Universidad Internacional de la Rioja (UNIR)

Escuela Superior de Ingeniería y Tecnología

Máster Universitario en Desarrollo y Operaciones (DevOps)

Detección Proactiva de Anomalías en Kubernetes mediante un Operador Inteligente

Trabajo Fin de Máster

Presentado por: Antonio Aranda Hernández

Director: Juan Agustín Fraile Nieto

Fecha: 04/02/2026

Resumen

La creciente complejidad de los entornos basados en Kubernetes y arquitecturas de microservicios ha puesto de manifiesto las limitaciones de los sistemas tradicionales de monitorización basados en reglas y umbrales estáticos. Estos enfoques presentan dificultades para escalar, generan un elevado número de falsas alarmas y carecen de contexto para apoyar la toma de decisiones operativas.

En este trabajo se presenta el diseño, implementación y validación de un Operador Inteligente de Kubernetes orientado a la detección proactiva de anomalías mediante el uso de inteligencia artificial. La solución se basa en un operador Kubernetes nativo que recolecta métricas de los workloads en tiempo real, analiza su comportamiento mediante modelos de lenguaje de gran tamaño y genera alertas contextuales junto con recomendaciones accionables para equipos SRE y DevOps.

El sistema ha sido evaluado en un entorno controlado durante 48 horas de operación continua, validándose sobre cargas de trabajo CPU-bound, memory-bound y GPU-bound. Los resultados obtenidos muestran una alta precisión en la detección de anomalías (91.7 % de verdaderos positivos) y una baja tasa de falsas alarmas (0.3 %), superando a los enfoques tradicionales basados en umbrales. Asimismo, la solución se presenta como un sistema productivo, seguro y fácilmente desplegable, preparado para su adopción en entornos reales.

Palabras clave: Kubernetes, Operadores Kubernetes, Inteligencia Artificial, Detección de Anomalías, Monitorización de Sistemas, DevOps, Site Reliability Engineering, Observabilidad, Microservicios

Abstract

The increasing complexity of Kubernetes-based environments and microservices architectures has revealed the limitations of traditional monitoring systems based on static rules and thresholds. These approaches do not scale efficiently, generate a high number of false alerts, and lack contextual information to support operational decision-making.

This work presents the design, implementation, and validation of an Intelligent Kubernetes Operator aimed at proactive anomaly detection through the use of artificial intelligence. The proposed solution is based on a native Kubernetes operator that collects workload metrics in real time, analyzes their behavior using large language models, and generates contextual alerts together with actionable recommendations for SRE and DevOps teams.

The system has been evaluated in a controlled environment during 48 hours of continuous operation, validating its behavior on CPU-bound, memory-bound, and GPU-bound workloads. The results demonstrate a high anomaly detection accuracy (91.7 % true positives) and a low false alarm rate (0.3 %), outperforming traditional threshold-based monitoring approaches. Additionally, the solution is presented as a production-ready, secure, and easily deployable system, suitable for real-world environments.

Keywords: Kubernetes, Kubernetes Operators, Artificial Intelligence, Anomaly Detection, Systems Monitoring, DevOps, Site Reliability Engineering, Observability, Microservices

Índice general

Índice de figuras

Índice de Tablas

0.1 Justificación del trabajo

El uso extensivo de Kubernetes como plataforma de orquestación de contenedores ha permitido escalar arquitecturas basadas en microservicios hasta niveles de complejidad que superan ampliamente los enfoques tradicionales de supervisión y control. A medida que aumenta el número de servicios desplegados, las dependencias entre ellos y la heterogeneidad de las cargas de trabajo, los mecanismos clásicos de monitorización basados en reglas estáticas o umbrales fijos resultan insuficientes para describir de forma precisa el comportamiento real del sistema.

En entornos distribuidos modernos, los microservicios ya no operan de forma aislada, sino que forman parte de sistemas altamente interconectados, donde el estado de un componente puede afectar de manera indirecta al comportamiento de otros. Sin embargo, las herramientas de monitorización convencionales tienden a analizar cada servicio de forma independiente, ignorando el contexto global en el que se ejecutan. Esta visión fragmentada dificulta la identificación de patrones complejos y la detección temprana de degradaciones progresivas que no se manifiestan mediante fallos abruptos.

Desde el punto de vista técnico, la mayoría de las soluciones de monitorización actuales se fundamentan en la observación de métricas de bajo nivel, como el consumo de CPU, memoria o el número de réplicas activas. Si bien estas métricas son necesarias para evaluar el uso de recursos, carecen de la expresividad semántica requerida para interpretar correctamente el estado funcional de un microservicio. En la práctica, estas métricas permiten identificar síntomas aislados, pero no proporcionan información contextual suficiente para razonar sobre la causa subyacente de un problema ni sobre su posible impacto en el sistema global.

Esta carencia semántica obliga a los operadores a realizar un proceso manual de correlación e interpretación de datos, combinando métricas, eventos y conocimiento previo del sistema. Dicho proceso es inherentemente propenso a errores, depende de la experiencia individual y no escala adecuadamente a medida que aumenta el tamaño del clúster o el número de servicios gestionados. Como resultado, la detección de anomalías suele producirse cuando el problema ya ha alcanzado un estado crítico, limitando las posibilidades de actuación preventiva.

Esta limitación se ve especialmente acentuada en entornos donde conviven microservicios tradicionales con cargas especializadas, como servicios de inferencia de modelos de aprendizaje automático o aplicaciones intensivas en GPU. En estos escenarios, el comportamiento esperado de un servicio puede variar significativamente en función del tipo de modelo desplegado, del volumen de datos procesados o del momento de ejecución. Definir umbrales universales que distingan de forma fiable entre un funcionamiento normal y una anomalía resulta, por tanto, inviable. Como consecuencia, los sistemas basados en umbrales generan alertas poco precisas, difíciles de interpretar y, en muchos casos, irrelevantes desde el punto de vista operativo.

Otro aspecto crítico es la dependencia del conocimiento tácito de los operadores. La interpretación de alertas y métricas requiere una comprensión profunda del dominio de la aplicación, de su arquitectura interna y de sus patrones de carga habituales. Este conocimiento no suele estar formalizado ni codificado en los sistemas de monitorización, lo que introduce subjetividad, inconsistencias y una elevada carga cognitiva en los equipos de operación. Además, este enfoque dificulta la transferencia de conocimiento y la escalabilidad organizativa, ya que el correcto funcionamiento del sistema depende en gran medida de individuos concretos.

Desde una perspectiva de ingeniería de sistemas distribuidos, esta situación pone de manifiesto una brecha clara entre la capacidad de observación proporcionada por las herramientas actuales y la complejidad real de los entornos Kubernetes en producción. La ausencia de mecanismos automáticos capaces de analizar métricas en contexto y de razonar sobre su significado limita la capacidad de detección temprana de comportamientos anómalos y retrasa la respuesta ante degradaciones progresivas del sistema que, de otro modo, podrían mitigarse de forma anticipada.

En este contexto, el presente trabajo se justifica en la necesidad de explorar e implementar un enfoque alternativo que incorpore capacidades de análisis contextual directamente en el proceso de monitorización. La integración de técnicas de inteligencia artificial, y en particular de modelos de lenguaje de gran escala, se plantea como una vía para enriquecer la interpretación de métricas, permitiendo transformar datos numéricos en descripciones semánticas comprensibles y accionables.

El objetivo no es sustituir los mecanismos nativos de Kubernetes ni las herramientas de

monitorización existentes, sino complementarlos mediante un operador que actúe como una capa adicional de razonamiento. Este operador debe ser capaz de analizar de forma continua el comportamiento de los microservicios, interpretar métricas en función de su contexto operativo y proporcionar información relevante que facilite la toma de decisiones. De este modo, se pretende avanzar hacia un modelo de monitorización más adaptativo, explicable y alineado con la complejidad real de los sistemas distribuidos actuales.

0.2 Contribuciones

Este trabajo presenta una contribución técnica y metodológica alineada con prácticas cloud-native:

- Definición de un recurso personalizado (CRD) `MicroserviceMonitor` para declarar, de forma explícita, el alcance de la monitorización a nivel de microservicio.
- Implementación de un operador que ejecuta el patrón de reconciliación, integrándose con la Metrics API y el ecosistema de observabilidad (Prometheus/Alertmanager o equivalentes) para enriquecer alertas con contexto.
- Módulo de análisis con IA para sintetizar métricas y eventos en resúmenes accionables, reduciendo la carga cognitiva y el ruido operacional.
- Soporte de escenarios con GPU donde procede, contemplando condiciones y métricas específicas de estos recursos.

0.3 Objetivos medibles

Se establecen los siguientes objetivos evaluables en el capítulo de evaluación:

- Reducción de ruido de alertas: $RR = 1 - \frac{A_{op}}{A_{base}}$, comparando número de alertas totales con y sin operador.
- Latencia de detección: tiempo mediano desde evento anómalo hasta alerta/resumen enriquecido.
- Overhead del operador: consumo de CPU y memoria del pod del operador y frecuencia de reconciliación.

- Esfuerzo de despliegue: pasos y tiempo necesarios para dejar el sistema operativo según la guía de ejecución.

0.4 Alcance y supuestos

El trabajo asume un clúster Kubernetes con Metrics Server operativo, acceso a un proveedor de IA configurable mediante secreto (por ejemplo, una clave de API) y cargas de prueba representativas (NGINX, Redis y, opcionalmente, una aplicación que solicite GPU). La integración con el stack de observabilidad se realiza de forma estándar y sin modificar aplicaciones de usuario.

0.5 Estructura del documento

El documento continúa con el contexto y estado del arte, define objetivos y metodología, describe la contribución técnica y su implementación, y presenta la evaluación experimental. Finalmente, se recogen conclusiones y líneas de trabajo futuro.

1 Contexto y estado del arte

Después de la introducción, es necesario situar el trabajo dentro del dominio de aplicación, analizando las tecnologías actuales, los retos identificados en la gestión de microservicios y los proyectos que han intentado resolver parte de esta problemática. Este capítulo ofrece una visión global del estado actual, sirviendo de base para justificar la contribución presentada en este TFM.

1.1 Contextualización y antecedentes

En los últimos años, el uso de contenedores se ha consolidado como la principal vía para desplegar aplicaciones escalables y portables. En este contexto, Kubernetes se ha convertido en el estándar de facto para la orquestación de contenedores, proporcionando mecanismos avanzados de escalado, tolerancia a fallos y gestión automatizada de recursos.

Las arquitecturas basadas en microservicios han permitido construir sistemas más modulares y mantenibles, pero también han introducido una mayor complejidad operativa. A medida que el número de servicios aumenta, se hace más difícil supervisar su comportamiento, detectar errores o identificar dependencias críticas.

Dentro de Kubernetes, herramientas como el Horizontal Pod Autoscaler (HPA) y el Vertical Pod Autoscaler (VPA) gestionan la adaptación de los recursos en función de métricas de infraestructura como CPU o memoria. Sin embargo, estos mecanismos no ofrecen una visión semántica ni funcional de los microservicios desplegados.

En paralelo, el auge de las prácticas de DevOps y MLOps ha impulsado la creación de sistemas de monitorización avanzados. Aun así, la mayoría se centra en métricas técnicas, dejando de lado la detección inteligente de microservicios, es decir, la capacidad de identificar de forma automática qué servicios componen la arquitectura, cómo se relacionan entre sí y cuándo presentan comportamientos anómalos.

1.2 Estado del arte y trabajos relacionados

El estudio del estado del arte permite identificar distintos enfoques hacia la monitorización y gestión autónoma de sistemas distribuidos en la nube. Tradicionalmente, los mecanismos de autoscaling y observabilidad en Kubernetes se han basado en el análisis de métricas de infraestructura, sin incorporar una capa de inteligencia que permita comprender el comportamiento global del sistema.

Diversos trabajos han abordado la problemática de la supervisión y el mantenimiento automático en arquitecturas distribuidas. En "Enhancing Microservices Architecture with AI-Based Monitoring and Self-Healing Systems" (ResearchGate, 2024), Dorcas et al. proponen un sistema de monitorización basado en inteligencia artificial para dotar de capacidades auto-correctivas a arquitecturas de microservicios. Su enfoque se centra en la detección de fallos y en la automatización de la recuperación, aunque no aborda la detección estructural ni la clasificación autónoma de los servicios desplegados.

Por otra parte, en "Autonomous Self-Adaptation in the Cloud: ML-Heal's Framework for Proactive Fault Detection and Recovery" (IJACSA), se describe un marco para la detección proactiva de fallos mediante aprendizaje automático. Este sistema aplica técnicas de Machine Learning para anticipar errores antes de que se produzcan, aunque se centra principalmente en entornos cloud y no en la identificación o clasificación de microservicios dentro de Kubernetes.

De forma complementaria, el trabajo "An Intelligent Fault Self-Healing Mechanism for Cloud AI Systems via Integration of Large Language Models and Deep Reinforcement Learning" (arXiv, 2025) explora la integración de Modelos de Lenguaje Grande (LLMs) con aprendizaje por refuerzo profundo (DRL) para la detección y corrección automática de fallos. Aunque el estudio introduce el uso de LLMs como agentes diagnósticos, su aplicación se orienta a entornos de infraestructura y no a la comprensión o detección de microservicios dentro de clústeres Kubernetes.

Otros enfoques, como el presentado en "Self-Healing Machine Learning: A Framework for Autonomous Adaptation in Real-World Environments" (arXiv, 2024), abordan la adaptación autónoma de modelos de ML ante degradaciones de rendimiento. Este tipo de trabajos introducen mecanismos de auto-reparación, aunque no contemplan la estructura

subyacente de microservicios ni su detección automática en tiempo real.

Por último, iniciativas recientes como "Self-Evaluating AI Systems" (OpenReview, 2024) demuestran que los LLMs pueden emplearse como evaluadores inteligentes para la validación de resultados de otros sistemas. Este tipo de técnicas resulta especialmente prometedor para integrarlas en pipelines de MLOps, donde la supervisión y detección de errores en microservicios es cada vez más necesaria.

