UNIVERSIDAD POLITÉCNICA DE VICTORIA

DISEÑO E IMPLEMENTACIÓN DE UN MIDDLEWARE PARA LA INTEROPERABILIDAD ENTRE LA PLATAFORMA NEZ Y EL CONCENTRADOR DE SERVICIOS JUB

T E S I N A

QUE PARA OBTENER EL GRADO DE

INGENIERÍA EN TECNOLOGÍAS DE LA INFORMACIÓN

PRESENTA:

JOSE MANUEL ALONSO CEPEDA

DIRECTOR

DR. MARCO AURELIO NUÑO MAGANDA

CO-DIRECTOR

DR. JOSÉ LUIS GONZÁLEZ COMPEÁN

ORGANISMO RECEPTOR

CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS DEL INSTITUTO POLITÉCNICO NACIONAL (CINVESTAV) UNIDAD TAMAULIPAS

CIUDAD VICTORIA, TAMAULIPAS, SEPTIEMBRE DE 2025

Ciudad Victoria, Tamaulipas, a **23 de Septiembre de 2025**

CENTRO DE INVESTIGACION Y DE ESTUDIOS AVANZADOS DEL INSTITUTO POLITECNICO NACIONAL UNIDAD TAMAULIPAS (CINVESTAV)
DR. JOSÉ LUIS GONZÁLEZ COMPÉAN
PRESENTE

La Universidad Politécnica de Victoria tiene a bien presentar a **ALONSO CEPEDA JOSE MANUEL** estudiante del programa académico de **INGENIERÍA EN TECNOLOGÍAS DE LA INFORMACIÓN**, con número de matrícula **2130207** y seguro facultativo IMSS número **46-18-03-6197-7;** quién deberá realizar su práctica profesional de **ESTADÍA**, a partir del **01 de Septiembre de 2025** al **05 de Diciembre de 2025**, con duración de **600 horas** desarrollando el proyecto **"Diseño e implementación de un middleware para la interoperabilidad entre la plataforma Nez y el concentrador de servicios Jub"** que le fué asignado.

Al concluir se le extenderá la carta de liberación al evaluarlo satisfactoriamente y además le solicitaremos su valiosa opinión respondiendo el formulario que se le enviará por mail, referente al desempeño del practicante, la información que nos proporcione, es de vital importancia para la mejora de los programas académicos que ofrece la Universidad Politécnica de Victoria para la formación de profesionistas altamente especializados.

El practicante deberá cumplir con el Reglamento Interno aplicable al personal en su centro de trabajo.

Sin otro particular.

ATENTAMENTE

OTHÓN CANO GARZA
DIRECTOR DE VINCULACIÓN

C.C.P. MARCO AURELIONUÑOMAGANDA
ASESOR INSTITUCIONAL

UNIVERSIDAD POLITÉCNICA DE VICTORIA

Av. Nuevas Tecnologías 5902
Parque Científico y Tecnológico de Tamaulipas
Carretera Victoria Soto La Marina Km. 5.5
Cd. Victoria, Tamaulipas. C.P. 87138

Tel: (834) 1711100 al 10
www.upvictoria.edu.mx

CENTRO DE INVESTIGACION Y DE ESTUDIOS AVANZADOS DEL INSTITUTO POLITECNICO NACIONAL UNIDAD TAMAULIPAS (CINVESTAV)

Victoria Tamaulipas, a 23 de Septiembre del 2025
Asunto: Carta de Aceptación

M.A OTHÓN CANO GARZA
DIRECTOR DE VINCULACIÓN
UNIVERSIDAD POLITÉCNICA DE VICTORIA
PRESENTE

Hacemos de su conocimiento que hemos aceptado al estudiante **ALONSO CEPEDA JOSE MANUEL** del programa académico de **INGENIERÍA EN TECNOLOGÍAS DE LA INFORMACIÓN**, con número de matrícula **2130207** de la Universidad Politécnica de Victoria, para realizar su **ESTADÍA** en nuestra empresa **CENTRO DE INVESTIGACION Y DE ESTUDIOS AVANZADOS DEL INSTITUTO POLITECNICO NACIONAL UNIDAD TAMAULIPAS (CINVESTAV)**, durante el periodo comprendido del día **01 de Septiembre de 2025** al **05 de Diciembre de 2025**, el estudiante estará colaborando en el proyecto **"Diseño e implementación de un middleware para la interoperabilidad entre la plataforma Nez y el concentrador de servicios Jub"** con una carga horaria total de **600 horas**

Al concluir satisfactoriamente sus prácticas profesionales, se le entregará al estudiante su carta de liberación debidamente formalizada.

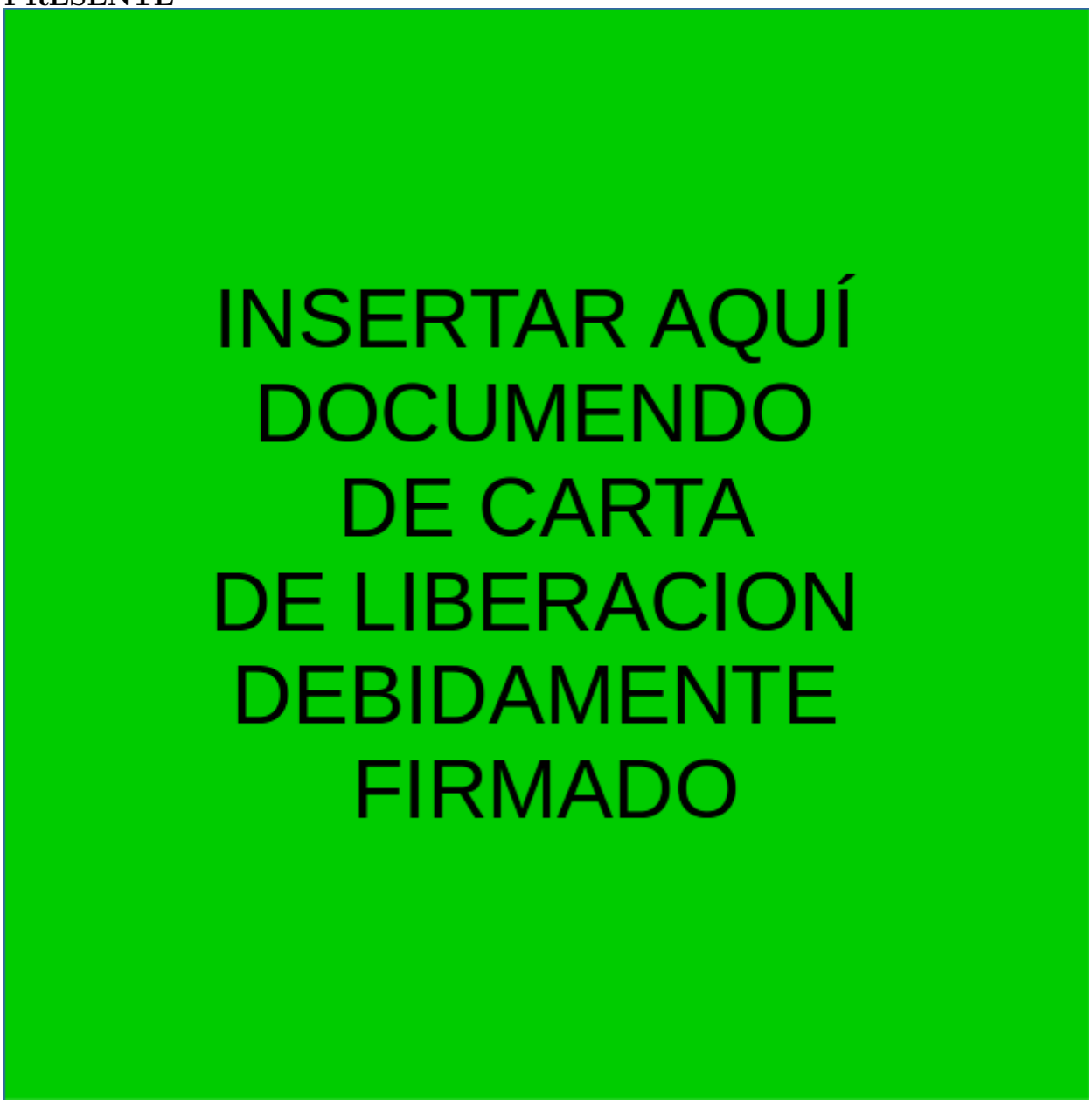
Con el objetivo de colaborar en la mejora de los programas académicos de la Universidad Politécnica de Victoria, estamos de acuerdo en responder a la brevedad posible el formulario de evaluación del desempeño del practicante previo a emitir la liberación de **ESTADÍA**.

Sin otro particular.

