'});
});

app.get('/ratings/:productId', (req, res) => {
    const productId = req.params.productId;
    res.json({
        productId: productId,
        ratings: {
            'reviewer1': 5,
            'reviewer2': 4
        },
        average: 4.5
    });
});

const port = process.argv[2] || 9080;
app.listen(port, '0.0.0.0', () => {
    console.log(`Server running on port ${port}`);
});
```

A.5. Análisis Estadístico Adicional

A.5.1. Pruebas de Normalidad

Cuadro A.4: Resultados de pruebas Shapiro-Wilk para normalidad

Servicio	Métrica	Estadístico W	p-valor	Distribución
productpage	Throughput	0.945	0.023	No normal
ratings	Throughput	0.968	0.142	Normal
reviews	Throughput	0.952	0.067	Normal (límite)
productpage	Latencia	0.931	0.008	No normal
ratings	Latencia	0.974	0.234	Normal
reviews	Latencia	0.961	0.089	Normal

A.5.2. Análisis de Tamaño del Efecto

Cuadro A.5: Cálculo de d de Cohen para medir tamaño del efecto

Servicio	Media Std	Media Opt	d de Cohen	Interpretación
productpage	1846.34	1171.00	-1.82	Efecto negativo muy grande
ratings	8820.00	11280.00	1.57	Efecto positivo muy grande
reviews	3243.00	3507.00	0.52	Efecto mediano

A.5.3. Intervalos de Confianza

Cuadro A.6: Intervalos de confianza al 95 % para mejoras de throughput

Servicio	Mejora (%)	IC Inferior	IC Superior	Significativo
productpage	-36.6	-41.2	-32.0	Sí (negativo)
ratings	27.9	24.1	31.7	Sí (positivo)
reviews	8.2	-1.3	17.7	No

A.6. Guía de Reproducción

A.6.1. Requisitos del Sistema

A.6.1.1. Hardware Mínimo

- CPU: ARM64 (Apple Silicon M1/M2 o Raspberry Pi 4)

- RAM: 8 GB mínimo, 16 GB recomendado
- Almacenamiento: 20 GB libres
- Sistema Operativo: macOS 12+ o Ubuntu 20.04+

A.6.1.2. Software Requerido

Listing A.14: Versiones de software utilizadas

```
# Docker
Docker version 24.0.5, build ced0996
Docker Compose version v2.20.2

# Lenguajes y herramientas
Python 3.9.7
Node.js v18.16.0
Java OpenJDK 11.0.19
Gradle 7.6

# Herramientas de compilaci n
Nuitka 1.8.4
pkg 5.8.0
GraalVM CE 22.3.3

# Herramientas de an lisis
Apache Bench (ab) 2.3
jq 1.6
pandas 2.0.3
matplotlib 3.7.1
scipy 1.11.1
```

A.6.2. Pasos para Reproducir

A.6.2.1. 1. Clonar el Repositorio

```
git clone https://github.com/rartigues/MT-Uni
cd MT-Uni/Codigo
```

A.6.2.2. 2. Inicializar el Entorno

```
# Hacer scripts ejecutables
chmod +x scripts/**/*.sh
```

```
# Verificar dependencias
./scripts/init_environment.sh

# Instalar dependencias Python para an lisis
pip install -r scripts/requirements.txt
```

A.6.2.3. 3. Ejecutar Benchmarks

```
# Ejecutar todos los benchmarks con m tricas mejoradas
./scripts/run_enhanced_benchmarks.sh

# O ejecutar por lenguaje espec fico
./scripts/python/run_all_benchmarks.sh
./scripts/java/run_all_benchmarks.sh
./scripts/nodejs/run_all_benchmarks.sh

# Para macro-benchmarks (Bookinfo)
cd macro-benchmarks
./scripts/run_enhanced_benchmarks.sh
```

A.6.2.4. 4. Generar Reportes

```
# Generar reporte comprehensivo
./scripts/generate_comprehensive_report.sh

# Los reportes se guardar n en:
# - reports/comprehensive_report.html
# - reports/[language]_[benchmark]_report.html
```

A.6.3. Troubleshooting

A.6.3.1. Errores Comunes

Error: Permisos de Docker

```
# Soluci n
sudo usermod -aG docker $USER
newgrp docker
# O temporalmente
sudo chmod 666 /var/run/docker.sock
```

Error: Arquitectura incompatible

```
# Para Raspberry Pi, especificar plataforma
export DOCKER_DEFAULT_PLATFORM=linux/arm64
# O usar scripts espec ficos
./scripts/run_enhanced_benchmarks_raspberry.sh
```

Error: Memoria insuficiente

```
# Aumentar swap en Raspberry Pi
sudo dphys-swapfile swapoff
sudo sed -i 's/CONF_SWAPSIZE=.*/CONF_SWAPSIZE=2048/' \
    /etc/dphys-swapfile
sudo dphys-swapfile setup
sudo dphys-swapfile swapon
```

Error: Thermal throttling

```
# Monitorear temperatura
vcgencmd measure_temp
# Reducir carga simult nea
export MAX_PARALLEL_BUILDS=1
```

A.7. Recursos Adicionales

A.7.1. Enlaces a Repositorios

- **Código fuente completo**: https://github.com/rartigues/MT-Uni
- **Bookinfo original**: https://github.com/nocalhost/bookinfo
- **Scripts de análisis**: Incluidos en `Codigo/scripts/`
- **Resultados experimentales**: Directorio `Codigo/macro-benchmarks/enhanced_results/`

A.7.2. Herramientas Utilizadas

A.7.2.1. Herramientas de Compilación

- **Python**: Nuitka 1.8.4, Cython (para variante cython)
- **Node.js**: pkg 5.8.0
- **Java**: OpenJDK 11 con optimizaciones JVM

A.7.2.2. Herramientas de Benchmarking y Análisis

- **Carga HTTP**: Apache Bench (ab) 2.3
- **Monitoreo**: Docker stats
- **Análisis**: Python 3.9 con pandas, matplotlib, scipy
- **Procesamiento**: jq 1.6 para JSON, awk para cálculos