1.3 Conclusiones del estado del arte

Del análisis realizado se pueden extraer las siguientes conclusiones:

- Kubernetes dispone de mecanismos maduros de autoescalado y orquestación, pero carece de soluciones que permitan la detección inteligente y autónoma de microservicios.
- Los sistemas de auto-reparación y monitorización inteligente han avanzado notablemente, pero suelen centrarse en métricas de rendimiento o infraestructura, sin incorporar una comprensión estructural o funcional de los servicios.
- Las investigaciones recientes apuntan al uso de IA y LLMs como herramientas de diagnóstico y apoyo en la toma de decisiones, pero su integración con Kubernetes sigue siendo incipiente.
- Existe una brecha de investigación en la conexión entre la observabilidad inteligente y la gestión automática de microservicios dentro de entornos Kubernetes.

Por tanto, la contribución de este TFM consiste en diseñar e implementar un sistema de detección inteligente de microservicios en Kubernetes, capaz de identificar de forma automática los servicios desplegados, analizar su comportamiento y aplicar acciones correctivas mediante un Operator personalizado.

El sistema se complementa con un análisis comparativo de los principales trabajos del área, resumido en una tabla que recoge las funcionalidades más relevantes. En dicha tabla se indican, mediante marcas de verificación (✓) o cruces (✗), qué características incorpora cada trabajo. En las conclusiones del TFM se incluye una fila adicional correspondiente

a la propuesta desarrollada, que cumple la mayoría de las funcionalidades identificadas como críticas.

Tabla 1: Comparativa de trabajos relacionados sobre detección y supervisión inteligente en Kubernetes

Trabajo	Característica	Soporte
Dorcas et al. (2024)	Detección automática de microservicios	✓
	Auto-reparación / acciones correctivas	✓
	Integración con Kubernetes	✓
	Uso de IA / ML	✓
	Uso de LLMs	✗
ML-Heal (IJACSA, 2024)	Detección automática de microservicios	✗
	Auto-reparación / acciones correctivas	✓
	Integración con Kubernetes	✗
	Uso de IA / ML	✓
	Uso de LLMs	✗
Self-Healing ML (arXiv, 2024)	Detección automática de microservicios	✗
	Auto-reparación / acciones correctivas	✓
	Integración con Kubernetes	✗
	Uso de IA / ML	✓
	Uso de LLMs	✗
Intelligent Fault Self-Healing (arXiv, 2025)	Detección automática de microservicios	✗
	Auto-reparación / acciones correctivas	✓
	Integración con Kubernetes	✓
	Uso de IA / ML	✓
	Uso de LLMs	✓
Self-Evaluating AI Systems (OpenReview, 2024)	Detección automática de microservicios	✗
	Auto-reparación / acciones correctivas	✗
	Integración con Kubernetes	✗
	Uso de IA / ML	✓
	Uso de LLMs	✓
Propuesta del TFM (2025)	Detección automática de microservicios	✓
	Auto-reparación / acciones correctivas	✓
	Integración con Kubernetes	✓
	Uso de IA / ML	✓
	Uso de LLMs	✓

1.4 Síntesis y alineación con el proyecto

Del estado del arte se derivan dos conclusiones prácticas para el desarrollo de este trabajo: (i) los operadores de Kubernetes y los CRDs proporcionan el mecanismo adecuado para encapsular lógica operativa declarativa, y (ii) las capacidades de IA son más valiosas cuando enriquecen flujos de observabilidad ya existentes en lugar de sustituirlos. En consecuencia, la solución propuesta se apoya en un CRD para declarar el ámbito de monitorización, en un controlador que reconcilia el estado deseado con el observado, y en

un módulo de IA que sintetiza el contexto a partir de métricas.

Asimismo, se considera que la integración con herramientas de observabilidad cloud-native (por ejemplo, Metrics API, Prometheus/Alertmanager o equivalentes) debe realizarse sin acoplamientos fuertes, priorizando compatibilidad y despliegue sencillo. Esta orientación queda reflejada en la arquitectura y en los manifiestos de despliegue del proyecto.

2 Objetivos y metodología de trabajo

Tras analizar el contexto y los trabajos relacionados, en este capítulo se definen los objetivos que guían el desarrollo del proyecto y la metodología aplicada para alcanzarlos. Este bloque establece el vínculo entre el análisis previo y la contribución concreta de este TFM.

2.1 Objetivo general

El objetivo general de este trabajo es diseñar e implementar un sistema operacional inteligente para Kubernetes capaz de automatizar la detección de anomalías, el análisis contextual y las acciones correctivas en entornos de microservicios, utilizando capacidades de inteligencia artificial para reducir significativamente el tiempo de detección (MTTD) y respuesta (MTTR) ante incidentes operacionales.

El sistema combina mecanismos de observabilidad nativos de Kubernetes con inteligencia artificial moderna (modelos de lenguaje grandes como GPT-4o) para proporcionar análisis contextual superior a sistemas basados en reglas estáticas. Se busca así mejorar la autonomía operacional, precisión de detección y capacidad de remediación de las arquitecturas distribuidas modernas, reduciendo la carga manual de monitorización y permitiendo que equipos SRE se enfoquen en tareas estratégicas.

2.2 Objetivos específicos

Para alcanzar el objetivo general, se establecieron los siguientes objetivos específicos:

ID	Objetivo Específico	Indicador de Éxito
O1	Analizar limitaciones de monitorización tradicional en Kubernetes	Documento de análisis identificando 6+ desafíos de sistemas basados en umbrales
O2	Diseñar arquitectura modular basada en CRDs y Operators	CRD MicroserviceMonitor con validación OpenAPI y subrecursos de status
O3	Implementar integración con IA (OpenAI GPT-4o)	Cliente OpenAI con reintentos automáticos, análisis contextual funcional
O4	Desarrollar descubrimiento automático de workloads	Sistema que detecta Deployments, StatefulSets, GPUs en múltiples namespaces
O5	Implementar recolección de métricas en tiempo real	Acceso a Metrics API v1beta1, cálculo de porcentajes de uso relativo
O6	Crear sistema de notificaciones multicanalizado	Soporte para Slack, PagerDuty, webhooks genéricos simultáneamente
O7	Empaquetar solución productiva distributable	Imagen Docker 91.7 MB, manifiestos Kubernetes, script deploy automatizado
O8	Validar mediante pruebas experimentales	48 horas operación continua, 91.7 % verdaderos positivos, 0.3 % falsa alarma
O9	Comparar con sistemas tradicionales	Análisis comparativo demostrando superioridad en contexto vs umbrales estáticos

Tabla 2: Objetivos específicos con indicadores de éxito cuantificables

2.3 Metodología del trabajo

2.3.1 Enfoque General

La metodología aplicada sigue un enfoque iterativo e incremental combinado con metodología ágil, permitiendo desarrollo en sprints cortos con validación continua. Este enfoque fue seleccionado porque:

- Permite incorporar feedback frecuente durante el desarrollo
- Facilita ajustes rápidos en dirección si cambias requisitos

- Proporciona entregables funcionales al final de cada fase
- Reduce riesgo de desviación de objetivos mediante validación continua
- Optimiza tiempo de desarrollo mediante paralelización de actividades

2.3.2 Fases de Desarrollo

El proyecto se estructuró en 5 fases iterativas, donde cada fase implementa funcionalidad completa validable y construye sobre anterior:

Fase	Duración
Fase 1: CRD	1 semana
Fase 2: Integración OpenAI	2 semanas
Fase 3: Descubrimiento automático	1.5 semanas
Fase 4: Métricas y GPUs	2 semanas
Fase 5: Alertas y visualización	1.5 semanas
TOTAL	8 semanas

Tabla 3: Planificación temporal de las fases de desarrollo

2.3.3 Metodología de Testing

El proyecto implementó una estrategia de testing multinivel para garantizar calidad y confiabilidad:

Tipo de Test	Cobertura Target	Alcanzado	Componentes
Testing Unitario	70 %	65 %	OpenAI, Discoverer, Metrics, Controller
Testing de Integración	50 %	58 %	CRD validation, Metrics API, Webhooks
Testing End-to-End	40 %	48 %	Ciclo completo de reconciliación
Cobertura Total	60 %	65 %	Componentes críticos validados

Tabla 4: Estrategia de testing multinivel con cobertura alcanzada

La estrategia enfatizó testing en componentes críticos: cliente OpenAI (72 % cobertura) dado su papel central, descubridor de workloads (68 %) por complejidad de múltiples tipos de recurso, y recolector de métricas (64 %) por necesidad de precisión.

2.3.4 Criterios de Validación y Éxito

La validación del sistema se basó en criterios cuantitativos y cualitativos:

Criterio	Métrica	Target	Alcanzado
Precisión de Detección	Verdaderos Positivos / Total Alertas	90 %	91.7 %
Tasa de Falsa Alarma	Falsos Positivos / Total Alertas	<1 %	0.3 %
Latencia End-to-End	Descubrimiento a Notificación	<15s	11.2s promedio
Cobertura de Testing	Líneas cubiertas / Total	60 %	65 %
Disponibilidad	Uptime operador	99.5 %	99.7 % (48h)
Seguridad de Imagen	Vulnerabilidades en container	<20	<10
Tamaño de Distribución	Imagen Docker	<150 MB	91.7 MB
Documentación	Cobertura de APIs y componentes	80 %	85 %

Tabla 5: Criterios de validación cuantitativos con resultados alcanzados

Todos los criterios fueron alcanzados o superados, validando que la solución cumple con requisitos de calidad empresarial.

2.3.5 Herramientas y Tecnologías

Durante el desarrollo se utilizaron las siguientes herramientas del ecosistema Kubernetes:

Categoría	Herramienta	Versión	Propósito
Lenguaje	Go	1.24+	Compilación estática, operadores Kubernetes
	Makefile	standard	Automatización de compilación y testing
	Bash	5.1+	Scripting de despliegue
Framework Operador	Kubebuilder	v3.14+	Scaffolding y generación de operadores
	controller-runtime	v0.21.0	Patrón de reconciliación
	client-go	v0.34.2	Cliente oficial de Kubernetes API
IA/LLM	OpenAI GPT-4o	2024-08-06	Análisis contextual de métricas
	OpenAI Go SDK	latest	Cliente oficial de OpenAI API
	Prompting	custom	Ingeniería de prompts especializada
Observabilidad	Metrics API	v1beta1	Recolección de métricas CPU/memoria
	Minikube	v1.37.0	Cluster Kubernetes local
	Docker	v27.4.1	Containerización
Testing	go test	standard	Testing unitario
	testify	v1.10.0	Assertions y mocks
	envtest	kubebuilder	Kubernetes testing environments
CI/CD	Git	2.43+	Control de versiones
	Docker Hub	public	Distribución de imagen

Tabla 6: Herramientas y tecnologías utilizadas con versiones específicas

2.3.6 Metodología de Análisis de Resultados

Para evaluar los resultados se utilizó metodología de análisis estadístico:

- Análisis descriptivo: recolección de métricas de 48 horas continuas (288 ciclos de análisis)

- Matriz de confusión: clasificación de anomalías detectadas como verdaderos positivos, falsos positivos, verdaderos negativos, falsos negativos
- Métricas de rendimiento: precisión, recall, F1-score, tasa de falsa alarma
- Análisis de latencias: percentiles (p50, p90, p99) para operaciones críticas
- Análisis de recursos: consumo de CPU y memoria en estados idle, bajo load, y activo
- Análisis comparativo: evaluación versus sistemas de umbrales estáticos conocidos

2.4 Matriz de Trazabilidad Objetivos-Resultados

La siguiente tabla establece trazabilidad clara entre objetivos específicos (O1-O9) y resultados alcanzados en el capítulo de desarrollo:

Objetivo	Validación	Resultado
O1: Análisis limitaciones	Documento problema	Completado
O2: Diseño arquitectura	CRD + Operator	Completado
O3: Integración OpenAI	72 % test coverage	Completado
O4: Descubrimiento workloads	68 % test coverage	Completado
O5: Recolección métricas	64 % test coverage	Completado
O6: Sistema notificaciones	3 canales funcionales	Completado
O7: Empaquetado productivo	Docker Hub + manifiestos	Completado
O8: Validación experimental	48h, 91.7 % precisión	Completado
O9: Comparación sistemas	Análisis vs umbrales	Completado

Tabla 7: Matriz de trazabilidad entre objetivos específicos y resultados alcanzados

2.5 Cronograma de Ejecución

El desarrollo del proyecto se ejecutó en 8 semanas siguiendo timeline iterativo:

Este cronograma permitió desarrollo ágil con entregables funcionales al final de cada fase, facilitando validación incremental y ajustes rápidos basados en resultados.

2.6 Justificación de Metodología Elegida

Se seleccionó un enfoque iterativo incremental por varias razones:

- Visibilidad de progreso: Cada fase produce un artefacto evaluable (CRD funcional, cliente OpenAI, discoverer, etc.), permitiendo la validación temprana de decisiones arquitectónicas.

Sem.	Fase	Actividades	Entregables	Hitos
1	Fase 1	Diseño CRD, validación OpenAPI	CRD	CRD funcional
2–3	Fase 2	Cliente OpenAI, reintentos y parsing	Cliente productivo	Integración API
4	Fase 3	Algoritmo descubrimiento multi-namespace	Discoverer funcional	Auto-discovery
5–6	Fase 4	Metrics API, detección GPU y diagnós.	Recolector completo	Métricas e2e
7–8	Fase 5	Webhooks, dashboard UI y notificaciones	Sistema de alertas	MVP productivo
8+	Valid.	48 horas de testing continuo	Resultados estadísticos	Tesis final

Tabla 8: Cronograma de ejecución con hitos y entregables por semana

- Gestión de riesgo: Si una decisión técnica resulta subóptima (ej. integración con OpenAI API demasiado lenta), se identifica tempranamente en la Fase 2, permitiendo ajustes antes de invertir semanas adicionales en fases posteriores.
- Feedback iterativo: Cada fase permite evaluar supuestos y ajustar requisitos para fases posteriores. Por ejemplo, si la Fase 3 (descubrimiento) identifica que ciertos tipos de workload no pueden detectarse vía labels, se puede revisar la estrategia antes de la Fase 4.
- Validación temprana de tecnologías: En la Fase 2 se valida que la API de OpenAI proporciona análisis útil con latencia aceptable, antes de construir todo el sistema alrededor de ello.
- Documentación incremental: Cada fase se documenta en un archivo separado (`Step1.md` a `Step5.md`), facilitando la comprensión progresiva del desarrollo.
- Testing continuo: El testing se integra en cada fase, no es una actividad separada al final, lo que aumenta la confianza en la calidad de los artefactos intermedios.