ATENTAMENTE
DR. JOSÉ LUIS GONZÁLEZ COMPÉAN
ASESOR EMPRESARIAL

C.C.P. ARCHIVO

Ing. Juan Camaney de Alba Rojas
PRESENTE



CARTA DE ACEPTACIÓN DEL DOCUMENTO PARA SU IMPRESIÓN

Cd. Victoria, Tamaulipas a 26 de Abril de 2025

Jose Manuel Alonso Cepeda
PRESENTE

Le comunico que el Programa Académico de Ingeniería en Tecnologías de la Información le ha otorgado la autorización para la impresión de su Tesina de Estadía Práctica cuyo título es:

Diseño e implementación de un middleware para la interoperabilidad entre la plataforma Nez y el concentrador de servicios Jub

ATENTAMENTE

Dr. Marco Aurelio Nuño Maganda
ASESOR INSTITUCIONAL

c.c.p Director de programa académico

UNIVERSIDAD POLITÉCNICA DE VICTORIA

Av. Nuevas Tecnologías 5902
Parque Científico y Tecnológico de Tamaulipas
Carretera Victoria Soto La Marina Km. 5.5
Cd. Victoria, Tamaulipas. C.P. 87138

Tel: (834) 1711100 al 10
www.upvictoria.edu.mx

EVALUACIÓN DE ESTADÍA

Rúbrica para evaluación de la presentación y el reporte de estadía

Nombre del alumno: **JOSE MANUEL ALONSO CEPEDA** Calificación final: _____

Periodo: **MAYO-AGOSTO 2025**

Ponderación	Aspecto a Evaluar	Competente 10	Independiente 9	Básico Avanzado 8	No Competente 5
40	Resultados y Actividades	Estrechamente relacionados al perfil de egreso de su programa académico	Parcialmente relacionados al perfil de egreso de su programa académico	Escasamente relacionados al perfil de egreso de su programa académico	Escasamente relacionados al perfil de egreso de su programa académico
30	Exposición de las actividades de la estadía	Detalladas y sustentadas con respecto a los resultados que se obtuvieron	Detalladas y sustentadas parcialmente con respecto a los resultados que se obtuvieron	Detalladas parcialmente con respecto a los resultados que se obtuvieron	Detalladas escazamente con respecto a los resultados que se obtuvieron
10	Material visual Lenguaje verbal	Uso el lenguaje y la terminología apropiadas; El material visual está organizado, adecuado y suficiente	Uso el lenguaje y la terminología apropiadas El material visual está parcialmente organizado y es suficiente	Uso el lenguaje y la terminología son parcialmente apropiadas; El material visual está parcialmente organizado y es suficiente	Uso el lenguaje y terminología es inapropiado; El material visual no está organizado y es insuficiente
10	Exposición en Idioma Inglés	Pronunciation is clear so language is easily understood (2.5) Uses fluent connected speech, occasionally disrupted by search for correct form of expression (2.5) Uses topic related vocabulary without problems (2.5) Responds to questions using varied and descriptive vocabulary and language structures (2.5)	Pronunciation is understandable, but there are slight errors (2.25) Speech is connected but frequently disrupted by search for correct form of expression (2.25) Uses some topic related vocabulary sufficient to communicate ideas (2.25) Responds to questions using simple but accurate vocabulary and language structures (2.25	Pronunciation is understandable most of the time, marked native accent and many errors (2) Speaks with simple sentences, sometimes not connected, but is understood (2) Uses basic vocabulary to communicate ideas (2) Partly responds to simple questions, with limited vocabulary and language structures (2)	Pronunciation makes language very difficult to understand (1) Uses one-word/two-word utterances (1) Unable to communicate ideas due to lack of vocabulary (1) Uses isolated words or sentence fragments to respond to questions (1)
5	Respuesta a los cuestiona-mientos de los evaluadores	Clara y satisfactoria	Clara y parcialmente satisfactoria	Clara e insuficiente	Confusa e insuficiente
5	Autorización de tesina en tiempo y forma	Presenta en tiempo y forma	Presenta en tiempo y forma con la mayoría de requerimientos solicitados	Presenta en tiempo y con algunas limitantes de los requerimientos solicitados.	Presenta fuera de tiempo y con los mínimos requerimientos solicitados.

Dr. Marco Aurelio Nuño Maganda
ASESOR INSTITUCIONAL

Dr. Hiram Herrera Rivas
EVALUADOR

EVALUADOR DE INGLÉS

REGISTRO DE EVALUACIÓN DE EXPOSICIÓN DE ESTADÍA

Siendo las 10:00 horas del día 11 de Agosto de 2021, el alumno **Jose Manuel Alonso Cepeda**, del programa académico **Ingeniería en Tecnologías de la Información**, con matricula **1730505**, presentó la exposición de la estadía realizada durante el cuatrimestre **Mayo-agosto 2025**, en la **CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS DEL INSTITUTO POLITÉCNICO NACIONAL (CINVESTAV) UNIDAD TAMAULIPAS**, con el proyecto titulado **Diseño e implementación de un middleware para la interoperabilidad entre la plataforma Nez y el concentrador de servicios Jub**.

Una vez concluido el proceso de evaluación, y con base a la rúbrica establecida para éste propósito, se determina que la calificación de la estadía es _____.

Dr. Marco Aurelio Nuño Maganda
ASESOR INSTITUCIONAL

Dr. Hiram Herrera Rivas
EVALUADOR

EVALUADOR DE INGLÉS

Agradecimientos

Resumen

El avance de la ciencia de datos y el aprendizaje profundo depende críticamente de la capacidad para integrar sistemas de software heterogéneos y especializados. Plataformas como *Nez*, un *framework* para el procesamiento de datos a gran escala basado en el modelo *PuzzleMesh*, y *Jub*, un concentrador de servicios para el monitoreo de fenómenos atmosféricos, ofrecen capacidades potentes pero operan en silos aislados, limitando su potencial sinérgico. Este trabajo aborda el desafío de la interoperabilidad entre estos dos sistemas mediante el diseño e implementación de un *middleware* de acoplamiento ligero.

La solución propuesta se basa en una arquitectura de API *RESTful*, que actúa como un puente de comunicación estandarizado, permitiendo que *Jub* invoque los servicios de procesamiento avanzado de *Nez* de manera transparente y eficiente. El *middleware* implementado no solo facilita el intercambio de datos y procesos en tiempo real, sino que también establece las bases para la creación de una malla de servicios de ciencia de datos unificada, gestionando las operaciones de almacenamiento a través del cliente *MictlanX*.

El prototipo fue desarrollado utilizando el *framework FastAPI* de Python, seleccionado por su alto rendimiento y sus capacidades para la rápida creación de APIs robustas. La validación del sistema se realizó a través de un caso de uso de procesamiento de imágenes, demostrando la viabilidad y eficacia del *middleware* como catalizador para la integración de sistemas complejos en entornos de investigación científica.

Palabras clave: Middleware, Interoperabilidad, Malla de Servicios, API REST, FastAPI, Nez, Jub, Sistemas Distribuidos.

Summary

The advancement of data science and deep learning critically depends on the ability to integrate heterogeneous and specialized software systems. Platforms such as *Nez*, a framework for large-scale data processing based on the *PuzzleMesh* model, and *Jub*, a service hub for monitoring atmospheric phenomena, offer powerful capabilities but operate in isolated silos, limiting their synergistic potential. This work addresses the challenge of interoperability between these two systems through the design and implementation of a lightweight *middleware*.

The proposed solution is based on a RESTful API architecture, which acts as a standardized communication bridge, allowing *Jub* to invoke *Nez*'s advanced processing services transparently and efficiently. The implemented *middleware* not only facilitates the real-time exchange of data and processes but also lays the foundation for creating a unified data science service mesh, managing storage operations through the *MictlanX* client.

The prototype was developed using the Python *FastAPI* framework, chosen for its high performance and its capabilities for the rapid creation of robust APIs. The system's validation was conducted through an image processing use case, demonstrating the feasibility and effectiveness of the *middleware* as a catalyst for integrating complex systems in scientific research environments.

Keywords: Middleware, Interoperability, Service Mesh, REST API, FastAPI, Nez, Jub, Distributed Systems.

Índice

1. Introducción

Vivimos en una era definida por la explosión de datos, comúnmente denominada la era del *Big Data*. Campos tan diversos como la genómica, la astrofísica, las finanzas y las ciencias de la Tierra están generando volúmenes de información a una escala sin precedentes [1]. En particular, el sector de la salud y la observación geoespacial han emergido como dos de los dominios más prolíficos en la generación de datos complejos, como imágenes médicas de alta resolución y datos satelitales multiespectrales. El análisis de esta vasta cantidad de información es fundamental para la toma de decisiones críticas, desde el diagnóstico temprano de enfermedades hasta el monitoreo del cambio climático. Para enfrentar este desafío, han surgido tecnologías transformadoras como el cómputo en la nube, que proporciona la infraestructura escalable necesaria, y el aprendizaje profundo (*deep learning*), que ofrece métodos potentes para extraer conocimiento de estos datos masivos [2].

Sin embargo, la creciente especialización tecnológica ha llevado a una fragmentación significativa. Las organizaciones, tanto en el ámbito académico como en el industrial, desarrollan o adoptan sistemas de software altamente especializados, optimizados para tareas muy concretas. Si bien esta especialización es beneficiosa para resolver problemas específicos, a menudo conduce a la creación de "silos de datos y procesamiento". Estos silos son ecosistemas tecnológicos aislados que, aunque potentes individualmente, carecen de la capacidad de comunicarse e interactuar entre sí de manera fluida. Esta falta de interoperabilidad se ha convertido en uno de los desafíos más significativos de la ingeniería de software moderna en sistemas distribuidos [3]. La heterogeneidad en lenguajes de programación, protocolos de comunicación y formatos de datos impide la creación de flujos de trabajo integrados, limitando el potencial científico y operativo que podría surgir de la combinación sinérgica de estas herramientas.

Para superar esta barrera, la industria ha adoptado el uso de *middleware* como una solución estratégica. El *middleware* actúa como un "pegamento de software", una capa de abstracción que se sitúa entre aplicaciones dispares y les permite comunicarse de manera estandarizada, ocultando la complejidad subyacente de la red y los sistemas operativos. En el contexto de la investigación científica, el *middleware* es crucial para la construcción de flujos de trabajo científicos (*scientific workflows*), que permiten automatizar y orquestar secuencias complejas de procesamiento y análisis de datos a través de múltiples sistemas distribuidos [4]. Este enfoque no solo mejora la eficiencia y la reproducibilidad de la investigación, sino que también fomenta la colaboración al permitir que diferentes herramientas y servicios se combinen de formas novedosas.

1.1. Definición del problema y justificación del proyecto

En el Centro de Investigación y de Estudios Avanzados (CINVESTAV) Unidad Tamaulipas, el desafío de la interoperabilidad se manifiesta de manera concreta en la interacción (o la falta de ella) entre dos plataformas de software estratégicas. Por un lado, se encuentra **Nez**, un *framework* avanzado para el procesamiento de datos a gran escala. *Nez* no es solo una herramienta, sino la implementación del innovador modelo arquitectónico *PuzzleMesh*, que concibe las aplicaciones como "piezas de rompecabezas" modulares y autocontenidas que

pueden ser ensambladas para construir estructuras de procesamiento complejas y dinámicas. Esta arquitectura ha sido validada con éxito en dominios de alto impacto como el análisis de tomografías para diagnóstico médico y el procesamiento de imágenes satelitales para la observación de la Tierra.

La plataforma *Nez* es, de hecho, un componente clave dentro de un ecosistema tecnológico más amplio y ambicioso denominado **Muyal-Ilal**, un servicio en la nube diseñado por investigadores de la Universidad Autónoma Metropolitana para la gestión, aseguramiento, intercambio y procesamiento de grandes volúmenes de datos, con un enfoque particular en el sector salud [5]. Dentro de este ecosistema, *Nez* funciona como la plataforma de construcción de sistemas, guiando al personal de salud y de tecnologías de la información en la creación de flujos de trabajo de manera intuitiva, utilizando un enfoque de "bloques de Lego" para ensamblar herramientas de inteligencia artificial. Este ecosistema se complementa con otros servicios especializados como *Muyal-Painal*, dedicado al almacenamiento y distribución de datos, y *Muyal-Xelhua*, enfocado en la analítica de datos masivos. La filosofía de *Nez* es, por tanto, facilitar la creación de sistemas complejos sin requerir conocimientos avanzados de programación, resolviendo dependencias tecnológicas a través de su arquitectura modular.

Por otro lado, se encuentra **Jub**, un concentrador y distribuidor de datos diseñado específicamente para el monitoreo de fenómenos atmosféricos, actuando como un punto central para la ingesta y distribución de flujos de datos en tiempo real. Adicionalmente, un tercer componente, *MictlanX*, gestiona las operaciones de almacenamiento subyacentes para este ecosistema.

El problema central que aborda esta tesina es la ausencia de un mecanismo de comunicación nativo y estandarizado entre *Nez* y *Jub*. A pesar de sus capacidades evidentemente complementarias, estas dos plataformas operan en silos tecnológicos. Esta desconexión impone una barrera significativa: los usuarios de *Jub* no pueden invocar de manera programática y automatizada las potentes capacidades de análisis y aprendizaje profundo de *Nez*, y este último no puede ser alimentado directamente por los flujos de datos curados y distribuidos por *Jub*. Este aislamiento tecnológico no solo representa una subutilización de los recursos existentes, sino que impide activamente la formación de una malla de servicios de ciencia de datos (*Service Mesh*) cohesiva y eficiente, un paradigma arquitectónico moderno para gestionar la comunicación entre microservicios de manera fiable y segura [6].

La justificación de este proyecto es, por tanto, clara y apremiante. El desarrollo de un *middleware* que actúe como un puente de comunicación estandarizado es un paso indispensable para romper estos silos. Al permitir una integración transparente y ligera, este *middleware* no solo conectará dos sistemas, sino que sentará las bases para una arquitectura de servicios distribuidos mucho más ambiciosa y escalable. Permitirá desbloquear nuevas y valiosas líneas de investigación y aplicación al combinar el análisis de datos atmosféricos con el procesamiento avanzado de imágenes y otras formas de análisis computacional. En resumen, este trabajo es un paso fundamental y estratégico para maximizar el valor y el impacto de los activos tecnológicos existentes en la institución, alineándolos con las mejores prácticas de la ingeniería de software contemporánea.

1.2. Objetivo General

Diseñar e implementar un *middleware* de acoplamiento ligero que permita la interoperabilidad entre la plataforma *Nez* y el concentrador de servicios *Jub*, habilitando la creación de una malla de servicios de ciencia de datos y aprendizaje profundo para el procesamiento distribuido de datos e imágenes de distintos dominios.

1.3. Objetivos Particulares

- Diseñar la arquitectura del *middleware* para permitir la comunicación eficiente entre las plataformas *Nez* y *Jub*, basándose en principios de arquitecturas orientadas a servicios.
- Implementar el *middleware* para lograr un acoplamiento ligero que habilite el intercambio de datos y la invocación de procesos en tiempo real.
- Integrar y validar el flujo de trabajo completo mediante la implementación de servicios de procesamiento distribuido de datos e imágenes, utilizando algoritmos de aprendizaje profundo.
- Mejorar las interfaces gráficas de usuario existentes en la plataforma *Jub* para incorporar la nueva funcionalidad, facilitando la gestión y visualización de los datos procesados.
- Elaborar la documentación técnica detallada y los manuales de usuario del sistema para garantizar su correcto uso, mantenimiento y extensibilidad futura.

1.4. Alcances y limitaciones del Proyecto

El alcance de este proyecto se centra en la entrega de un prototipo funcional y bien documentado del *middleware* de interoperabilidad. Este prototipo será capaz de recibir solicitudes desde *Jub*, orquestar la ejecución de procesos de análisis en *Nez* y gestionar el flujo de los resultados de vuelta al sistema de origen. El proyecto incluye la validación de la solución completa a través de un caso de uso específico de procesamiento de imágenes, así como las mejoras necesarias a la interfaz de usuario de *Jub* para integrar esta nueva funcionalidad de manera intuitiva.

No obstante, el proyecto presenta las siguientes limitaciones en su fase actual:

- El prototipo se validará con un conjunto limitado y bien definido de casos de uso, sin abarcar la totalidad de las capacidades potenciales de *Nez* y *Jub*.
- El manejo de errores en el *middleware* se centrará en la notificación de fallos y el registro de eventos, sin implementar mecanismos avanzados de resiliencia como reintentos automáticos con retroceso exponencial (*exponential backoff*) o patrones de *circuit breaking*.
- La solución no incluirá una integración nativa con herramientas de monitoreo y observabilidad de nivel de producción, como *Prometheus* para métricas o *Grafana* para

visualización, aunque su arquitectura permitirá dicha integración en el futuro.

- El modelo para consultar el estado de los trabajos de procesamiento se basará en un mecanismo de sondeo (*polling*) iniciado por el cliente, en lugar de un sistema de notificaciones proactivas basadas en eventos (como *webhooks*), que podría ser más eficiente en ciertos escenarios.

1.5. Organización del Documento de Tesina

Este documento se organiza en seis capítulos para presentar de manera clara, lógica y estructurada el desarrollo completo del proyecto.

El **Capítulo 2** establece el **Marco Teórico**, donde se exploran y definen los conceptos fundamentales que sustentan este trabajo. Se abordan en profundidad temas como las arquitecturas de mallas de servicios (*Service Mesh*), la teoría y práctica del *middleware*, los desafíos de la interoperabilidad en sistemas heterogéneos, y se describen las tecnologías clave involucradas, con especial énfasis en el modelo conceptual *PuzzleMesh* de la plataforma *Nez*.

El **Capítulo 3** se dedica por completo al **Diseño Arquitectónico** del *middleware*. En esta sección se detallan los requisitos funcionales y no funcionales, se justifican las decisiones de diseño cruciales, como la elección de un protocolo de comunicación personalizado implementado en Python como interfaz de comunicación, y se especifica formalmente el contrato de la API, incluyendo *endpoints*, formatos de datos y códigos de estado.

El **Capítulo 4** describe la **Implementación** del prototipo. Este capítulo abarca la selección del *stack* tecnológico (lenguajes, *frameworks*), la estructura del código fuente, las decisiones de diseño a bajo nivel y las estrategias de empaquetado y despliegue mediante contenerización con *Docker*.

En el **Capítulo 5** se presentan las **Pruebas y Validación** del sistema. Aquí se detalla el escenario de prueba configurado, se exponen los resultados de las pruebas de integración que demuestran la correcta comunicación entre *Jub* y *Nez*, y se realiza un análisis de rendimiento preliminar para evaluar la latencia y el *throughput* del *middleware*.