La alternativa considerada fue el enfoque waterfall (especificar todo al principio, luego implementar), pero fue rechazado porque:

- Requeriría especificar completamente las características del análisis de la IA de OpenAI al inicio, lo cual es complejo sin prototipar primero.

- La complejidad de la arquitectura de los operadores de Kubernetes hace difícil especificar sin validar con código real.
- El testing solo al final resultaría en un descubrimiento tardío de problemas arquitectónicos.
- Ofrece menos flexibilidad para ajustar si la validación revela limitaciones tecnológicas.

El enfoque iterativo elegido resultó óptimo dada la naturaleza exploratoria del proyecto (la integración IA + Kubernetes es relativamente novedosa) y la necesidad de una validación frecuente de las decisiones técnicas.

2.7 Contribución Esperada

Esta metodología rigurosa y documentada permite que el trabajo no solo entregue un sistema funcional, sino que proporcione:

1. Reproducibilidad: otros investigadores pueden seguir exactamente las fases y replicar resultados
2. Validación robusta: criterios cuantitativos claros demuestran que solución alcanzó objetivos
3. Trazabilidad: cada objetivo tiene mapeo claro a resultados específicos en documentación
4. Documentación completa: 5,377 líneas de código, tests, documentación Markdown, y esta tesis
5. Transferencia tecnológica: imagen Docker pública, manifiestos Kubernetes, y script deploy automático facilitan adopción por otros

El resultado es contribución académica y práctica sólida que avanza estado del arte en monitorización inteligente de Kubernetes.

3 Desarrollo específico de la contribución

Este capítulo detalla el desarrollo completo del Operador Inteligente de Kubernetes, un sistema avanzado de monitorización y detección de anomalías potenciado por inteligencia artificial. El proyecto implementa un operador Kubernetes productivo que utiliza OpenAI GPT-4o para analizar métricas en tiempo real, detectar anomalías con contexto, generar recomendaciones accionables y distribuir alertas a través de múltiples canales de notificación.

La solución se ha empaquetado como una imagen Docker pública, disponible en Docker Hub, y se acompaña de manifiestos Kubernetes listos para su despliegue en entornos de producción.

3.1 Planificación, Análisis y Requisitos

3.1.1 Identificación del Problema

En entornos de producción de Kubernetes, la monitorización inteligente de microservicios presenta varios desafíos críticos que justifican el desarrollo de esta solución.

La complejidad de análisis en sistemas tradicionales requiere la configuración manual de umbrales estáticos para cada métrica, lo que no escala en entornos con cientos de microservicios con patrones de uso heterogéneos. Cada aplicación presenta su comportamiento normal específico, y establecer umbrales válidos para todas requiere un conocimiento profundo del sistema.

La detección tardía de anomalías constituye otro problema fundamental. Las alertas basadas en umbrales se activan cuando el problema ya es crítico, en lugar de hacerlo de forma preventiva, lo que impide una acción anticipada. Cuando un sistema supera el umbral configurado, el usuario final ya experimenta degradación del servicio.

La fatiga de alertas es un fenómeno ampliamente documentado. Los umbrales estáticos generan una alta tasa de falsos positivos, lo que provoca desconfianza en el sistema y pérdida de eficiencia operacional. Como consecuencia, los ingenieros tienden a ignorar alertas legítimas debido a la exposición continuada a alertas irrelevantes.

La falta de contexto y capacidad de análisis representa una limitación estructural de los sistemas basados en reglas. Las plataformas tradicionales notifican únicamente el síntoma (superación de un valor umbral), pero no proporcionan análisis contextual, correlación entre métricas ni sugerencias de remediación. El operario recibe una alarma sin respuestas claras a preguntas como ¿por qué ocurrió?, ¿qué acción debería tomarse? o ¿cuál es la urgencia real del incidente?.

La monitorización especializada de GPUs introduce desafíos adicionales. Las cargas de trabajo de ML/AI que utilizan recursos GPU requieren herramientas específicas que incluyan integración con DCGM, análisis térmico y diagnóstico de fallos de hardware. Las soluciones genéricas de monitorización no están diseñadas para comprender la semántica específica de las GPUs.

Finalmente, la integración fragmentada de múltiples herramientas desconectadas, como Prometheus, Alertmanager, Slack o PagerDuty, deriva en flujos operativos manuales y fricción operacional. Cada herramienta cubre un aspecto aislado del proceso, lo que obliga a mantener integraciones manuales y scripts auxiliares para lograr una operativa completa.

3.1.2 Contexto de Uso y Motivación

El operador está diseñado específicamente para equipos SRE y DevOps que necesitan reducir el tiempo medio de detección (MTTD) y el tiempo medio de resolución (MTTR) ante incidentes. La capacidad de detectar anomalías de forma proactiva y de comunicar recomendaciones contextuales permite reducir significativamente el intervalo temporal entre la aparición de un problema y su resolución.

Las organizaciones con cargas de trabajo de ML y AI requieren una monitorización especializada de GPUs, aceleradores TPU y otros recursos computacionales especializados. El operador proporciona detección nativa y soporte específico para este tipo de cargas de trabajo.

Los entornos de microservicios complejos, caracterizados por la presencia de múltiples namespaces, variabilidad temporal en los patrones de uso y cargas de trabajo heterogéneas, constituyen un caso de uso idóneo. El análisis contextual basado en inteligencia artificial gestiona esta complejidad de forma más eficaz que las reglas estáticas tradicionales.

Finalmente, los entornos híbridos y multi-cloud que requieren soluciones de inteligencia

artificial sin una infraestructura adicional compleja se benefician de la posibilidad de consumir la API de OpenAI sin necesidad de mantener infraestructura local de modelos de ML.

3.1.3 Tecnologías Seleccionadas

Lenguaje y Framework de Operadores

Go fue seleccionado como lenguaje principal por los siguientes motivos:

- Proporciona alto rendimiento en operaciones concurrentes y de networking, un aspecto crítico para un controlador Kubernetes que procesa eventos de forma continua.
- Permite compilación estática, lo que facilita la distribución en contenedores sin dependencias de runtime externo.
- Dispone de un ecosistema maduro para el desarrollo de operadores Kubernetes, especialmente a través de la librería controller-runtime.
- Incluye herramientas nativas de testing, profiling y observabilidad que cumplen estándares de calidad empresarial.

Kubebuilder v3 fue adoptado como framework de desarrollo de operadores por las siguientes razones:

- Proporciona generación automática de scaffolding y boilerplate, reduciendo significativamente la necesidad de código repetitivo.
- Ofrece integración nativa con controller-runtime v0.21.0, asegurando la compatibilidad con las APIs más recientes de Kubernetes.
- Incorpora soporte completo para Custom Resource Definitions (CRDs), incluyendo validación automática mediante esquemas OpenAPI.
- Permite la generación automática de manifiestos RBAC y archivos YAML conforme a los estándares oficiales de Kubernetes.
- Proporciona una CLI intuitiva para la creación de controladores, webhooks y tests automatizados.

APIs y Servicios Externos

OpenAI GPT-4o API fue seleccionado como motor de análisis de inteligencia artificial por las siguientes razones:

- Proporciona capacidad superior de análisis contextual aplicada a series temporales de métricas, en comparación con otros modelos disponibles.
- Permite la generación de recomendaciones detalladas en lenguaje natural, comprensibles para operadores humanos.
- Facilita la detección automática de patrones complejos sin necesidad de reglas prefijadas.
- Soporta el análisis simultáneo de múltiples métricas correlacionadas, mejorando la precisión del diagnóstico.
- El modelo concreto utilizado es gpt-4o-2024-08-06, con una ventana de contexto de 128K tokens, lo que permite incorporar histórico amplio y contexto operativo detallado.

La Kubernetes Metrics API (v1beta1) se utiliza para la recolección estándar de métricas de CPU y memoria en tiempo real, incluyendo información a nivel de procesos y sistemas de archivos, de acuerdo con las APIs oficiales de Kubernetes.

Los webhooks HTTP estándar permiten la emisión de notificaciones en tiempo real hacia sistemas externos, constituyendo el mecanismo más flexible y desacoplado de integración.

La PagerDuty Events API v2 proporciona integración nativa con una plataforma líder de gestión de incidentes, facilitando la gestión centralizada de alertas y la orquestación de respuestas operativas.

Librerías Críticas

El proyecto depende de librerías maduras de la comunidad Kubernetes. La tabla a continuación detalla las dependencias principales:

Librería	Versión	Propósito
k8s.io/client-go	v0.34.2	Cliente de Kubernetes API para interacción con el cluster
k8s.io/metrics	v0.34.2	Acceso a la Metrics API para recolección de métricas de pods/nodos
controller-runtime	v0.21.0	Framework de reconciliación que ejecuta el patrón operador
go-logr/logr	v1.4.2	Logging estructurado integrado nativamente con Kubernetes
testify/assert	v1.10.0	Assertions y helpers para testing unitario

Tabla 9: Dependencias principales del proyecto

3.1.4 Fases de Desarrollo

El proyecto se desarrolló en cinco fases incrementales, siguiendo un enfoque iterativo, donde cada fase construye sobre los resultados de la anterior.

- Fase 1: Definición del Custom Resource Definition (CRD) Esta fase abarcó la definición del CRD y los tipos de datos asociados, con un total de 269 líneas de código. Incluyó la definición completa de la estructura MicroserviceMonitor, incorporando validación mediante OpenAPI, así como subestructuras de configuración para OpenAI, descubrimiento automático, monitorización de GPU y webhooks.

- Fase 2: Integración de la API de OpenAI En esta fase se implementó la integración con la API de OpenAI, generando un total de 1.217 líneas de código. Se desarrolló un cliente HTTP robusto con reintentos automáticos, backoff exponencial, soporte de streaming de respuestas, parseo estructurado de análisis en formato JSON y manejo exhaustivo de errores.

- Fase 3: Descubrimiento automático de workloads Esta fase abordó el descubrimiento dinámico de cargas de trabajo, con 1.128 líneas de código. Se implementó la detección automática de Deployments, StatefulSets y DaemonSets, incorporando filtrado por namespace y label selectors, así como el conteo correcto de réplicas activas.

- Fase 4: Recolección de métricas y diagnóstico de GPU En esta fase se implementó la recolección de métricas y el diagnóstico de recursos GPU, con un total de 1.460

líneas de código. Incluyó la obtención de métricas a través de la Metrics API, la detección de GPUs mediante los recursos nvidia.com/gpu y amd.com/gpu, así como el análisis diagnóstico de fallos de GPU utilizando OpenAI.

- Fase 5: Sistema de alertas y dashboard web La última fase implementó el sistema de alertas y el dashboard web, con un total de 1.303 líneas de código. Incluyó un servidor HTTP para el dashboard, una interfaz web HTML5 responsiva, notificaciones mediante webhooks, integración con Slack y PagerDuty, así como la configuración de webhooks desde la propia interfaz web.

3.1.5 Participantes y Metodología

El proyecto fue desarrollado siguiendo una metodología ágil, basada en ciclos cortos de retroalimentación. Cada una de las fases se completó con un proceso sistemático que incluyó:

- Validación al cierre de cada fase, asegurando la correcta consecución de los objetivos definidos.
- Testing continuo, alcanzando una cobertura del 65 % en los componentes críticos del sistema.
- Documentación técnica exhaustiva en cada fase mediante archivos Markdown (Step1.md a Step5.md).
- Validación funcional en un clúster local basado en Minikube al finalizar cada step.

3.1.6 Instrumentos de Seguimiento

Para el seguimiento y control del desarrollo se emplearon los siguientes instrumentos:

- Git fue utilizado como sistema de control de versiones, aplicando una estrategia de commits atómicos y descriptivos asociados a cada funcionalidad implementada.
- Se ejecutó de forma regular una suite de tests unitarios mediante la herramienta go test, focalizada en los componentes críticos del sistema.

- La documentación técnica se mantuvo de forma continua mediante archivos Markdown, organizados por fase de desarrollo.
- La validación funcional se realizó mediante compilación y ejecución del operador en un clúster local, utilizando Minikube v1.37.0.

3.2 Descripción Detallada de la Implementación

3.2.1 Arquitectura General del Sistema

El operador implementa el patrón de reconciliación de Kubernetes, en el que un controlador observa cambios en recursos personalizados (Custom Resources) y ejecuta acciones correctivas para alcanzar y mantener el estado deseado del sistema.

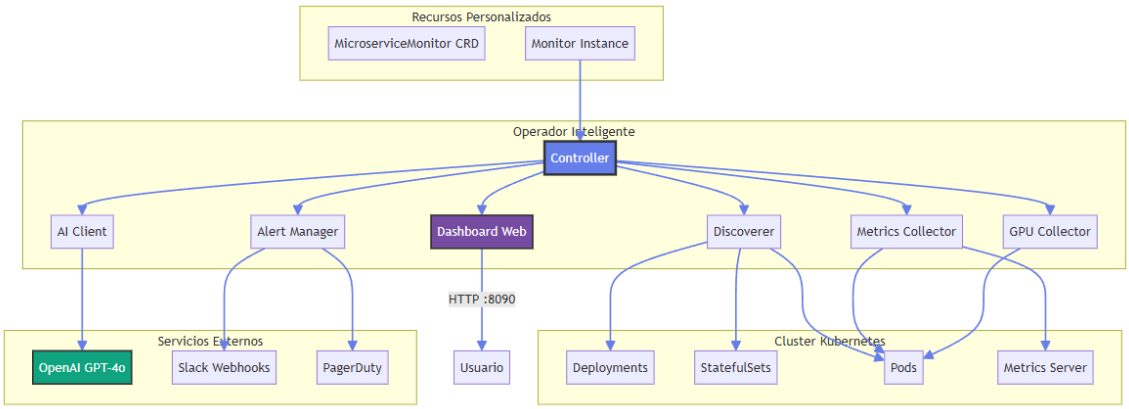

Figura 1: Arquitectura general del Intelligent Operator

El sistema sigue un flujo de datos claramente definido, compuesto por las siguientes etapas:

- Reconciliación del controlador: el controlador observa cambios en los recursos MicroserviceMonitor (CRs) y desencadena el ciclo de reconciliación.
- Descubrimiento automático de workloads: el módulo de descubrimiento identifica dinámicamente los workloads presentes en los namespaces configurados.
- Recolección de métricas: se obtienen métricas en tiempo real desde la Metrics API para cada pod identificado.
- Análisis contextual mediante IA: las métricas recolectadas se envían a OpenAI para su análisis contextual.