Finalmente, el **Capítulo 6** expone las **Conclusiones y el Trabajo Futuro**. En esta última sección se resumen las contribuciones más importantes del proyecto, se reflexiona sobre los resultados obtenidos y se delinean posibles líneas de investigación y desarrollo para extender y mejorar el trabajo realizado, como la implementación de mecanismos de seguridad más robustos o la integración con sistemas de orquestación avanzados.

2. Marco Teórico

Este capítulo establece el marco conceptual y tecnológico sobre el cual se fundamenta el proyecto. Se exploran en profundidad las arquitecturas de software para sistemas distribuidos, con especial énfasis en los microservicios y las mallas de servicios, que constituyen el paradigma moderno para la construcción de aplicaciones escalables. Posteriormente, se formaliza el modelo conceptual **PuzzleMesh**, detallando su enfoque innovador para la composición de sistemas agnósticos a la infraestructura. Se introduce el rol del *middleware* como un catalizador indispensable para la interoperabilidad, contextualizando su función dentro del ecosistema de integración de este proyecto. Finalmente, se describen las plataformas de software clave (Nez, Jub, MictlanX) y los mecanismos tecnológicos, como la monitorización del sistema de archivos y la contenerización, que habilitan la implementación de la solución propuesta.

2.1. Arquitecturas de Software para Sistemas Distribuidos

Un sistema distribuido es una colección de componentes de software autónomos, ubicados en diferentes nodos de una red, que se comunican y coordinan entre sí para aparecer ante el usuario como un único sistema coherente [7]. El diseño de estos sistemas presenta desafíos únicos en áreas como la comunicación, la coordinación, la escalabilidad y la tolerancia a fallos. Las arquitecturas de software proporcionan el andamiaje conceptual para abordar esta complejidad.

2.1.1. La Arquitectura de Microservicios

La arquitectura de microservicios ha surgido como el enfoque predominante para construir aplicaciones complejas y escalables. Este estilo arquitectónico estructura una aplicación como una colección de servicios pequeños, autónomos y débilmente acoplados, que se comunican a través de APIs bien definidas, comúnmente sobre HTTP [8].

Principios y Características. A diferencia de las arquitecturas monolíticas tradicionales, donde toda la funcionalidad reside en una única base de código, los microservicios se adhieren a los siguientes principios:

- **Alta Cohesión y Bajo Acoplamiento:** Cada servicio encapsula una capacidad de negocio específica y completa. Las dependencias entre servicios se minimizan y se gestionan a través de interfaces estables, lo que reduce el impacto de los cambios.
- **Autonomía:** Cada servicio puede ser desarrollado, desplegado, escalado y actualizado de forma independiente del resto de la aplicación. Esto permite a los equipos trabajar en paralelo y acelera los ciclos de entrega.
- **Propiedad Descentralizada de los Datos:** Idealmente, cada microservicio es propietario de sus propios datos y es el único responsable de su persistencia. La comunicación con otras bases de datos se realiza a través de la API del servicio propietario, evitando el acoplamiento a nivel de base de datos [9].

- **Diversidad Tecnológica (Poliglotismo):** Los equipos pueden elegir la pila tecnológica (lenguaje de programación, base de datos, etc.) más adecuada para los requisitos específicos de su servicio, sin estar atados a una decisión tecnológica unificada para toda la aplicación.

Ventajas y Desafíos. Las principales ventajas de los microservicios son la agilidad, la escalabilidad granular (se pueden escalar solo los servicios que lo necesitan) y la resiliencia (un fallo en un servicio no tiene por qué derribar toda la aplicación). Sin embargo, esta arquitectura introduce una complejidad operativa significativa. La gestión de la comunicación entre docenas o cientos de servicios, el descubrimiento de servicios, el balanceo de carga, la gestión de fallos en cascada y la observabilidad del sistema en su conjunto se convierten en desafíos de primer orden [10].

2.1.2. Mallas de Servicios (Service Mesh)

Para abordar la complejidad operativa de los microservicios, ha surgido el paradigma de la Malla de Servicios (*Service Mesh*). Una malla de servicios es una capa de infraestructura de software dedicada y configurable que gestiona la comunicación entre los servicios de una aplicación. Su objetivo es externalizar la lógica de comunicación de red fuera del código de la aplicación, proporcionando funcionalidades críticas de manera uniforme y transparente [6].

Arquitectura: Plano de Datos y Plano de Control. La arquitectura de una malla de servicios se divide conceptualmente en dos componentes principales:

- **Plano de Datos (Data Plane):** Está compuesto por una red de *proxies* ligeros que se despliegan junto a cada instancia de un microservicio, en un patrón conocido como *sidecar*. Todo el tráfico de red de entrada y salida de un servicio es interceptado y enrutado a través de su *proxy sidecar*. Estos *proxies* gestionan el balanceo de carga, el descubrimiento de servicios, la aplicación de políticas de seguridad y la recopilación de métricas de telemetría.

- **Plano de Control (Control Plane):** Es el çerebro"de la malla de servicios. Proporciona una API para que los operadores configuren y administren el comportamiento de los *proxies* en el plano de datos. El plano de control centraliza la configuración de políticas, agrega datos de telemetría de todos los *proxies* y proporciona una visión unificada del estado y el rendimiento de la malla.

Herramientas como Istio, Linkerd y Consul son implementaciones populares de este patrón, que se ha convertido en un estándar de facto para la gestión de aplicaciones nativas de la nube [11].

2.1.3. Patrones de Comunicación en Sistemas Distribuidos

La comunicación es la columna vertebral de cualquier sistema distribuido. La elección del patrón de comunicación tiene un impacto profundo en las características del sistema, como

el acoplamiento, la resiliencia y la escalabilidad. Fundamentalmente, los patrones se dividen en dos categorías: síncronos y asíncronos.

Comunicación Síncrona. En este patrón, el cliente envía una petición a un servicio y bloquea su ejecución, esperando una respuesta. Si el servicio no responde (ya sea por una falla o por latencia), el cliente permanece bloqueado. La comunicación a través de APIs REST sobre HTTP, como se describe en la tesis de Fielding
citefielding2000rest, es el ejemplo más común de este patrón.

- **Ventajas:** Es un modelo mentalmente simple y familiar, similar a una llamada a una función local. El cliente recibe una respuesta inmediata sobre el éxito o fracaso de la operación, lo que facilita el manejo de errores directos.
- **Desventajas:** Introduce un fuerte acoplamiento temporal. El cliente y el servidor deben estar disponibles simultáneamente. Una falla en el servicio puede causar fallas en cascada, ya que los clientes que dependen de él quedan bloqueados. La latencia del sistema general está determinada por la suma de las latencias de las llamadas en la cadena.

Comunicación Asíncrona. En el patrón asíncrono, el cliente envía un mensaje o evento y continúa con su procesamiento sin esperar una respuesta inmediata. La comunicación se realiza típicamente a través de un intermediario conocido como **Message Broker** o bus de eventos (por ejemplo, RabbitMQ, Apache Kafka). El cliente deposita un mensaje en una cola y el servicio lo consume cuando está disponible.

- **Ventajas:**
  - **Desacoplamiento Temporal:** El cliente y el servidor no necesitan estar disponibles al mismo tiempo. Si el servicio consumidor está caído, el mensaje permanece en la cola hasta que el servicio se recupera, aumentando la resiliencia del sistema.
  - **Balanceo de Carga y Escalabilidad:** Múltiples instancias de un servicio pueden consumir mensajes de la misma cola, lo que permite distribuir la carga de trabajo de manera natural y escalar los servicios de forma independiente.
  - **Amortiguación de Carga (Load Leveling):** El message broker actúa como un búfer que puede absorber picos de carga, protegiendo a los servicios consumidores de ser sobrecargados. Los mensajes se encolan y se procesan al ritmo que el consumidor puede manejar.
- **Desventajas:** Aumenta la complejidad del sistema al introducir un nuevo componente (el broker). El modelo de programación es más complejo, ya que se debe lidiar con la consistencia eventual y la correlación de mensajes si se necesita una respuesta.

La elección entre estos patrones no es excluyente; muchos sistemas complejos utilizan una combinación de ambos. Para el middleware de este proyecto, aunque la interacción final

con MictlanX puede ser una llamada síncrona, el disparador del proceso es inherentemente asíncrono: el middleware reacciona a un evento (la creación de un archivo) que ocurrió en un momento indeterminado, desacoplando así el proceso de generación de datos de Nez del proceso de registro en Jub. Este enfoque basado en eventos se alinea con los principios de la comunicación asíncrona, proporcionando resiliencia y autonomía al flujo de trabajo de integración [12].

2.2. El Modelo Conceptual de PuzzleMesh

El modelo **PuzzleMesh** es un marco de trabajo formal, desarrollado en el CINVESTAV, para la construcción de estructuras de procesamiento de datos que son, por diseño, agnósticas a la infraestructura subyacente. Utiliza una poderosa metáfora de rompecabezas"para abstraer la complejidad de la composición de sistemas distribuidos, alineándose con los principios de modularidad y reutilización promovidos por las mallas de servicios [13].

2.2.1. La Pieza de Software (P): La Unidad Fundamental

La unidad básica y fundamental del modelo es la **Pieza de Software (P)**. Una pieza no es solo una aplicación, sino un artefacto de software **autocontenido** que encapsula una funcionalidad específica junto con todos los componentes necesarios para su correcto despliegue y ejecución. La arquitectura de una pieza se puede visualizar como una pila con las siguientes capas:

- **Capa de Acceso:** Verifica que el acceso a la pieza y sus componentes sea válido, usualmente mediante un sistema de tokenización.
- **Interfaces de Entrada/Salida (E/S):** Definen los puntos de conexión de la pieza. Las interfaces de entrada (*sockets*) leen datos de una fuente, mientras que las de salida (*loops*) escriben los resultados en un destino. Representan la "forma.º çurvatura"de la pieza que le permite encajar con otras.
- **Aplicación:** Contiene el código fuente o el binario de la aplicación que realiza el procesamiento principal.
- **Metadatos:** Incluyen los parámetros de configuración, la ruta de ejecución y cualquier otra información necesaria para invocar la aplicación.
- **Dependencias:** Empaqueta todas las librerías, variables de entorno y otros artefactos de los que depende la aplicación.

Esta estructura garantiza que cada pieza sea un componente autónomo y portable, una verdadera çaja negra"funcional.

2.2.2. El Rompecabezas (R): Componiendo Flujos de Trabajo

Un **Rompecabezas (R)** es una estructura de procesamiento de alto nivel que se crea al acoplar un conjunto de piezas. La lógica de composición se define mediante un **Grafo Acíclico**

Dirigido (DAG), donde los nodos del grafo son las piezas y los vértices representan las dependencias y el flujo de datos entre ellas. Un rompecabezas define un flujo de trabajo completo, desde una o más fuentes de datos hasta uno o más destinos, orquestando la ejecución de las piezas que lo componen.

Parte I

title

2.2.3. El Metarompecabezas (Ω): Interoperabilidad Inter-Sistemas

El modelo PuzzleMesh escala el concepto de composición al introducir el **Metarompecabezas (Ω)**. Un metarompecabezas se construye encadenando múltiples rompecabezas. Esta capacidad es la que permite modelar flujos de trabajo extremadamente complejos que pueden cruzar fronteras departamentales, organizacionales o de infraestructura. Por ejemplo, un metarompecabezas podría modelar un flujo donde los datos se originan en el sistema A (representado por un rompecabezas), se procesan en el sistema B (otro rompecabezas) y finalmente se consumen en el sistema C (un tercer rompecabezas). Esta abstracción es clave para lograr una interoperabilidad real y a gran escala [14].

2.3. Middleware como Catalizador de la Interoperabilidad

La interoperabilidad, o la capacidad de sistemas heterogéneos para comunicarse y trabajar conjuntamente, es uno de los mayores desafíos en la ingeniería de software moderna. El *middleware* es la solución estratégica a este problema. En su nivel más fundamental, se define como una capa de software que se sitúa entre el sistema operativo y las aplicaciones, o entre aplicaciones dispares, para abstraer la complejidad de la comunicación en un entorno distribuido [15]. Funciona como un "pegamento de software.º una "fontanería de software"que conecta componentes de software que de otro modo estarían aislados, permitiéndoles interactuar de manera coherente [16].

2.3.1. Definición y Objetivos del Middleware

El concepto de middleware surge de la necesidad de simplificar el desarrollo de sistemas distribuidos. En lugar de que cada aplicación deba resolver problemas complejos de comunicación, concurrencia y heterogeneidad, el middleware proporciona una capa de abstracción que ofrece soluciones reusables a estos problemas.

Sus objetivos fundamentales son:

- **Abstracción y Homogeneización:** Ocultar las diferencias entre plataformas de hardware, sistemas operativos y protocolos de red. El middleware crea un entorno de eje-

cución homogéneo que simplifica drásticamente el desarrollo, ya que los programadores pueden escribir código para una única API abstracta en lugar de para múltiples plataformas concretas.

- **Interoperabilidad:** Facilitar que sistemas desarrollados con diferentes lenguajes de programación y tecnologías puedan intercambiar datos y servicios de manera significativa. Actúa como un traductor universal que permite a dos sistemas hablar entre sí sin necesidad de entender los detalles internos del otro.

- **Provisión de Servicios Comunes:** Ofrecer un catálogo de servicios de alto nivel que las aplicaciones distribuidas necesitan comúnmente. Esto incluye la gestión de la comunicación, la seguridad (autenticación y cifrado), la gestión de transacciones distribuidas, la concurrencia y la notificación de eventos.

- **Mejora de la Productividad del Desarrollador:** Al encargarse de la "fontanería"de la comunicación y otros servicios de bajo nivel, el middleware permite que los equipos de desarrollo se centren en la lógica de negocio de la aplicación, acelerando el tiempo de desarrollo y reduciendo la probabilidad de errores.

2.3.2. Características Clave de un Sistema Middleware

Para cumplir con sus objetivos, un sistema middleware robusto típicamente exhibe las siguientes características:

- **Transparencia de Red:** Oculta los detalles de la comunicación en red (sockets, puertos, protocolos). Para el desarrollador, la interacción con un servicio remoto se siente como si fuera una llamada a una función local.

- **Independencia de la Plataforma:** El middleware está diseñado para ser portable y operar sobre diferentes sistemas operativos y arquitecturas de hardware, garantizando que las aplicaciones que lo utilizan también hereden esta portabilidad.

- **Confiabilidad:** Ofrece mecanismos para garantizar la entrega de mensajes y la gestión de errores en la comunicación. Puede incluir reintentos automáticos, confirmaciones de entrega y gestión de transacciones para asegurar la consistencia.

- **Escalabilidad:** Debe ser capaz de gestionar un número creciente de peticiones, conexiones y volúmenes de datos sin que su rendimiento se degrade significativamente.

- **Seguridad:** Proporciona servicios integrados de autenticación, autorización y cifrado para proteger la comunicación entre los componentes del sistema distribuido.

2.3.3. El Middleware en el Contexto de Este Proyecto

El middleware diseñado en este proyecto es una manifestación práctica de los conceptos anteriores, adaptado a las necesidades específicas de interoperabilidad entre las plataformas

Nez y Jub. No es un intermediario pasivo, sino un componente de software **activo y reactivo**, cuya función es automatizar un flujo de trabajo que de otro modo sería manual.

Arquitectura Reactiva Basada en Eventos. La decisión de diseño clave es que el middleware no espera a ser invocado, sino que reacciona a eventos que ocurren en su entorno. Esta arquitectura se implementa a través de dos componentes principales:

- **El Demonio (Daemon):** El núcleo del middleware se implementa como un demonio, un proceso que se ejecuta de forma continua en segundo plano, sin intervención del usuario. Esta es la decisión arquitectónica que garantiza su **operación continua y autónoma**, permitiendo una vigilancia desatendida del sistema, lo cual es esencial para un servicio de automatización.

- **El Monitor del Sistema de Archivos (Watcher):** Este componente, implementado con la biblioteca `watchdog`, actúa como los "sentidos"del middleware. Se suscribe a los eventos del sistema de archivos en un directorio específico: el directorio de salida de Nez. En lugar de sondear (polling) ineficientemente el directorio a intervalos fijos, utiliza las notificaciones del sistema operativo para ser informado instantáneamente cuando ocurre un cambio.

Flujo de Trabajo de Interoperabilidad. El funcionamiento del middleware sigue una secuencia clara y automatizada que materializa el puente entre Nez y Jub:

1. El demonio del middleware se inicia y entra en un estado de vigilancia pasiva, consumiendo mínimos recursos del sistema.

2. El monitor `watchdog` se registra en el sistema operativo para observar el directorio de salida de la plataforma Nez.

3. La plataforma Nez completa una de sus tareas de procesamiento y genera un nuevo producto de datos (un archivo o una carpeta) en dicho directorio.

4. El sistema operativo notifica inmediatamente al proceso `watchdog` sobre el evento de creación de un nuevo archivo.

5. Esta notificación actúa como el **disparador (trigger)** que "despierta"la lógica principal del middleware.

6. El middleware lee el nuevo archivo y sus metadatos asociados para identificar el producto de datos.

7. A continuación, invoca al cliente de la plataforma MictlanX, pasándole el artefacto para su registro y publicación en el ecosistema de Jub. En este paso, el middleware actúa como un **cliente de servicio**.

8. Una vez completada la operación, el middleware vuelve a su estado de espera, listo para

el próximo evento.

Este diseño cumple con los objetivos de un middleware al proporcionar interoperabilidad y abstracción de una manera eficiente y de bajo acoplamiento, funcionando como un puente automatizado y reactivo que cohesiona dos sistemas previamente aislados.

2.4. Observabilidad en Sistemas Distribuidos

A medida que los sistemas se vuelven más distribuidos y complejos, los métodos tradicionales de depuración y monitoreo se vuelven insuficientes. La observabilidad es una propiedad de un sistema que permite entender su estado interno a partir de los datos que genera externamente. No se trata solo de monitorear si un sistema está .ªrriba.º .ªbajo"(monitoreo de caja negra), sino de tener la capacidad de hacer preguntas arbitrarias sobre su comportamiento sin necesidad de predefinirlas [17]. En el contexto de este proyecto, la observabilidad es crucial para garantizar que el middleware, un componente autónomo y desatendido, funcione de manera correcta y confiable.