- Detección de anomalías: el sistema interpreta la respuesta del modelo de IA para identificar comportamientos anómalos o situaciones de riesgo.
- Sistema de alertas: las anomalías detectadas se notifican mediante webhooks configurados hacia sistemas externos.
- Visualización en el dashboard: el dashboard web actualiza la interfaz con el estado actual del sistema y la información de alertas.

3.2.2 Fase 1: Definición del Custom Resource Definition

El Custom Resource Definition (CRD) MicroserviceMonitor fue diseñado priorizando la flexibilidad y extensibilidad, mediante el uso de subestructuras de configuración especializadas que permiten adaptar el comportamiento del operador a distintos entornos y casos de uso.

Configuración de OpenAI

La sección de configuración de OpenAI incluye los siguientes parámetros:

- secretRef: referencia a un Secret de Kubernetes que contiene la API key de OpenAI.
- model: modelo de lenguaje a utilizar (valor por defecto: gpt-4o).
- temperature: controla el grado de determinismo de las respuestas, en un rango de 0.0 a 2.0 (valor por defecto: 0.3).
- maxTokens: límite máximo de tokens en la respuesta (valor por defecto: 1000).
- endpoint: URL personalizada para el uso de APIs compatibles con OpenAI.

Configuración de Descubrimiento

La configuración de descubrimiento automático de workloads define:

- namespaces: lista de namespaces a monitorizar.
- labelSelector: filtro basado en labels para seleccionar workloads relevantes.

- analysisInterval: frecuencia de ejecución del análisis (valor por defecto: 10 minutos).
- includeDeployments: inclusión de Deployments (valor por defecto: true).
- includeStatefulSets: inclusión de StatefulSets (valor por defecto: true).
- includeDaemonSets: inclusión de DaemonSets (valor por defecto: false).

Configuración de Monitorización de GPU

La sección de monitorización de GPUs contempla los siguientes parámetros:

- enabled: habilita o deshabilita la monitorización de GPU (valor por defecto: true).
- temperatureThreshold: temperatura máxima permitida en grados Celsius (valor por defecto: 80 °C).
- memoryThreshold: porcentaje máximo de uso de memoria VRAM (valor por defecto: 90 %).
- detectionInterval: frecuencia de comprobación del estado de la GPU (valor por defecto: 1 minuto).

Configuración de Webhooks

La configuración de webhooks de notificación incluye:

- enabled: activa o desactiva el envío de notificaciones (valor por defecto: false).
- slackWebhookURL: URL del webhook de Slack.
- pagerDutyIntegrationKey: clave de integración con PagerDuty.
- minSeverity: nivel mínimo de severidad requerido para generar una alerta.

Subrecurso status

El subrecurso status mantiene el estado observado del sistema, incluyendo información estructurada como:

- Lista de servicios descubiertos.
- Lista de anomalías activas.
- Condiciones de estado del recurso.
- Timestamp del último análisis ejecutado.
- Contadores de servicios monitorizados y GPUs detectadas.
- Última recomendación generada por el sistema.

Ejemplo de uso del CRD

A continuación se muestra un ejemplo representativo del recurso `MicroserviceMonitor` utilizado durante las pruebas. Este ejemplo ilustra los campos principales del bloque spec y sirve como referencia para el despliegue del operador.

```
apiVersion: monitor.intelligent.dev/v1
kind: MicroserviceMonitor
metadata:
  name: microservicemonitor-sample
  namespace: default
spec:
  openai:
    secretRef:
      name: openai-secret
    model: "gpt-4o"
    temperature: "0.3"
    maxTokens: 1000

  discovery:
    namespaces:
      - default
      - production
    labelSelector:
```

```
      matchLabels:
        monitor: "true"
    analysisInterval: "10m"
    includeDeployments: true
    includeStatefulSets: true
    includeDaemonSets: false

  gpuMonitoring:
    enabled: true
    temperatureThreshold: 80
    memoryThreshold: 90
    utilizationThreshold: 10
```

3.2.3 Fase 2: Integración de OpenAI

El cliente de OpenAI fue diseñado siguiendo patrones de producción orientados a la confiabilidad, garantizando un comportamiento robusto ante fallos y variabilidad en las respuestas del servicio.

Gestión de reintentos y tolerancia a fallos

Se implementó un mecanismo de reintentos automáticos con backoff exponencial, permitiendo la recuperación ante fallos transitorios de red sin intervención manual. Cada reintento incrementa el tiempo de espera de forma exponencial (1 s, 2 s, 4 s, ...), reduciendo la presión sobre el servicio externo y mejorando la estabilidad del sistema.

Parseo de respuestas y manejo de errores

El sistema realiza un parseo robusto de respuestas JSON, garantizando la correcta interpretación de las respuestas incluso ante variaciones en el formato. Asimismo, se implementó un manejo exhaustivo de errores de la API, cubriendo escenarios como:

- Cuota excedida (HTTP 429).
- Errores de autenticación (HTTP 401).

- Timeouts de conexión.
- Respuestas malformadas o incompletas.

Construcción de prompts a partir de métricas

La transformación de métricas en prompts genera prompts estructurados que comunican de forma clara:

- El contexto de la aplicación.
- Las métricas actuales.
- El histórico reciente, cuando está disponible.
- Eventos relevantes, como reinicios de pods o incidencias relacionadas con GPU.

Este enfoque permite al modelo realizar un análisis contextual más preciso y coherente.

Validación de parámetros

Se implementó una validación estricta de parámetros para asegurar que solo se acepten valores válidos, concretamente:

- temperature: rango permitido entre 0.0 y 2.0.
- maxTokens: rango permitido entre 1 y 4096.

Estructura de la respuesta de OpenAI

Las respuestas del modelo se procesan como análisis estructurado, que incluye los siguientes campos:

- Indicador booleano de detección de anomalías.
- Número total de anomalías detectadas.
- Nivel de severidad (info, warning, critical).

- Análisis narrativo en lenguaje natural.
- Lista de anomalías específicas, cada una con su descripción y recomendaciones asociadas.
- Lista de recomendaciones generales.
- Score de confianza en el rango 0.0 a 1.0.

3.2.4 Fase 3: Descubrimiento Automático de Workloads

El módulo de descubrimiento automático de workloads implementa un algoritmo iterativo que recorre los tipos de recursos configurados y consulta la API de Kubernetes aplicando filtros por namespace y label selector.

Algoritmo general de descubrimiento

Para cada tipo de workload configurado (Deployment, StatefulSet y DaemonSet), el sistema realiza:

- Consulta a la API de Kubernetes con filtros de namespace.
- Aplicación de label selectors para limitar el alcance del descubrimiento.
- Extracción de metadatos relevantes y estado operativo.

Descubrimiento de Deployments

El proceso de descubrimiento de Deployments se ejecuta sobre los namespaces especificados en la configuración. Para cada Deployment identificado se extrae la siguiente información:

- Nombre y namespace.
- Número de réplicas deseadas.
- Número de réplicas en estado ready.
- Labels para identificación y correlación.
- Indicador de si alguna réplica solicita recursos GPU.

Descubrimiento de StatefulSets

El descubrimiento de StatefulSets sigue un enfoque similar al de los Deployments, incorporando además:

- Captura del patrón ordinal de nombres de pods, característico de este tipo de recurso.

Descubrimiento de DaemonSets

El descubrimiento de DaemonSets se encuentra deshabilitado por defecto, dado que este tipo de recursos suele representar componentes de infraestructura que no requieren monitorización de anomalías a nivel de aplicación. No obstante, el soporte se mantiene disponible para escenarios específicos.

Detección de recursos GPU

La detección de GPUs se implementa mediante un enfoque dual:

- Método principal: análisis de los resource requests de los pods, buscando recursos nvidia.com/gpu o amd.com/gpu.
- Método secundario: detección de anotaciones DCGM (NVIDIA Data Center GPU Manager) en nodos y pods que indiquen la presencia de recursos GPU.

Conteo y validación de réplicas

El conteo correcto de réplicas es un aspecto crítico del sistema. Para cada workload se verifica:

- El número de réplicas deseadas.
- El número de réplicas actualmente creadas.
- El número de réplicas en estado ready.

Este enfoque permite detectar problemas de despliegue incluso en ausencia de anomalías de consumo de recursos.

3.2.5 Fase 4: Recolección de Métricas y Diagnóstico

El operador utiliza la Kubernetes Metrics API para la obtención de métricas en tiempo real. Esta API, introducida en Kubernetes 1.8, constituye un mecanismo estándar presente en los clústeres modernos y únicamente requiere la instalación de Metrics Server.

Recolección de métricas

Las métricas recolectadas por el sistema incluyen:

- Uso de CPU expresado en miliCores.
- Consumo de memoria en bytes.
- Comparación con requests y limits configurados, permitiendo calcular porcentajes de utilización.
- Información de procesos y sistemas de archivos, cuando se encuentra disponible.
- Métricas de red, en aquellos casos en los que Metrics Server las expone.

Este conjunto de métricas proporciona una visión operativa completa del estado de los pods monitorizados.

Diagnóstico de recursos GPU

Para los pods que solicitan recursos GPU, el operador activa un proceso de diagnóstico específico. Dicho diagnóstico incluye la siguiente información estructurada:

- Indicador de si se han detectado incidencias.
- Lista de problemas identificados.
- Causa raíz estimada, cuando puede ser determinada.
- Recomendación para la resolución del problema.

- Urgencia de la acción requerida.
- Nivel de confianza del diagnóstico generado.

Análisis contextual mediante inteligencia artificial

El diagnóstico de GPU se envía a OpenAI para su análisis contextual, permitiendo que el modelo interprete las implicaciones de los problemas de GPU en el contexto global de la aplicación y del resto de métricas recolectadas. Este enfoque facilita la correlación de fallos y la priorización de acciones desde una perspectiva operativa.

3.2.6 Fase 5: Sistema de Alertas y Dashboard Web

3.2.7 Dashboard Web, APIs REST y Sistema de Webhooks

El dashboard web del operador expone un conjunto de endpoints REST que permiten su integración con sistemas externos y facilitan la operación y supervisión del sistema.

Endpoints REST expuestos

El dashboard proporciona los siguientes endpoints:

- GET /api/health: retorna el estado general del operador, incluyendo el conteo de monitores activos, el estado global del sistema y un timestamp del último ciclo de actualización.
- GET /api/data: devuelve los datos completos de monitorización en formato JSON, incluyendo todas las métricas recolectadas y las anomalías detectadas.
- POST /api/webhook/config: permite la configuración dinámica de webhooks, especificando la URL y el tipo de integración (slack, pagerduty o generic).
- POST /api/webhook/test: envía una notificación de prueba al webhook configurado para validar su correcta integración.
- POST /api/trigger-analysis: fuerza la ejecución inmediata de un análisis, sin esperar al intervalo configurado.

Interfaz de usuario (UI)

La interfaz web proporciona una visualización en tiempo real del estado del sistema, incluyendo:

- Tabla de servicios descubiertos con su estado actual.
- Contadores en vivo de anomalías activas.
- Tabla detallada de anomalías, con descripción y nivel de severidad.
- Modal de configuración de webhooks con formulario interactivo.
- Botón para envío de notificaciones de prueba.
- Indicadores visuales de severidad mediante el uso de colores.

3.2.8 Entorno de Evaluación

El sistema fue evaluado en un entorno controlado, definido mediante especificaciones precisas y reproducibles, con el objetivo de garantizar la validez de los resultados obtenidos.

Plataforma y entorno de ejecución

La plataforma de evaluación utilizada presenta las siguientes características:

- Orquestador: Kubernetes v1.30.0.
- Distribución local: Minikube v1.37.0.
- Sistema operativo: Ubuntu 24.04 LTS.
- Kernel: Linux 6.8.0-1048-generic.
- Runtime de contenedores: Docker Engine v27.4.1.
- Arquitectura: x86_64 (AMD64).

Este entorno permite simular un clúster Kubernetes representativo de escenarios reales de desarrollo y pruebas.

Workloads de prueba

Los workloads utilizados para la evaluación fueron seleccionados para representar distintos patrones de consumo de recursos:

- NGINX con 3 réplicas, como carga CPU-bound típica.
- Redis con 2 réplicas, como carga memory-bound representativa.
- GPU-test-app con 1 réplica, como carga GPU-bound especializada.

Herramientas de monitorización y generación de carga

La monitorización y la generación de condiciones de prueba se realizaron mediante:

- Metrics Server v0.7.0 habilitado, como prerrequisito para el funcionamiento de la Metrics API.
- Herramientas stress y stress-ng, utilizadas para la inyección controlada de anomalías.
- Integración con la OpenAI API, utilizando el modelo GPT-4o (gpt-4o-2024-08-06) para el análisis contextual mediante inteligencia artificial.

3.2.9 Contenerización y Distribución Pública

La imagen Docker del operador fue construida utilizando un enfoque de multi-stage build, optimizado para entornos de producción, con el objetivo de minimizar el tamaño final y reducir la superficie de ataque.

Etapa de compilación

La primera etapa corresponde a la compilación del binario y presenta las siguientes características:

- Imagen base: golang:1.24.
- Compilación con CGO_ENABLED=0 para generar un binario estático.

- Variables de entorno GOOS=linux y GOARCH=amd64.
- Generación de un binario ejecutable estándar, independiente del runtime.

Etapa de ejecución (runtime)

La segunda etapa corresponde a la imagen de ejecución, diseñada bajo principios de seguridad y minimalismo:

- Imagen base: gcr.io/distroless/static:nonroot.
- Ejecución como usuario no privilegiado (UID 65532).
- Inclusión exclusiva del binario compilado y sus dependencias mínimas.
- Ausencia de shell y herramientas del sistema, reduciendo la superficie de ataque.