La observabilidad se sustenta en tres pilares fundamentales de datos de telemetría:

2.4.1. Logs (Registros)

Los logs son registros inmutables y con marca de tiempo de eventos discretos que ocurrieron en un punto específico del sistema. Son la forma más antigua y fundamental de telemetría.

- **Función:** Proporcionan un contexto detallado y de alto nivel sobre un evento. Por ejemplo, el middleware podría generar un log al iniciarse, al detectar un nuevo archivo, al invocar al cliente de MictlanX, o si ocurre un error durante el proceso.
- **Características:** Los logs deben ser estructurados (por ejemplo, en formato JSON) en lugar de texto plano. Un log estructurado permite realizar búsquedas, filtros y agregaciones de manera eficiente. Un buen log incluye información como la marca de tiempo, el nivel de severidad (INFO, WARN, ERROR), un mensaje descriptivo y metadatos relevantes (por ejemplo, el nombre del archivo procesado).
- **En este proyecto:** La implementación de logs detallados en el middleware es esencial para la depuración post-mortem. Si un archivo no se procesa correctamente, los logs son el primer lugar para investigar qué sucedió, qué errores se produjeron y en qué punto del flujo de trabajo falló el proceso.

2.4.2. Métricas

Las métricas son agregaciones numéricas de datos sobre el comportamiento de un sistema a lo largo del tiempo. A diferencia de los logs, que registran eventos individuales, las métricas resumen el estado y el rendimiento.

- **Función:** Son ideales para la monitorización en tiempo real, la creación de dashboards y la configuración de alertas automáticas. Permiten identificar tendencias y anomalías

en el comportamiento del sistema.

- **Tipos Comunes:**
  - **Contadores (Counters):** Un valor que solo se incrementa, como el número total de archivos procesados o el número de errores ocurridos.
  - **Medidores (Gauges):** Un valor que puede subir o bajar, como el número de archivos actualmente en cola para ser procesados.
  - **Histogramas (Histograms):** Miden la distribución de un conjunto de valores, como la latencia de las llamadas a la API de MictlanX o el tamaño de los archivos procesados.
- **En este proyecto:** El middleware podría exponer métricas como 'archivos_procesados_total', 'errores_de_procesamiento_total', y 'latencia_registro_mictlanx_seconds'. Estas métricas permitirían crear un dashboard para visualizar la salud y el rendimiento del middleware de un vistazo y alertar si, por ejemplo, la tasa de errores aumenta.

2.4.3. Trazas Distribuidas (Distributed Tracing)

Mientras que los logs se centran en un evento y las métricas en una agregación, las trazas se centran en el ciclo de vida de una petición o flujo de trabajo a medida que se propaga a través de múltiples servicios.

- **Función:** Son indispensables para entender la latencia y las dependencias en un sistema de microservicios o distribuido. Una traza sigue una única solicitud desde el principio hasta el fin, registrando cuánto tiempo pasó en cada servicio o componente.
- **Anatomía de una Traza:** Una traza está compuesta por *spans*. Cada span representa una unidad de trabajo (por ejemplo, una llamada a una función o una petición de red) y contiene un nombre, una hora de inicio y fin, y metadatos. Los spans se anidan para mostrar la relación padre-hijo.
- **En este proyecto:** Aunque la implementación de trazas distribuidas completas puede ser compleja, el concepto es relevante. Se podría crear una "traza"para cada archivo procesado. El span raíz sería "Procesar Nuevo Archivo", y tendría spans hijos como "Leer Archivo del Disco", Çalcular Hash SHA-256z Registrar en MictlanX". Esto permitiría identificar cuellos de botella en el flujo de trabajo del middleware.

En conjunto, estos tres pilares proporcionan una visión completa y profunda del sistema, permitiendo no solo detectar fallos, sino también entender por qué ocurrieron, un requisito indispensable para la operación robusta de cualquier sistema de software crítico.

2.5. Plataformas del Ecosistema de Integración

El ecosistema tecnológico en el que se enmarca este proyecto está compuesto por tres plataformas clave, todas ellas desarrolladas internamente en el **CINVESTAV Tamaulipas**, lo que proporciona un conocimiento profundo de su funcionamiento interno.

- **Nez:** Es un *framework* avanzado para el procesamiento de datos a gran escala. Su principal innovación es ser la implementación de referencia del modelo conceptual Puzzle-Mesh, lo que le confiere una arquitectura modular y agnóstica a la infraestructura.
- **Jub:** Es una plataforma diseñada como un concentrador y distribuidor de datos, especializada en el monitoreo de fenómenos atmosféricos. Actúa como un punto central para la ingesta y distribución de flujos de datos en tiempo real.
- **MictlanX:** Es la plataforma de almacenamiento subyacente para el ecosistema. Funciona como un sistema de almacenamiento de objetos descentralizado, distribuido y respaldado por enrutadores. Este tipo de sistemas, a diferencia del almacenamiento de archivos tradicional, gestiona los datos como objetos discretos, cada uno con sus metadatos y un identificador único. Este enfoque facilita la escalabilidad masiva y la resiliencia, principios clave en sistemas de almacenamiento para la nube [18, 19]. El *middleware* de este proyecto interactuará directamente con el cliente Python de MictlanX para la persistencia de los artefactos.

2.6. Mecanismos para la Automatización del Middleware

Para que el *middleware* funcione como un sistema autónomo y reactivo, es crucial implementar mecanismos de automatización que le permitan operar sin intervención humana.

2.6.1. Servicios en Segundo Plano (Demonios)

En sistemas operativos derivados de UNIX, un **demonio** (del inglés *daemon*) es un proceso informático que se ejecuta en segundo plano (
textitbackground), sin una terminal de control asociada. Los demonios son el pilar para la implementación de servicios que deben estar siempre disponibles, como servidores web, bases de datos o, en el caso de este proyecto, un monitor de sistema de archivos. El ciclo de vida de un demonio (su inicio, parada y reinicio en caso de fallo) es gestionado por el sistema operativo a través de un sistema de *init* como `systemd` [20]. Implementar el *middleware* como un demonio es, por tanto, el enfoque arquitectónico natural para garantizar una vigilancia continua y desatendida del sistema de archivos de Nez [21].

2.6.2. Monitorización del Sistema de Archivos

El evento que debe disparar la lógica del *middleware* es la creación de un nuevo archivo en el directorio de salida de Nez. Existen dos enfoques técnicos para esta tarea:

- **Sondeo (Polling):** Este método consiste en verificar periódicamente (por ejemplo, cada segundo) el contenido de un directorio para ver si han ocurrido cambios. Aunque

es conceptualmente simple, es muy ineficiente, ya que consume ciclos de CPU y realiza operaciones de E/S de disco de forma constante, incluso cuando no hay ninguna actividad en el directorio [22].

- **Notificación por Eventos:** Los sistemas operativos modernos ofrecen APIs mucho más eficientes. En Linux, la API `inotify` permite que una aplicación se registre en el kernel para recibir notificaciones directas cuando ocurren eventos específicos en un directorio (como la creación, modificación o eliminación de un archivo). El kernel actúa como un observador y notifica a la aplicación de forma asíncrona, eliminando la necesidad de sondeos y reduciendo drásticamente el consumo de recursos [23].

Para este proyecto, se adopta el enfoque de notificación por eventos, que es superior en rendimiento. Se utilizará la biblioteca de Python `watchdog`, la cual proporciona una API unificada y multiplataforma que abstrae las complejidades de las APIs nativas como `inotify` (Linux), FSEvents (macOS) o ReadDirectoryChangesW (Windows), garantizando una solución portable y eficiente [24].

2.7. Tecnologías de Soporte para la Implementación

La construcción del *middleware* se apoya en un conjunto de tecnologías estándar en la industria del software, seleccionadas por su robustez, portabilidad, y el amplio soporte de sus ecosistemas.

2.7.1. Python como Lenguaje de Scripting

Python ha sido seleccionado como el lenguaje de implementación para el demonio del *middleware*. Las razones para esta elección son múltiples: su sintaxis limpia y legible facilita el desarrollo y el mantenimiento; su vasto ecosistema de bibliotecas de terceros incluye soporte nativo para todas las necesidades del proyecto (como el cliente de MictlanX y la biblioteca `watchdog`); y su naturaleza como lenguaje interpretado de alto nivel lo hace ideal para tareas de *scripting* y automatización de sistemas [25, 26].

2.7.2. Contenerización de Servicios con Docker

Para asegurar que el demonio del *middleware* sea portable y que su despliegue sea consistente y reproducible en cualquier entorno, se utiliza la tecnología de contenedores **Docker**. Un contenedor empaqueta una aplicación junto con todas sus dependencias (bibliotecas, binarios, archivos de configuración) en una imagen estandarizada y portable [27].

Un concepto crucial para esta implementación es el de los **volúmenes de Docker**. Por defecto, el sistema de archivos de un contenedor es efímero. Los volúmenes son el mecanismo que proporciona persistencia a los datos y, más importante aún, permiten compartir directorios entre el sistema anfitrión (
textithost) y un contenedor. En esta implementación, se utilizará un volumen para mapear el directorio de salida de Nez del sistema anfitrión al sistema de archivos interno del contenedor del *middleware*. Esto permite que el proceso `watchdog`, que se ejecuta dentro del contenedor,

pueda monitorear eventos que ocurren en el exterior, en el sistema de archivos del *host*, de manera transparente y eficiente [28].

2.7.3. Garantía de Integridad de Datos con SHA-256

Al transferir archivos entre sistemas, especialmente en un entorno distribuido, es vital asegurar que los datos no se corrompan durante la transmisión o el almacenamiento. El cliente de MictlanX, al igual que muchos sistemas de almacenamiento robustos, utiliza funciones *hash* criptográficas para garantizar la integridad de los datos de extremo a extremo. Específicamente, se emplea el algoritmo **SHA-256** (Secure Hash Algorithm 256-bit).

Este algoritmo, especificado en el estándar FIPS 180-4 del NIST (Instituto Nacional de Estándares y Tecnología de EE. UU.) [29], toma un bloque de datos de cualquier tamaño y produce una firma digital de longitud fija (256 bits) llamada *hash*. Esta firma es, para todos los efectos prácticos, única para ese conjunto de datos específico. Cualquier cambio en los datos, por mínimo que sea, resultará en un *hash* completamente diferente. El proceso de verificación es el siguiente: al subir un archivo, se calcula su *hash*; al descargarlo o recuperarlo, se vuelve a calcular y se compara con el original. Si ambos *hashes* coinciden, se tiene una garantía matemática muy alta de que el archivo no ha sido alterado o corrompido [30].

3. Sistema Propuesto

En esta sección se detalla el diseño y la arquitectura del *Nez-Daemon Watcher*, un servicio de software diseñado para funcionar como un **agente de ingesta de datos** para el **Espacio de Almacenamiento Virtual (VSS) de MictlanX**. El rol principal de este componente es actuar como una puerta de enlace robusta y desatendida entre un sistema de archivos local y el ecosistema de almacenamiento distribuido de MictlanX, centrándose en la eficiencia, la resiliencia y la automatización.

3.1. Análisis de Requerimientos

El análisis de requerimientos es la base sobre la cual se construye el sistema, definiendo las capacidades y restricciones del mismo.

3.1.1. Requerimientos Funcionales

Los requerimientos funcionales describen las operaciones principales que el sistema puede ejecutar para cumplir su misión como agente de ingesta.

- **RF-01: Monitoreo de Directorio.** El sistema debe monitorear de forma continua y recursiva un directorio predefinido (`watch_dir`). Esta es la principal fuente de eventos que dispara el flujo de trabajo de carga de archivos.
- **RF-02: Carga de Archivos a MictlanX.** Al detectar un nuevo archivo, el sistema debe orquestar su carga al VSS de MictlanX. Este proceso implica almacenar el archivo como un objeto atómico denominado **Ball** dentro de un contenedor lógico llamado **Bucket**. La carga debe incluir metadatos clave como el checksum SHA-256 (para integridad), el tamaño, y un factor de replicación (`replication_factor`) que instruye a MictlanX sobre cuántas copias del "Ball"deben distribuirse entre los nodos de almacenamiento (**Peers**) para garantizar la redundancia.
- **RF-03: Verificación de Existencia.** Para optimizar el uso de la red y el almacenamiento, el sistema debe verificar si un "Ballçon la misma clave ya existe en el "Bucket"de MictlanX antes de iniciar una carga, evitando así la duplicación de datos.
- **RF-04: Gestión de Cuarentena.** Para asegurar la alta disponibilidad, si un archivo falla repetidamente durante el proceso de carga, debe ser movido a un directorio de çuarentena", aislando los archivos problemáticos sin detener el servicio.
- **RF-05: Manejo de Solicitudes de Descarga.** El sistema debe ser capaz de procesar solicitudes de descarga desde MictlanX, actuando como un punto de acceso local al VSS. Los dos mecanismos soportados son a través de archivos `.mictlanx_download` y un socket de dominio Unix para comunicación entre procesos (IPC).

3.1.2. Requerimientos No Funcionales

Los requerimientos no funcionales definen las características de calidad y operativas que garantizan que el sistema sea eficiente y robusto.

- **RNF-01: Concurrencia.** El sistema debe procesar múltiples archivos de forma concurrente mediante un grupo de "trabajadores.ªsíncronos para maximizar el rendimiento.
- **RNF-02: Estabilidad de Archivo.** Antes de procesar un archivo, el sistema debe esperar un breve período para asegurar que la operación de escritura ha finalizado por completo, evitando así la ingesta de datos corruptos o incompletos.
- **RNF-03: Resiliencia.** Las operaciones de red deben ser resilientes a fallos transitorios, utilizando una estrategia de reintentos con retroceso exponencial para las llamadas a la API de MictlanX.
- **RNF-04: Configurabilidad.** Siguiendo la metodología de 12-Factores, la configuración del sistema (rutas, URI del Router de MictlanX, etc.) debe ser externa al código y gestionada mediante variables de entorno.
- **RNF-05: Portabilidad.** El sistema está empaquetado como una imagen Docker [27], garantizando un entorno de ejecución consistente y simplificando su despliegue.

3.2. El Ecosistema de Almacenamiento MictlanX

Para comprender el rol y el diseño del *Nez-Daemon Watcher*, es fundamental primero describir el sistema de almacenamiento con el que se integra. MictlanX no es un sistema de almacenamiento monolítico, sino un **Espacio de Almacenamiento Virtual (VSS)**, una abstracción programable sobre un conjunto de recursos de almacenamiento distribuidos. Su propósito es ofrecer una plataforma de almacenamiento flexible, elástica y resiliente.

La arquitectura de MictlanX, basada en el repositorio `mictlanx-service`, se compone de tres entidades principales que operan de forma desacoplada:

- **Storage Peers (Pares de Almacenamiento):** Son los nodos de almacenamiento individuales y autónomos. Su única responsabilidad es guardar los datos físicos que se les entregan. En la terminología de MictlanX, la unidad atómica de datos es un **Ball**.
- **Storage Replica Management (SPM):** Es el cerebro o la .ªutoridad de metadatos"del ecosistema. El SPM es un subsistema distribuido que mantiene un registro global del estado del clúster. Sabe qué Peers están activos, qué "Balls.exisTen, en qué "Peersreside cada réplica de un "Ball", y cómo están organizados lógicamente dentro de contenedores llamados **Buckets**.
- **Router (Enrutador):** Es el único punto de entrada y el orquestador de operaciones de entrada/salida (I/O) para todo el VSS. Los clientes, como el *Nez-Daemon Watcher*,

no interactúan directamente con los Peers o el SPM. Toda la comunicación se realiza a través de la API del Router.

Cuando el Watcher necesita cargar un archivo, se comunica con el Router. El Router, a su vez, consulta al SPM para tomar decisiones inteligentes sobre dónde y cómo almacenar el archivo (por ejemplo, seleccionando los Peers menos cargados y asegurando el factor de replicación solicitado). Una vez tomada la decisión, el Router guía al cliente para que complete la transferencia de datos. Esta arquitectura desacoplada, que separa la gestión de metadatos (el "plano de control"del SPM) de las operaciones de datos (el "plano de datos"del Router y los Peers), es lo que le da a MictlanX su escalabilidad y flexibilidad. El *Nez-Daemon Watcher* actúa, por tanto, como un cliente especializado que traduce eventos del sistema de archivos en operaciones complejas dentro de este ecosistema distribuido.

3.3. Arquitectura del Sistema

La arquitectura del *Nez-Daemon Watcher*, descrita en esta sección, está diseñada específicamente para funcionar como un componente de borde (edge component) eficiente y robusto para el VSS de MictlanX. Su diseño modular y orientado a eventos le permite abstraer la complejidad del ecosistema de almacenamiento y operar de forma autónoma. La Figura 1 ilustra la interacción entre sus componentes internos.

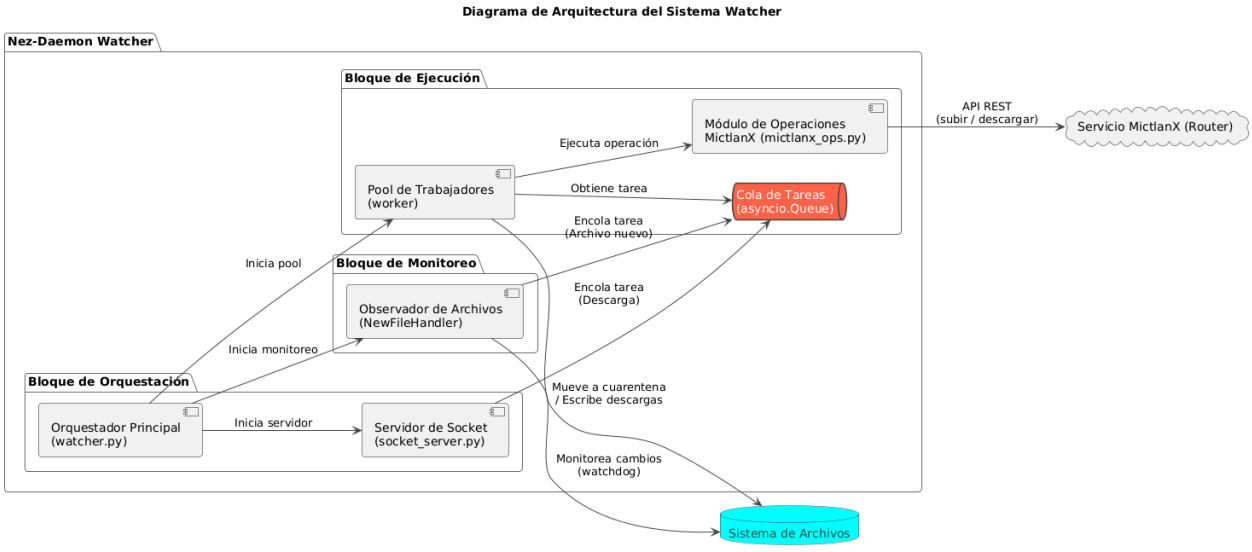


Figura 1: Diagrama de alto nivel de la arquitectura del Watcher.