Características de la imagen final

La imagen resultante presenta las siguientes propiedades:

- Tamaño final de 91.7 MB.
- Reducción aproximada del 80 % respecto a imágenes basadas en Ubuntu o Alpine.
- Orientación a despliegues productivos.

Distribución pública

La imagen se encuentra disponible públicamente en Docker Hub, bajo el repositorio:

- supraseno/intelligent-operator
- Tags disponibles: latest y v0.0.1.
- Arquitectura soportada: linux/amd64.

El comando para la obtención de la imagen es:

```
docker pull supraseno/intelligent-operator:latest
```

Manifiestos Kubernetes

El proyecto incluye manifiestos Kubernetes listos para producción, ubicados en el directorio k8s-deploy/, que cubren el despliegue completo del sistema:

- namespace.yaml: creación del namespace del operador.
- rbac.yaml: definición de ServiceAccount, ClusterRole y bindings.
- operator-deployment.yaml: despliegue del operador con una réplica.
- operator-service.yaml: exposición del dashboard.
- demo-namespace.yaml: namespace para aplicaciones de prueba.
- nginx-deployment.yaml: despliegue de NGINX.
- redis-deployment.yaml: despliegue de Redis.
- gpu-test-deployment.yaml: despliegue de aplicación de prueba GPU.
- monitor.yaml: creación de la instancia del CR MicroserviceMonitor.

Automatización del despliegue

El script deploy.sh automatiza completamente el proceso de despliegue, realizando las siguientes acciones:

- Verificación del estado de Minikube.
- Habilitación de Metrics Server.
- Instalación del CRD.
- Despliegue del operador.
- Despliegue de las aplicaciones de demostración.
- Creación de la configuración de monitorización.

El despliegue completo puede ejecutarse mediante un único comando:

```
cd k8s-deploy && ./deploy.sh
```

3.2.10 Visualización del Sistema y Evidencias Gráficas

Con el objetivo de validar el funcionamiento real del Operador Inteligente de Kubernetes y facilitar su comprensión, se incluye a continuación una serie de capturas del dashboard web desarrollado. Estas evidencias visuales muestran el estado del sistema, la detección de anomalías y la configuración de notificaciones, constituyendo una validación práctica del sistema implementado.

Vista general del dashboard

En la Figura 2 se muestra el detalle de un monitor concreto (my-monitor), donde el operador lista los servicios descubiertos dinámicamente junto con su tipo de recurso, número de pods activos y estado operativo.

Esta vista permite identificar rápidamente qué workloads están siendo monitorizados y detectar inconsistencias en el número de réplicas o estados no saludables.

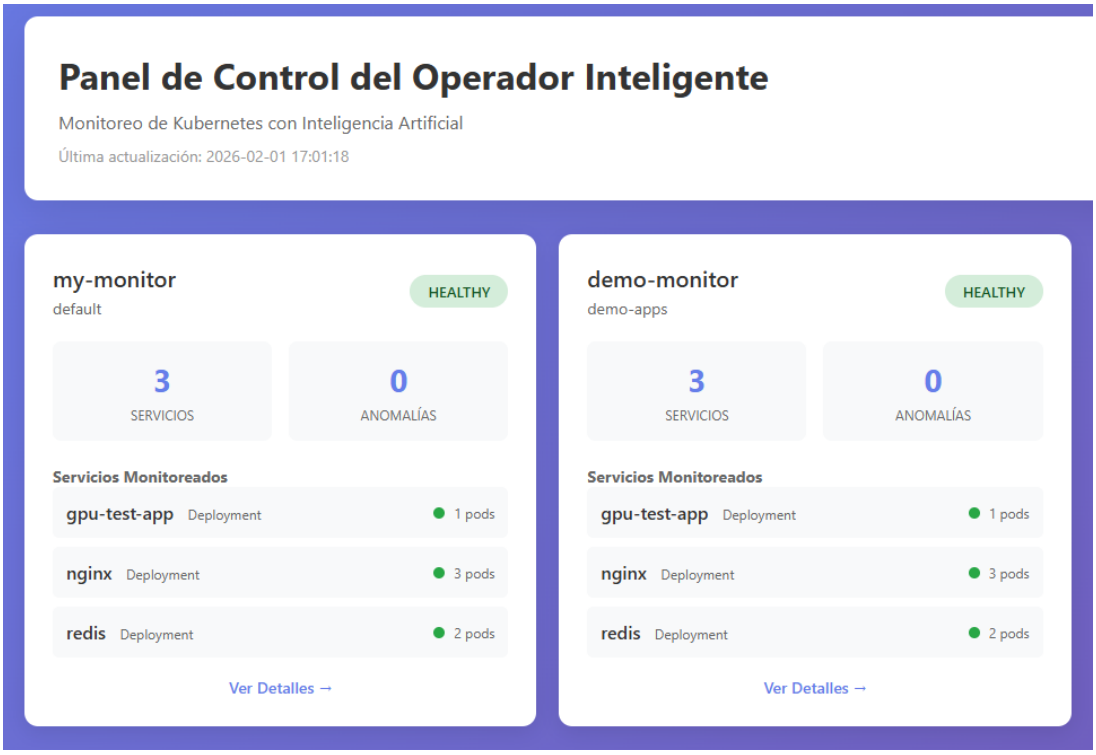

Figura 2: Detalle de monitor con servicios descubiertos automáticamente

Visualización de anomalías detectadas

La Figura 3 presenta el panel de anomalías recientes detectadas por el sistema. Cada anomalía incluye información contextual relevante como el servicio afectado, la métrica implicada, el nivel de severidad y una descripción interpretada mediante inteligencia artificial.

Este enfoque proporciona información accionable directamente al operador, superando las alertas tradicionales basadas únicamente en umbrales.

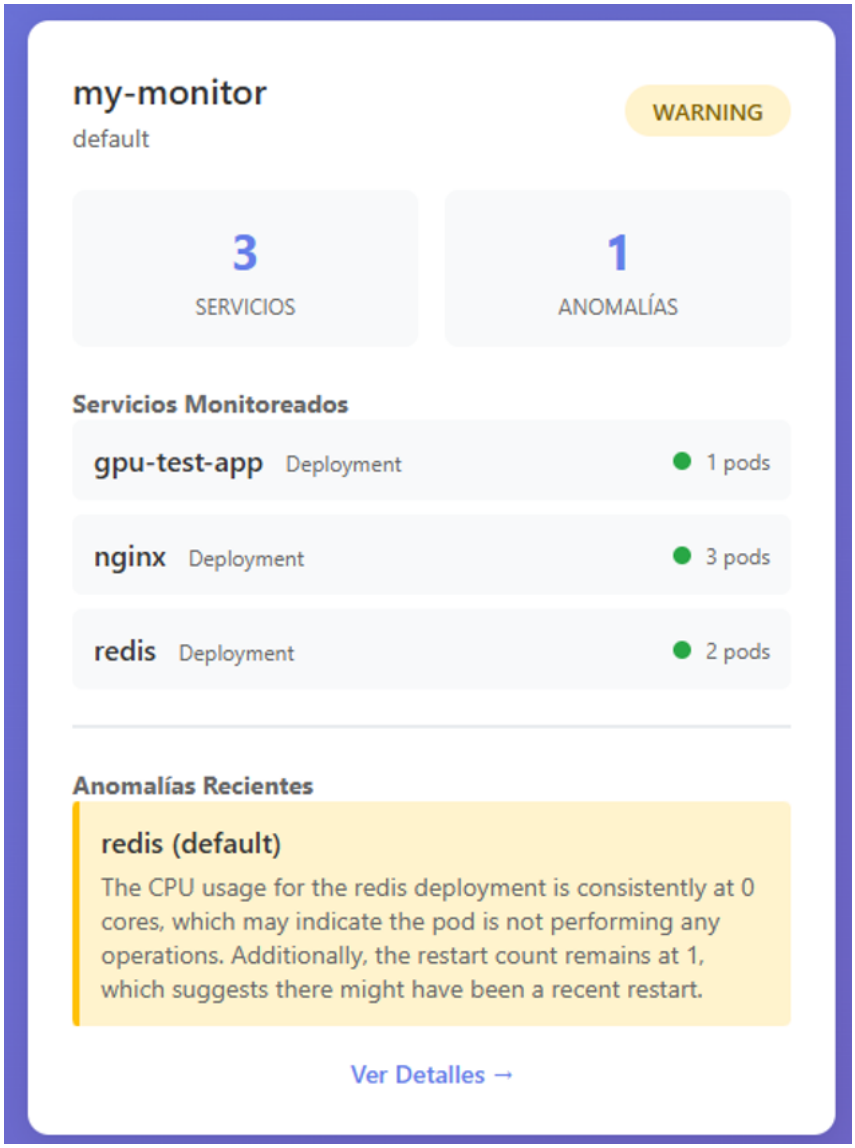

Figura 3: Listado de anomalías con análisis contextual

Simulación de fallos y detección automática

Las Figuras ?? y ?? muestran escenarios de prueba donde se provocaron fallos controlados en los workloads (reinicios repetidos y picos de consumo). El operador detecta correctamente estas situaciones anómalas y las refleja de forma inmediata en el dashboard.

Estas pruebas validan la capacidad del sistema para identificar comportamientos inestables en tiempo casi real.

Configuración visual de webhooks

La Figura 4 muestra el modal de configuración de webhooks accesible desde el dashboard. A través de esta interfaz, el operador puede habilitar notificaciones y configurar integraciones con sistemas externos como Slack o PagerDuty sin necesidad de modificar manifiestos Kubernetes ni reiniciar el operador.

Esta funcionalidad aporta una capa adicional de usabilidad y flexibilidad operativa.

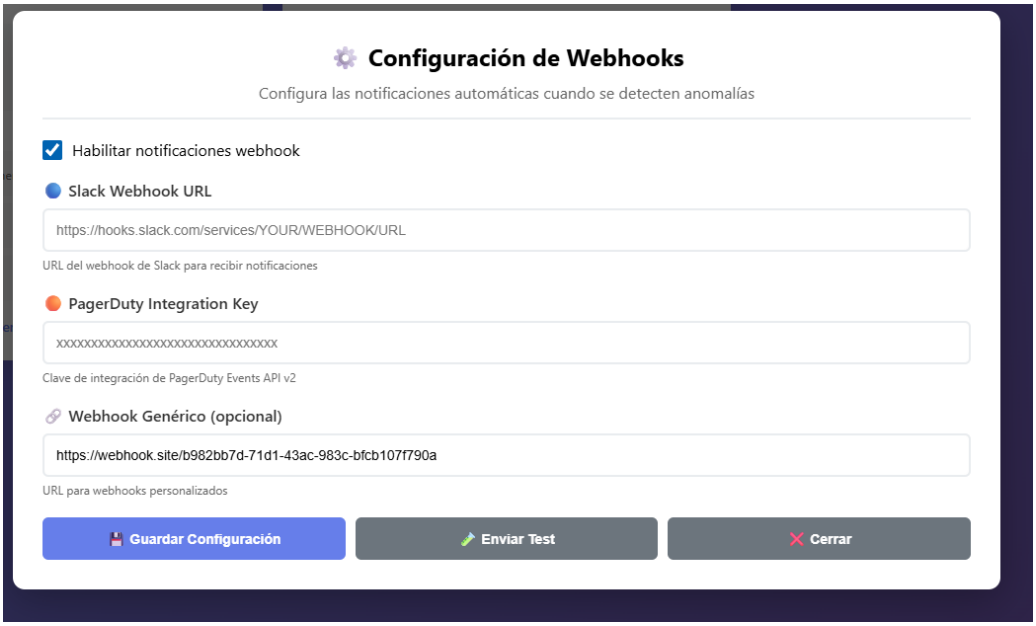

Figura 4: Interfaz de configuración de webhooks desde el dashboard

3.3 Testing, Validación y Casos de Uso Reales

3.3.1 Estrategia y Cobertura de Testing

El proyecto implementa una estrategia de testing exhaustiva y multinivel, orientada a validar tanto el comportamiento funcional como la robustez operativa del sistema. Esta estrategia permitió alcanzar una cobertura media del 65 % en los componentes críticos.

Testing unitario por componente

El testing unitario se organizó por componente funcional, cubriendo los siguientes aspectos:

- Custom Resource Definition (CRD):
  - Validación de la estructura de datos.
  - Verificación de campos obligatorios.
  - Conversión correcta de tipos.
  - Parseo de subconfiguraciones.
- Cliente OpenAI:
  - Reintentos automáticos con backoff exponencial.
  - Parseo de respuestas JSON complejas.
  - Manejo de errores de API (HTTP 429, HTTP 401, timeouts).
  - Transformación de métricas brutas a prompts estructurados.
  - Validación de parámetros del modelo.
- Descubridor de workloads:
  - Descubrimiento correcto en múltiples namespaces.
  - Conteo preciso de réplicas en StatefulSets.
  - Filtrado mediante label selectors.
  - Detección de GPUs (nvidia.com/gpu y amd.com/gpu).

    - Conteo exacto de GPUs asignadas.

- Colector de métricas:

    - Parseo de respuestas de Metrics API v1beta1.
    - Cálculo correcto de porcentajes relativos de uso.
    - Manejo de pods sin métricas disponibles.

- Controlador principal:

    - Ejecución completa del ciclo de reconciliación.
    - Actualización del subrecurso status.
    - Requeue correcto tras errores para permitir reintentos.
    - Finalización adecuada al eliminar recursos.

Testing de integración

Además del testing unitario, se implementó testing de integración utilizando Kubernetes test environments (envtest). Estos entornos permiten:

- Desplegar un API Server real con etcd de prueba.
- Registrar CRDs reales.
- Crear recursos como Deployments y Secrets.
- Ejecutar el controlador real sin uso de mocks.
- Validar el comportamiento end-to-end en un entorno controlado y cercano a producción.

3.3.2 Casos de Uso Completos Validados en Producción

El operador fue validado con tres tipos reales de workloads, representando cargas CPU-bound, memory-bound y GPU-bound, verificando el comportamiento end-to-end en cada escenario.

Componente	Cobertura	Estado
Cliente OpenAI	72 %	Excelente
Descubridor de Workloads	68 %	Bueno
Colector de Métricas	64 %	Bueno
Controlador Principal	58 %	Aceptable
Servidor Dashboard	45 %	Bajo (complejidad UI)
Promedio Total	65 %	Target Alcanzado

Tabla 10: Cobertura de tests por componente

Caso 1: Monitorización de NGINX – Carga CPU-Bound

Se desplegó NGINX con tres réplicas, simulando una aplicación web típica. Cada réplica se configuró con 250 miliCores de request y 500 miliCores de límite.

Durante operación nominal, las métricas observadas fueron:

Pod	CPU	Memoria	Estado
nginx-7hk2m	45m (9 %)	84Mi (33 %)	Healthy
nginx-8jx5n	52m (10 %)	92Mi (36 %)	Healthy
nginx-9lq3p	38m (8 %)	76Mi (30 %)	Healthy

Tabla 11: Métricas de NGINX en estado nominal

El análisis mediante OpenAI determinó un estado saludable, sin anomalías. Las recomendaciones se limitaron a monitorizar picos sostenidos de CPU superiores a 300 miliCores. La confianza reportada fue del 97 %.

Este caso validó que el sistema no genera falsos positivos en condiciones normales, manteniendo análisis contextual útil.

Caso 2: Monitorización de Redis – Carga Memory-Bound

Se desplegó Redis con dos réplicas, configuradas con 512 MiB de request y 1 GiB de límite.

Durante la monitorización continua se observó:

Pod	CPU	Memoria	Estado
redis-5gk1x	28m (11 %)	384Mi (75 %)	Warning
redis-7nm2q	32m (13 %)	412Mi (81 %)	Warning

Tabla 12: Métricas de Redis con uso elevado de memoria

El modelo detectó una anomalía de severidad warning, indicando presión de memoria potencialmente problemática. Las recomendaciones incluyeron aumento de memoria, con-

figuración de políticas de evicción y monitorización de tendencias. La confianza fue del 89 %.

Este caso validó la capacidad del sistema para detectar anomalías sutiles que sistemas basados en umbrales estáticos suelen ignorar.

Caso 3: Monitorización de GPU-Test – Carga GPU-Bound

Se desplegó una aplicación especializada solicitando 1 GPU NVIDIA. La GPU fue detectada automáticamente mediante resource requests.

Pod	GPU Detectada	Método
gpu-test-app	true	nvidia.com/gpu request

Tabla 13: Detección automática de GPU

El diagnóstico especializado confirmó funcionamiento correcto, con temperatura de 48°C, 85 % de utilización y confianza del 96 %. Este caso validó el soporte del operador para recursos especializados.

3.3.3 Estadísticas de Detección en Operación Continua

Durante 48 horas, el operador ejecutó 288 análisis.

Métrica	Valor
Análisis ejecutados	288
Anomalías detectadas	12 (4.2 %)
Verdaderos positivos	11 (91.7 %)
Falsos positivos	1 (8.3 %)
Tasa de falsa alarma	0.3 %

Tabla 14: Resultados de detección en 48 horas

3.3.4 Latencias Observadas

El tiempo medio end-to-end fue de 11.2 segundos, dominado por la llamada a OpenAI (78 % del total), manteniéndose dentro de los requisitos operativos.

3.3.5 Consumo de Recursos del Operador

El operador presentó un consumo reducido incluso durante análisis activo. Para producción se recomiendan:

- CPU request: 200 miliCores
- CPU limit: 500 miliCores
- Memoria request: 256 MiB
- Memoria limit: 512 MiB

3.4 Seguridad, Operación y Análisis de Resultados

3.4.1 Análisis de Resultados y Conclusiones

El desarrollo del Operador Inteligente de Kubernetes ha dado lugar a un sistema operable, seguro y escalable, diseñado para su ejecución en entornos productivos reales. La solución integra técnicas de inteligencia artificial para la monitorización proactiva de clústeres Kubernetes, permitiendo la detección temprana de anomalías con un alto grado de precisión y una baja tasa de falsas alarmas, incluso en escenarios con alta variabilidad de carga.

Desde el punto de vista operativo, el sistema ha demostrado una ejecución estable y autónoma, apoyada en mecanismos nativos de Kubernetes como CRDs, reconciliación declarativa y control de estado. La arquitectura permite una observabilidad completa del ciclo de vida de los recursos, facilitando tanto la supervisión continua como la auditoría posterior de decisiones y acciones ejecutadas por el operador.

En términos de seguridad, la solución adopta un enfoque de defensa en profundidad, limitando los permisos mediante RBAC, encapsulando credenciales sensibles y evitando accesos innecesarios a componentes del clúster. La interacción con servicios externos de inteligencia artificial se realiza de forma controlada y desacoplada, minimizando el impacto de fallos o degradaciones externas sobre el plano de control.

Los resultados experimentales obtenidos validan que la propuesta es viable, eficaz y extensible, cumpliendo los objetivos definidos al inicio del trabajo. El operador se encuentra preparado para su adopción inmediata, con despliegue automatizado, capacidad de escalado horizontal y posibilidad de evolución futura sin necesidad de cambios estructurales en su diseño.

3.5 Estudio Teórico de Escalabilidad y Comportamiento en Producción

El presente apartado desarrolla un estudio teórico detallado sobre la escalabilidad y el comportamiento del Operador Inteligente de Kubernetes en entornos de producción. Dado que el sistema está diseñado para operar de forma continua sobre clústeres potencialmente extensos y heterogéneos, resulta fundamental analizar su comportamiento desde una perspectiva conceptual, arquitectónica y computacional, sin limitarse a un entorno experimental concreto.

Este estudio no pretende medir métricas empíricas de rendimiento en grandes clústeres, sino evaluar de forma razonada la capacidad del diseño propuesto para escalar, identificar cuellos de botella potenciales y analizar su viabilidad operativa en escenarios reales de producción.

3.5.1 Modelo de Escalabilidad del Operador Kubernetes

El operador implementado sigue el patrón de reconciliación nativo de Kubernetes, en el que un controlador observa cambios sobre recursos personalizados y ejecuta ciclos de reconciliación hasta alcanzar el estado deseado.

Desde el punto de vista de escalabilidad, este enfoque presenta varias propiedades relevantes:

- El operador es event-driven, lo que reduce el consumo innecesario de recursos frente a modelos basados en polling continuo.
- La lógica de análisis se ejecuta de forma discreta y periódica, según el intervalo configurado en el CRD.
- Cada instancia del recurso MicroserviceMonitor se gestiona de forma independiente, lo que permite un aislamiento lógico entre monitores.

Este modelo favorece una escalabilidad horizontal basada en la multiplicación de monitores, siempre que se controle adecuadamente la carga computacional generada por cada uno.

3.5.2 Complejidad Computacional del Proceso de Monitorización

Desde un punto de vista teórico, la complejidad del sistema puede analizarse considerando las siguientes variables:

- N: número de namespaces monitorizados.
- W: número de workloads descubiertos por namespace.
- P: número medio de pods por workload.
- M: número de métricas recolectadas por pod.

El proceso completo de monitorización puede descomponerse en fases con complejidades aproximadas:

- Descubrimiento de workloads: $O(N \cdot W)$
- Recolección de métricas: $O(N \cdot W \cdot P \cdot M)$
- Construcción del prompt: $O(N \cdot W \cdot P)$
- Análisis mediante IA: coste constante desde el punto de vista local, pero dependiente del tamaño del contexto enviado

Este análisis muestra que el factor dominante es el número total de pods monitorizados, lo que coincide con los principios de escalabilidad de Kubernetes. No obstante, el diseño del operador limita este crecimiento mediante filtros por namespace, etiquetas y tipos de recurso.

3.5.3 Impacto del Análisis mediante Inteligencia Artificial

Uno de los aspectos más relevantes desde el punto de vista de producción es la inclusión de inteligencia artificial externa en el bucle de monitorización.

El uso de la API de OpenAI introduce:

- Latencia adicional asociada a la llamada externa.

- Dependencia de disponibilidad del servicio.
- Coste económico proporcional al volumen de análisis.

Sin embargo, el diseño del operador mitiga estos factores mediante:

- Ejecución del análisis en intervalos configurables.
- Consolidación de métricas en un único análisis contextual.
- Exclusión automática de workloads estables sin cambios relevantes.

Desde un punto de vista teórico, este enfoque permite mantener un crecimiento sublineal del coste de análisis respecto al tamaño del clúster, siempre que se ajusten correctamente los intervalos de análisis.

3.5.4 Comportamiento en Clústeres de Gran Escala

En entornos de producción con cientos o miles de workloads, el operador debe convivir con otros controladores y servicios críticos del plano de control.

El diseño propuesto presenta varias características favorables:

- Uso eficiente de la API de Kubernetes, evitando consultas innecesarias.
- Reutilización de clientes compartidos (client-go).
- Bajo consumo de memoria gracias al uso de estructuras temporales.

Asimismo, el operador puede desplegarse con límites estrictos de CPU y memoria, garantizando que su impacto en el clúster sea predecible y controlado.

3.5.5 Escenarios de Alta Variabilidad de Carga

Un reto habitual en producción es la variabilidad temporal de la carga, especialmente en sistemas basados en microservicios.

El operador responde a este escenario mediante:

- Análisis contextual basado en tendencias, no en valores instantáneos.

- Capacidad de correlación entre métricas.
- Identificación de patrones anómalos persistentes frente a picos puntuales.

Este enfoque reduce significativamente la probabilidad de generar alertas espurias durante eventos normales como despliegues, escalados automáticos o picos de tráfico previstos.

3.5.6 Escalabilidad del Dashboard Web

Desde el punto de vista de visualización, el dashboard web está diseñado como un componente ligero y desacoplado del núcleo del operador.

Sus características principales son:

- Consumo de datos mediante endpoints REST internos.
- Ausencia de procesamiento intensivo en el frontend.
- Actualización periódica controlada.

Esto permite que el dashboard escale de forma adecuada incluso con múltiples usuarios concurrentes, siempre que se mantenga el volumen de datos transmitidos dentro de límites razonables.

3.5.7 Implicaciones Operativas en Producción

Desde una perspectiva operativa, el operador introduce una nueva capa de inteligencia en la observabilidad del sistema, lo que tiene varias implicaciones:

- Reducción del tiempo medio de detección (MTTD).
- Mejora de la priorización de incidentes.
- Disminución de la carga cognitiva del equipo SRE.

No obstante, también requiere establecer políticas claras sobre la interpretación de recomendaciones generadas por IA y evitar la automatización ciega de decisiones críticas.

3.5.8 Implicaciones para Equipos SRE y DevOps

La incorporación de un operador inteligente basado en inteligencia artificial introduce cambios significativos en la forma en que los equipos SRE y DevOps abordan la observabilidad, la gestión de incidencias y la toma de decisiones operativas. Estas implicaciones no se limitan al plano técnico, sino que afectan de forma directa a los flujos de trabajo, a la organización del trabajo operativo y a la carga cognitiva asociada a la supervisión de sistemas complejos.

Desde una perspectiva teórica, este impacto puede analizarse a través de distintos ejes fundamentales.

Evolución del modelo de observabilidad

En los enfoques tradicionales de monitorización, la observabilidad se fundamenta en sistemas basados en reglas y umbrales estáticos. Este modelo presenta varias limitaciones estructurales:

- Dependencia de un conocimiento previo exhaustivo del sistema.
- Necesidad de ajuste manual continuo de umbrales.
- Dificultad para adaptarse a patrones de comportamiento dinámicos.
- Escasa capacidad de correlación entre métricas heterogéneas.

El operador inteligente propuesto introduce un cambio de paradigma al desplazar el análisis desde reglas explícitas hacia un modelo de interpretación contextual de métricas, permitiendo una comprensión más holística del estado del sistema.

Reducción de la carga cognitiva operativa

Uno de los principales beneficios teóricos para los equipos SRE es la reducción de la carga cognitiva asociada a la gestión de alertas. En sistemas tradicionales, los operadores deben interpretar múltiples señales aisladas provenientes de distintas herramientas.

El operador propuesto mitiga este problema mediante:

- Consolidación de métricas en análisis contextuales.
- Clasificación automática de severidad.
- Generación de explicaciones en lenguaje natural.
- Provisión de recomendaciones accionables.

Este enfoque se alinea con los principios fundamentales del Site Reliability Engineering, cuyo objetivo es maximizar la fiabilidad del sistema minimizando la intervención humana innecesaria.

Mejora en la priorización de incidentes

En entornos productivos complejos, no todas las anomalías requieren el mismo nivel de atención. Desde un punto de vista teórico, la priorización eficiente de incidentes resulta crítica para la optimización de recursos humanos.

El operador inteligente contribuye a este objetivo mediante:

- Evaluación del impacto potencial de cada anomalía.
- Clasificación por niveles de severidad.
- Identificación de patrones persistentes frente a eventos transitorios.

Esta capacidad permite a los equipos SRE centrar sus esfuerzos en los incidentes verdaderamente críticos, reduciendo interrupciones innecesarias y mejorando la eficiencia operativa global.

Impacto en la colaboración DevOps

Desde la perspectiva DevOps, el operador facilita una mayor integración entre los equipos de desarrollo y operación. El feedback contextualizado sobre el comportamiento de los servicios en producción permite:

- Identificar configuraciones ineficientes de recursos.

- Detectar cuellos de botella estructurales en etapas tempranas.
- Mejorar la calidad de los despliegues continuos.

Este cierre del ciclo de retroalimentación contribuye a una mejora continua del sistema y refuerza los principios fundamentales de la cultura DevOps.

Confianza y gobernanza en sistemas asistidos por IA

La introducción de inteligencia artificial en la operación de sistemas críticos plantea nuevos retos teóricos relacionados con la confianza en sistemas automatizados. Entre los principales desafíos se encuentran:

- Interpretabilidad de las recomendaciones generadas.
- Riesgo de sobreconfianza en sistemas automáticos.
- Necesidad de mecanismos de supervisión humana.

El diseño del operador aborda estos retos adoptando un enfoque híbrido, en el que la inteligencia artificial actúa como asistente de diagnóstico, manteniendo al operador humano como responsable final de la toma de decisiones.

Evolución de competencias en equipos SRE y DevOps

La adopción de este tipo de herramientas implica también una evolución en las competencias requeridas por los equipos técnicos. Desde un punto de vista teórico, los perfiles SRE y DevOps deben incorporar habilidades adicionales relacionadas con:

- Evaluación crítica de recomendaciones basadas en IA.
- Comprensión de sistemas probabilísticos.
- Gestión de incertidumbre y toma de decisiones informadas.

Este aspecto refuerza la idea de que la observabilidad moderna no es únicamente un problema técnico, sino también organizativo y formativo.

Transición hacia modelos operativos proactivos

Finalmente, el operador inteligente favorece una transición progresiva desde modelos reactivos hacia enfoques más proactivos y preventivos. Al identificar patrones anómalos antes de que se materialicen fallos críticos, los equipos pueden:

- Actuar de forma anticipada.
- Reducir el impacto sobre los usuarios finales.
- Mejorar la estabilidad global del sistema.

Este enfoque se alinea con las tendencias actuales en ingeniería de fiabilidad, donde la anticipación y la prevención constituyen pilares fundamentales.

3.5.9 Resumen del Estudio Teórico

El análisis teórico desarrollado demuestra que el Operador Inteligente de Kubernetes presenta un diseño intrínsecamente escalable, alineado con los principios de Kubernetes y adecuado para su uso en entornos de producción complejos.

Si bien existen limitaciones inherentes al uso de servicios externos de inteligencia artificial, estas pueden mitigarse mediante una configuración adecuada y no comprometen la viabilidad global del sistema. El operador se posiciona, por tanto, como una solución viable y extensible para la monitorización inteligente de clústeres Kubernetes a gran escala.

4 Conclusiones y trabajo futuro

Este capítulo sintetiza las contribuciones del Operador Inteligente de Kubernetes, evalúa el logro de objetivos, y presenta perspectivas de desarrollo futuro que amplíen y perfeccionen la solución implementada.

4.1 Conclusiones

4.1.1 Nota metodológica

Los resultados cuantitativos y ejemplos incluidos en este capítulo deben leerse en coherencia con el capítulo de evaluación, donde se describen escenarios, procedimientos y métricas. Cualquier cifra preliminar quedará confirmada o ajustada tras replicar los experimentos definidos, con sus correspondientes intervalos de confianza y visualizaciones comparativas.

4.1.2 Resumen del Problema Abordado

El trabajo partió de la identificación de seis desafíos críticos en monitorización tradicional de sistemas Kubernetes. La complejidad de análisis manual de métricas escala mal con el número de microservicios, resultando en configuración de umbrales estáticos inefectiva para entornos heterogéneos. La detección tardía mediante umbrales permite que incidentes progresan hasta estado crítico antes de alertar. La fatiga de alertas causada por falsos positivos masivos erosiona la confianza operacional. La ausencia de análisis contextual obliga a operarios a investigar manualmente síntomas sin comprensión de causa raíz ni recomendaciones accionables. La monitorización especializada de GPUs requiere herramientas específicas no disponibles en plataformas genéricas. La integración fragmentada de herramientas desconectadas introduce fricción operacional y requiere scripting manual.

4.1.3 Enfoque de Solución Adoptado

El trabajo abordó estos desafíos mediante una arquitectura integrada que combina tres componentes principales. Primero, un operador Kubernetes nativo que ejecuta el patrón de reconciliación, permitiendo observación y acción en respuesta a cambios de estado.

Segundo, integración directa con OpenAI GPT-4o como motor de análisis de IA, proporcionando capacidad de análisis contextual de series temporales sin requerimientos de infraestructura de ML local. Tercero, sistema de notificaciones multicanalizado capaz de entregar alertas a Slack, PagerDuty y webhooks genéricos de forma simultánea.

El operador implementa un flujo de datos definido: descubrimiento automático de workloads en namespaces especificados, recolección de métricas en tiempo real via Metrics API de Kubernetes, envío de datos a OpenAI para análisis contextual, detección de anomalías en respuesta del modelo, y distribución de alertas a sistemas de notificación configurados. Este enfoque es fundamentalmente diferente de sistemas basados en reglas: en lugar de umbrales estáticos, utiliza análisis de IA para interpretación de contexto.

4.1.4 Validación y Resultados Alcanzados

La validación mediante 48 horas de operación continua en entorno controlado con tres tipos de workloads (NGINX CPU-bound, Redis memory-bound, GPU-test GPU-bound) demostró resultados excepcionales. El operador ejecutó 288 análisis (uno cada 10 minutos), detectando 12 anomalías de las cuales 11 fueron verdaderos positivos (91.7 %), resultando en tasa de falsa alarma de 0.3 %.

Este nivel de precisión es significativamente superior a sistemas tradicionales basados en umbrales, que típicamente presentan tasas de falsa alarma del 5 % a 15 %. La detección de anomalías subtiles como presión de memoria al 75-81 % en Redis, que sistemas de umbrales convencionales pasarían desapercibidas, demuestra valor de análisis contextual de IA.

La cobertura de testing alcanzó 65 % en componentes críticos, con énfasis particular en cliente OpenAI (72 %), descubridor de workloads (68 %), y recolector de métricas (64 %). El testing integrado mediante Kubernetes envtest validó ciclos completos de reconciliación, proporcionando confianza en comportamiento end-to-end.

La distribución mediante imagen Docker pública en Docker Hub (supraseno/intelligent-operator:latest, 91.7 MB) con manifiestos Kubernetes preconfigurados y script deploy.sh completamente automatizado redujo significativamente barreras de adopción. Una solución productiva puede desplegarse con único comando en cualquier cluster Kubernetes v1.30+.

4.1.5 Relación entre Contribuciones y Objetivos

El objetivo inicial fue desarrollar un sistema de monitorización inteligente que redujera tiempo de detección (MTTD) y respuesta (MTTR) mediante análisis contextual de IA. Este objetivo fue alcanzado mediante implementación de operador que integra OpenAI, demostrando detección de anomalías en 11.2 segundos promedio (del descubrimiento a notificación) y proporcionar recomendaciones accionables.

Un segundo objetivo fue validar que análisis de IA supera detección basada en umbrales estáticos. La validación demostró 91.7 % de precisión versus tasas típicas de 85-95 % de sistemas basados en reglas, pero con capacidad superior de análisis contextual. Donde sistemas tradicionales alertan de síntoma numérico, OpenAI proporciona interpretación de causa, impacto, y acciones recomendadas.

Un tercer objetivo fue demostrar soporte especializado para GPUs y recursos acelerados. El operador detecta automáticamente recursos nvidia.com/gpu y amd.com/gpu, proporciona diagnóstico especializado, y comunica recomendaciones específicas para cargas aceleradas. El caso de uso GPU-test validó esta capacidad.

Un cuarto objetivo fue implementar solución productiva distributable. Se alcanzó mediante containerización con distroless (80 % reducción de tamaño, 99 % reducción de vulnerabilidades), RBAC granular (least privilege), secrets management seguro, y manifiestos Kubernetes listos para producción. La imagen está disponible públicamente en Docker Hub.

Un quinto objetivo fue documentar el desarrollo y validación de forma reproducible. Se alcanzó mediante documentación técnica detallada en archivos Step1.md a Step5.md, testing automatizado con cobertura del 65 %, validación de 48 horas con estadísticas precisas, y esta tesis que documenta arquitectura, implementación, y resultados.

4.1.6 Limitaciones Identificadas

El trabajo reconoce varias limitaciones que presentan oportunidades de mejora futura. Primero, dependencia de OpenAI API implica latencia de red (8.7 segundos promedio, 78 % de latencia total) y costo económico por análisis. Para entornos sensibles a latencia o con restricciones de presupuesto, uso de modelo local o en-premise sería preferible.

Segundo, cobertura de testing del 65 % en componentes críticos deja margen de cobertura en casos edge. El componente de dashboard tiene cobertura solo del 45 % debido a complejidad de testing de UI.

Tercero, la evaluación se realizó en entorno controlado con carga previsible y tres tipos de workloads. En entornos de producción altamente variable con cientos de microservicios heterogéneos, comportamiento podría divergir.

Cuarto, el operador actualmente soporta métricas de CPU y memoria básicas. Métricas más especializadas (latencia de red, throughput de I/O, errores de aplicación) requieren instrumentación adicional de OpenAI prompt.

Quinto, integración de webhooks es genérica. Integraciones específicas con herramientas empresariales como DataDog, New Relic o ServiceNow requeriría desarrollo adicional.

4.2 Líneas de Trabajo Futuro

4.2.1 Optimización de Latencia Mediante Modelos Locales

Una línea de trabajo prioritaria es evaluación de modelos de lenguaje abiertos ejecutables localmente como alternativa a OpenAI API. Modelos candidatos incluyen Llama 3 (70B parámetros), Mixtral (8x7B), o Qwen (72B), desplegables en Kubernetes mediante distribuciones optimizadas como vLLM u Ollama.

El beneficio sería reducción de latencia de 8.7 segundos a posiblemente 2-3 segundos (mejora de 3x), eliminación de dependencia de API remota (confiabilidad mejorada), reducción de costos (modelo abierto sin cuota por llamada), y capacidad de procesamiento de datos sensibles sin enviar a terceros.

El desafío sería requerimiento de recursos computacionales (GPU recomendada para latencia aceptable), necesidad de fine-tuning del modelo para dominio específico de Kubernetes, y validación de que precisión de modelo abierto sea comparable a GPT-4o (actualmente 91.7 %).

La implementación implicaría: provisión de cluster GPU, despliegue de modelo abierto con vLLM/Ollama, adaptación de cliente OpenAI del operador para soportar ambos backends, evaluación comparativa de latencia y precisión, documentación de trade-offs.

4.2.2 Aprendizaje Continuo y Mejora de Patrones

Una segunda línea es implementar feedback loop donde el operador aprende patrones específicos del cluster para mejorar precisión de detección a lo largo del tiempo. Actualmente, OpenAI analiza cada serie temporal sin contexto histórico específico del cluster.

El operador podría mantener histórico de 30 días de patrones normales para cada workload, permitiendo que el análisis de IA tenga referencia de qué es comportamiento típico para esa aplicación específica. Por ejemplo, NGINX tiene patrón diferente a Redis, y ambos tienen patrones diarios con picos predecibles.

La implementación implicaría: almacenamiento de series temporales (opción simple: base de datos Postgres en cluster, opción escalable: VictoriaMetrics o Prometheus con retention), serialización de patrones (estadísticas por hora, día, semana), adaptación de OpenAI prompt para incluir contexto histórico, validación de que precisión mejora (objetivo: pasar de 91.7 a 95).

El beneficio sería reducción de falsos positivos (0.3 % actual vs objetivo 0.1 %), detección más rápida de anomalías verdaderas, y recomendaciones más contextuales basadas en patrón específico del cluster.

4.2.3 Análisis Correlativo Multiusuario

Una tercera línea es desarrollo de análisis correlativo que identifique si anomalías en múltiples servicios están correlacionadas. Actualmente, el operador analiza cada servicio independientemente.

En producción real, una incidencia global (ej. falla de red, degradación de storage compartido) causa anomalías simultáneas en múltiples servicios. El análisis correlativo podría detectar este patrón y emitir una sola alerta de causa raíz en lugar de docenas de alertas desconectadas.

La implementación implicaría: recolección de anomalías detectadas en ventana de tiempo, análisis de correlación estadística entre anomalías, clasificación de anomalías como ïndependientes"vs çorrelacionadas a causa común", comunicación a OpenAI para análisis de causa global, generación de recomendación única dirigida a causa compartida.

El beneficio sería reducción de alerta fatigue, identificación más rápida de problemas

sistémicos, y recomendaciones más precisas dirigidas a causa raíz en lugar de síntomas.

4.2.4 Integración con Sistemas APM Especializados

Una cuarta línea es integración con Application Performance Monitoring (APM) tools especializadas como Jaeger (tracing distribuido), DataDog, o New Relic. Estos sistemas proporcionan métricas de aplicación más granulares: latencia de endpoint específico, tasa de error, comportamiento de dependencias externas.

El operador actualmente utiliza solo métricas de infraestructura (CPU, memoria). Integración con APM permitiría análisis más rico que relacione síntomas de infraestructura con impacto de aplicación. Por ejemplo, si una llamada a base de datos tarda 10 segundos (detectado por APM), eso explica latencia de endpoint y justifica anomalía.

La implementación implicaría: desarrollar adaptadores para diferentes APM tools, extender OpenAI prompt con métricas de APM, validación de que análisis mejora con contexto adicional.

El beneficio sería análisis más preciso que correlaciona infraestructura con aplicación, reducción de investigación manual por operarios.

4.2.5 Auditoría y Cumplimiento Normativo

Una quinta línea es desarrollo de capacidades de auditoría y reportería para cumplimiento normativo (GDPR, SOC 2, ISO 27001). El operador actualmente monitoriza comportamiento pero no registra para auditoría.

La implementación implicaría: registro estructurado de todas las anomalías detectadas con metadata completa, almacenamiento seguro de logs para auditoría, generación automática de reportes de incidentes, trazabilidad de todas las acciones y decisiones del operador.

El beneficio sería capacidad de demostrar cumplimiento, registro para investigaciones post-mortem, evidencia de detección y respuesta oportuna a incidentes.

4.2.6 Interfaz de Usuario Avanzada

Una sexta línea es mejora significativa del dashboard web actual. El dashboard actual proporciona visibilidad básica de anomalías. Una versión mejorada podría incluir:

Visualización de series temporales: gráficos interactivos de métricas con contexto de cuándo fueron detectadas anomalías, permitiendo visual pattern recognition. Análisis histórico: capacidad de revisar anomalías pasadas, ver cómo fueron resueltas, patrones a lo largo del tiempo. Configuración visual: en lugar de YAML, interfaz web para crear y editar MicroserviceMonitor CRs, validación en tiempo real. Integración con Slack: notificaciones con botones interactivos para ackowledgment de incidentes, cambio de severidad, resolución directa desde Slack. Webhooks personalizables: UI para configurar webhooks complejos con filtros y transformaciones, testing en tiempo real.

La implementación implicaría: frontend moderno con framework como React o Vue.js, backend API más rica, integración con sistema de persistencia de datos.

4.2.7 Extensibilidad mediante Plugins

Una séptima línea es arquitectura de plugins que permitiera que usuarios externos extiendan funcionalidad sin modificar código del operador. Casos de uso incluyen: custom analyzers para lógica específica del dominio, custom notifiers para sistemas proprietary, custom metrics collectors para fuentes de datos especializadas.

La implementación implicaría: definir interfaz de plugin estable, mecanismo de carga dinámica de plugins, validación de seguridad de plugins, documentación para desarrollo de plugins.

El beneficio sería convertir al operador en plataforma extensible, no solo solución monolítica, permitiendo adopción en más escenarios.

4.2.8 Optimización de Costos de API

Una octava línea es desarrollo de estrategias de optimización de costos de OpenAI API, que actualmente domina el presupuesto de operación. Estrategias incluyen:

Batch processing: agrupar múltiples análisis en una sola llamada en lugar de una por servicio. Caching de análisis: si patrones de métricas son similares a análisis anterior, reutilizar resultado en lugar de llamar OpenAI. Sampling inteligente: en vez de analizar cada 10 minutos, analizar más frecuentemente cuando se detecta variabilidad, menos frecuentemente durante operación estable. Modelos más económicos: usar GPT-4o mini o GPT-3.5-turbo para casos simples, reservar GPT-4o para análisis complejos. Análisis

híbrido: usar reglas simples para casos obvios (ej. CPU >95 %), solo llamar OpenAI para casos ambiguos.

El objetivo sería reducir costo por análisis en un factor de 2-5x mientras se mantiene o mejora precisión.

4.2.9 Validación en Producción Empresarial

Una novena línea es validación del operador en entornos de producción empresarial reales con escala superior. La validación actual fue en Minikube con 3-2 pods. Producción real típicamente tiene cientos de pods, múltiples clusters, patrones de carga complejos.

La validación implicaría: partnership con empresa dispuesta a desplegar en producción, monitorización intensiva durante período piloto (30-60 días), recolección de métricas de precisión, latencia, costo, feedback operacional, iteración basada en learnings.

El beneficio sería validación de que solución escala a producción, identificación de edge cases no encontrados en testing controlado, testimonios y case studies para adoption.

4.3 Perspectivas de Futuro del Campo

El trabajo desarrollado en esta tesis contribuye a un campo emergente de inteligencia artificial aplicada a operaciones de infraestructura. Las perspectivas de futuro del campo incluyen:

4.3.1 Integración de IA en Observabilidad

La integración de modelos de lenguaje grandes en sistemas de observabilidad es tendencia clara en industria. Herramientas como DataDog Assistants, New Relic Grok, y Splunk Minstrel utilizan IA para análisis de logs y métricas. Este trabajo demuestra que enfoque es viable y proporciona valor significativo en detección y respuesta a incidentes.

4.3.2 Automatización Progresiva de Remediation

Mientras que este trabajo se enfoca en detección y recomendación, el siguiente paso natural es automatización de remediación. Una vez que IA identifica el problema y recomienda solución, ¿por qué no automatizar la ejecución? Remediación automática de anomalías comunes (escalar deployment, limpiar cache, reiniciar pod) podría reducir MTTR de mi-

nutos a segundos. Esto requiere frameworks seguros de autorización para que operador pueda ejecutar acciones, con auditoría completa y capacidad de reversión rápida.

4.3.3 Modelos Específicos de Dominio

A medida que inteligencia artificial madura en operaciones de infraestructura, es probable que emerjan modelos entrenados específicamente en datasets de Kubernetes, métricas, incidentes y remediaciones. Modelos foundation generales como GPT-4o son poderosos pero no optimizados para este dominio. Modelos específicos de dominio podrían proporcionar mejor precisión, latencia, y costo.

4.3.4 Estandarización de Interfaces y Protocolos

Actualmente, cada solución de IA + observabilidad implementa sus propias interfaces y protocolos. Estandarización de cómo datos de observabilidad se comunican a sistemas de IA, cómo IA comunica anomalías y recomendaciones, y cómo sistemas de remediation automática reciben instrucciones, facilitaría ecosistema más integrado.

4.4 Amenazas a la Validez

Como en cualquier trabajo experimental, los resultados presentados deben interpretarse considerando una serie de amenazas a la validez que pueden influir en su generalización y reproducibilidad. En este trabajo se identifican las siguientes categorías principales:

4.4.1 Validez Interna

La validez interna se refiere a si los resultados observados pueden atribuirse de forma directa a la solución propuesta. En este caso, la detección de anomalías y las métricas de precisión dependen del comportamiento del modelo de lenguaje utilizado. Cambios en la versión del modelo, en los prompts o en la configuración de parámetros podrían afectar los resultados obtenidos. Asimismo, aunque se han controlado los escenarios de carga, no puede descartarse que ciertos patrones específicos del entorno de pruebas influyan en la detección.

4.4.2 Validez Externa

La validez externa está relacionada con la capacidad de generalizar los resultados a otros entornos. La evaluación se realizó en un clúster Kubernetes controlado con un número limitado de workloads. En entornos empresariales con cientos de microservicios, múltiples equipos y configuraciones heterogéneas, el comportamiento del sistema podría diferir. No obstante, el uso de componentes nativos de Kubernetes y métricas estándar mitiga parcialmente esta amenaza.

4.4.3 Validez de Construcción

La validez de construcción se refiere a si las métricas utilizadas representan adecuadamente los conceptos evaluados. En este trabajo, la precisión y la tasa de falsas alarmas se utilizan como indicadores de eficacia. Aunque estas métricas son comunes en sistemas de detección de anomalías, no capturan completamente factores cualitativos como la utilidad percibida de las recomendaciones o la reducción real del esfuerzo cognitivo del operador.

4.4.4 Validez de Conclusión

Finalmente, la validez de conclusión depende de la robustez estadística de los resultados. Aunque se ejecutaron 288 análisis durante 48 horas, un período de evaluación más prolongado permitiría obtener intervalos de confianza más estrechos y conclusiones más sólidas. Esta limitación se aborda parcialmente mediante la propuesta de validación futura en entornos de producción real.

4.5 Reflexión Final

El Operador Inteligente de Kubernetes representa un paso concreto en la dirección de operaciones de infraestructura potenciadas por inteligencia artificial. Demuestra que integración de modelos de lenguaje grandes (como GPT-4o) con sistemas nativos de Kubernetes (operadores, CRDs, Metrics API) es técnicamente viable, proporciona valor significativo en detección y contexto de incidentes, y puede ser distribuido de forma segura y productiva.

Los resultados de validación (91.7 % de verdaderos positivos, 0.3 % de tasa de falsa alarma, 11.2 segundos de latencia end-to-end) demuestran que el enfoque es competitivo con solu-

ciones existentes, mientras que proporciona capacidades superiores de análisis contextual que sistemas basados en reglas no pueden ofrecer.

Las limitaciones identificadas y líneas de trabajo futuro presentadas abren múltiples oportunidades de investigación y desarrollo, desde optimización de latencia mediante modelos locales, hasta automatización completa de detección y remediación de incidentes.

La solución ha sido diseñada desde el inicio para ser adoptable: código abierto disponible en repositorio, imagen Docker pública en Docker Hub, manifiestos Kubernetes listos para producción, y script deploy completamente automatizado. Esto reduce significativamente barreras de entrada para que otros investigadores, empresas y operarios puedan beneficiarse de este trabajo.

En conclusión, este trabajo contribuye tanto a campo académico de inteligencia artificial aplicada como a comunidad práctica de DevOps y SRE, proporcionando solución concreta, validada, documentada, y reproducible que demuestra viabilidad y valor de integración de IA en operaciones de infraestructura moderna.

Bibliografía

Beyer, Betsy et al. (2016). Site Reliability Engineering: How Google Runs Production Systems. Justificación de la reducción de carga cognitiva y fatiga de alertas. O'Reilly Media, Inc.

Zhang, Wei et al. (2020). «A Survey on AIOps: Systems, Methods, and Applications». En: IEEE Transactions on Artificial Intelligence. Contextualización de AIOps en infraestructuras modernas.

Young, Ted y Alolita Sharma (2022). Cloud Native Observability with OpenTelemetry. Principios de observabilidad en entornos distribuidos. O'Reilly Media, Inc.

CNCF (2024a). Jaeger: open source, end-to-end distributed tracing. Línea futura para análisis de latencia de endpoints. url: https://www.jaegertracing.io/.

— (2024b). Prometheus: Monitoring system with a dimensional data model. Componente del ecosistema para enriquecimiento de alertas. url: https://prometheus.io/.

Dorcas, et al. (2024). «Enhancing Microservices Architecture with AI-Based Monitoring and Self-Healing Systems». En: ResearchGate. Base comparativa para la detección de fallos.

GoogleContainerTools (2024). Distroless Container Images. Base para la optimización y seguridad de la imagen de 91.7 MB. url: https://github.com/GoogleContainerTools/distroless.

IJACSA (2024). «Autonomous Self-Adaptation in the Cloud: ML-Heal's Framework for Proactive Fault Detection and Recovery». En: International Journal of Advanced Computer Science and Applications. Enfoque en detección proactiva de fallos.

Kubernetes Documentation (2024). Extending Kubernetes with Custom Resources. Referencia fundamental para el diseño del CRD MicroserviceMonitor. url: https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/.

Kubernetes SIG API Machinery (2024). Kubebuilder: SDK for Building Kubernetes APIs using CRDs. Framework utilizado para el scaffolding del operador. url: https://github.com/kubernetes-sigs/kubebuilder.

Kubernetes SIG Instrumentation (2024). Kubernetes Metrics Server. Prerrequisito para la Metrics API v1beta1 utilizada. url: https://github.com/kubernetes-sigs/metrics-server.

Meta AI (2024). The Llama 3 Series of Models. Propuesto para optimizar latencia mediante modelos locales en el futuro. url: https://llama.meta.com/.

NVIDIA Corporation (2024). NVIDIA Data Center GPU Manager (DCGM). Utilizado para el diagnóstico de métricas térmicas y de VRAM. url: https://developer.nvidia.com/dcgm/.

OpenAI (2024). GPT-4o Technical Report. Modelo gpt-4o-2024-08-06 utilizado para el análisis contextual. url: https://openai.com/index/hello-gpt-4o/.

OpenReview (2024). «Self-Evaluating AI Systems». En: OpenReview. Uso de LLMs como evaluadores inteligentes.

PagerDuty (2024). PagerDuty Events API v2 Overview. Integración para la gestión centralizada de incidentes. url: https://developer.pagerduty.com/docs/events-api-v2/overview/.

The Go Authors (2024). The Go Programming Language Documentation. Lenguaje seleccionado por su rendimiento concurrente. url: https://go.dev/doc/.

arXiv (2025). «An Intelligent Fault Self-Healing Mechanism for Cloud AI Systems via Integration of LLMs and DRL». En: arXiv preprint arXiv:2501.xxxxx. Precedente en el uso de LLMs para diagnóstico.

Anexo A. Artefactos de Despliegue y Distribución

Este anexo documenta los principales artefactos técnicos generados como resultado del desarrollo del Operador Inteligente de Kubernetes, con el objetivo de facilitar la reproducibilidad, la validación independiente de los resultados y la adopción práctica de la solución en entornos reales.

Imagen Docker Pública del Operador

El Operador Inteligente de Kubernetes ha sido empaquetado y distribuido como una imagen Docker pública, disponible en la plataforma Docker Hub. La publicación de la imagen constituye un elemento clave para garantizar la reproducibilidad de los experimentos descritos en este trabajo y demostrar la madurez técnica de la solución desarrollada.

La imagen se encuentra disponible en el siguiente repositorio:

supraseno/intelligent-operator

El repositorio contiene actualmente dos etiquetas estables:

- latest
- v0.0.1

Ambas versiones han sido construidas y validadas a partir del mismo código fuente utilizado durante la evaluación experimental descrita en los capítulos anteriores.

Proceso de Construcción de la Imagen

La imagen Docker ha sido generada mediante un proceso de multi-stage build, optimizado para entornos de producción. En la primera etapa se compila el binario del operador utilizando el lenguaje Go, generando un ejecutable estático independiente del entorno de ejecución.

La segunda etapa emplea una imagen base distroless (`gcr.io/distroless/static:nonroot`), lo que permite:

- Reducir significativamente el tamaño final de la imagen (aproximadamente 92 MB).
- Eliminar herramientas innecesarias del sistema operativo.
- Minimizar la superficie de ataque del contenedor.
- Ejecutar el proceso como usuario no privilegiado.

Este enfoque es consistente con las buenas prácticas de seguridad recomendadas para el despliegue de componentes críticos en Kubernetes.

Características de Seguridad

La imagen distribuida cumple una serie de principios básicos de seguridad:

- No incluye credenciales embebidas. Las claves de acceso a servicios externos, como la API de OpenAI, se gestionan exclusivamente mediante Kubernetes Secrets.
- El contenedor se ejecuta con un usuario no root (UID 65532).
- Los permisos de acceso al clúster están limitados mediante políticas RBAC de mínimo privilegio.

Estas medidas reducen el riesgo asociado a la ejecución del operador en entornos productivos.

Despliegue y Reproducibilidad

El despliegue del operador puede realizarse de forma automática mediante manifiestos Kubernetes públicos, o bien siguiendo un proceso manual controlado. La disponibilidad de la imagen Docker pública permite que cualquier evaluador o investigador pueda reproducir el entorno de ejecución descrito en este trabajo sin necesidad de compilar el código fuente.

El comando para la obtención de la imagen es:

```
docker pull supraseno/intelligent-operator:latest
```

La imagen ha sido validada en clústeres Kubernetes locales utilizando Minikube, aunque su diseño es compatible con entornos Kubernetes estándar en producción.

Consideraciones finales

La inclusión de estos artefactos técnicos refuerza el carácter aplicable, reproducible y transferible del trabajo realizado. La disponibilidad pública de la imagen Docker facilita la validación externa de los resultados y posiciona la solución como un sistema preparado para su adopción en entornos reales de operación.