3.3.1. Componentes del Watcher

- **Orquestador Principal** (`watcher.py`)**:** Inicia y supervisa todos los componentes. Realiza una comprobación de salud inicial contra el Router de MictlanX antes de empezar a operar.
- **Observador del Sistema de Archivos** (`NewFileHandler`)**:** Utiliza `watchdog` [24]

para detectar eventos y actúa como un puente hacia el bucle de eventos asíncrono de la aplicación.

- **Cola de Tareas (`asyncio.Queue`):** Búfer central que desacopla la detección de eventos del procesamiento, permitiendo al sistema gestionar picos de trabajo.
- **Trabajadores (`worker`):** Tareas asíncronas que consumen eventos de la cola y ejecutan la lógica de negocio.
- **Módulo de Operaciones MictlanX (`mictlanx_ops.py`):** Abstrae la comunicación con el Router de MictlanX. Su función más importante es implementar el proceso de carga en dos fases dictado por la arquitectura de MictlanX.
- **Servidor de Socket (`socket_server.py`):** Expone un socket de dominio Unix para recibir órdenes de otros procesos locales.

3.4. Diagrama de Contexto

El diagrama de contexto, mostrado en la Figura 2, posiciona al sistema como un servicio intermediario entre el entorno local y el VSS de MictlanX.

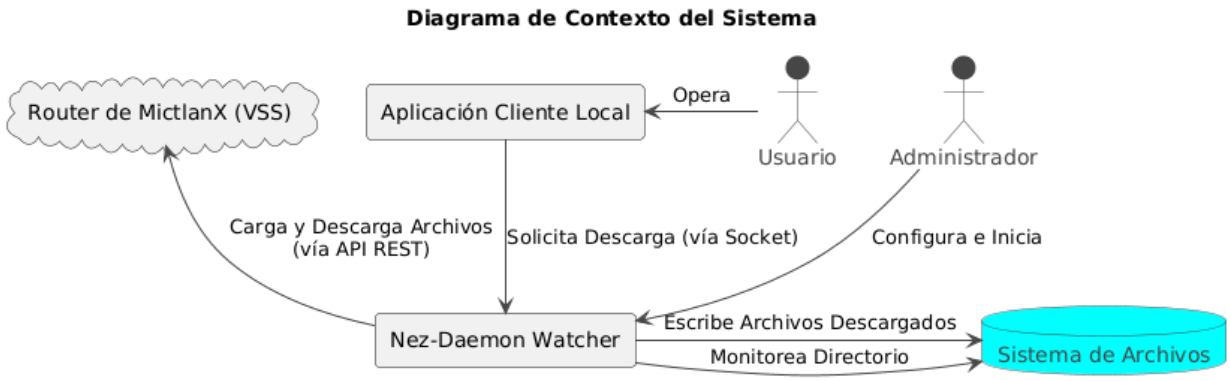


Figura 2: Diagrama de Contexto del Sistema Watcher.

Las entidades externas son:

- **Usuario/Administrador:** Inicia y configura el servicio.
- **Sistema de Archivos:** El entorno local que el watcher monitorea.
- **Router de MictlanX (VSS):** El punto de entrada al sistema de almacenamiento distribuido, con el que se comunica vía API REST.
- **Aplicación Cliente Local:** Proceso local que solicita descargas vía socket.

3.5. Diagramas de Casos de Uso

Los casos de uso definen las interacciones funcionales clave. La Figura 3 muestra su representación gráfica.

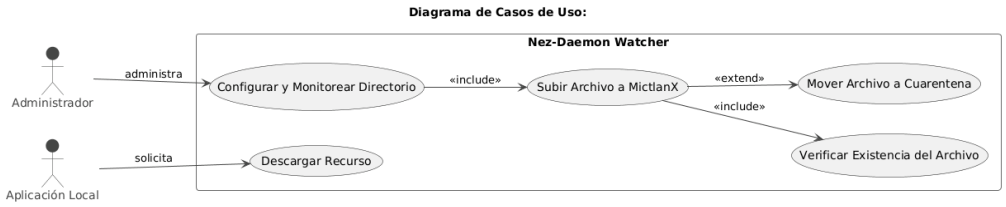


Figura 3: Diagrama de Casos de Uso.

3.6. Diagramas UML

El diagrama de secuencia en la Figura 4 detalla el flujo de la carga de un archivo, evidenciando la comunicación exclusiva con el Router de MictlanX.

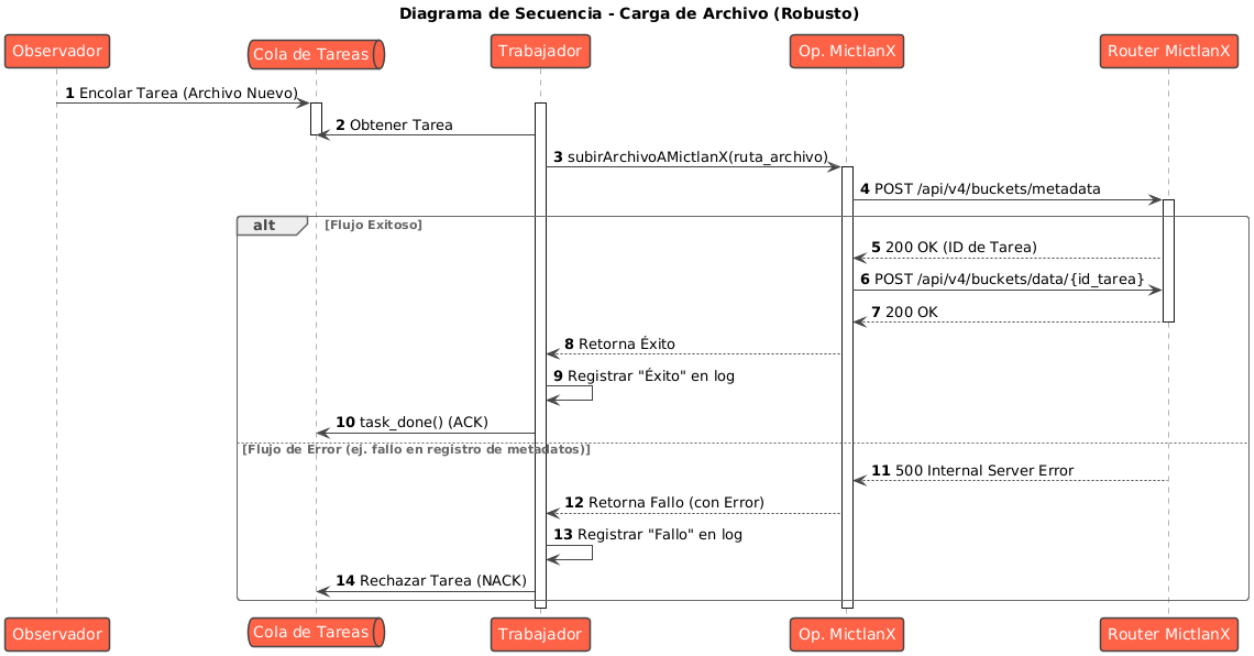


Figura 4: Diagrama de Secuencia para la carga de un nuevo archivo.

La secuencia es:

1. El `Observador` detecta un archivo y encola una tarea.
2. Un `Trabajador` toma la tarea.
3. El `Trabajador` invoca al módulo de `Operaciones MictlanX`.

4. El módulo de operaciones envía una petición de registro de metadatos al `Router de MictlanX`. El Router consulta al SPM y devuelve un ID de tarea.

5. El módulo de operaciones envía el contenido del archivo al `Router`, usando el ID de tarea. El Router gestiona la transferencia a los Peers apropiados.

3.7. Diseño de Entradas y Salidas

Las entradas y salidas del sistema se definen a nivel de datos y protocolos.

3.7.1. Entradas del Sistema

- **Archivos en Directorio:** Archivos binarios de cualquier tipo.
- **Mensajes de Socket:** Cadenas de texto UTF-8 con la ruta del recurso a descargar.
- **Variables de Entorno:** Pares clave-valor para la configuración.

3.7.2. Salidas del Sistema

- **Llamadas a la API del Router de MictlanX:** Peticiones HTTP POST a los endpoints `/api/v4/buckets/metadata` y `/api/v4/buckets/data/{task_id}`.
- **Archivos en Cuarentena:** Archivos movidos al directorio de cuarentena.
- **Registros de Actividad (Logs):** Salida de texto estructurada en la consola.
- **Archivos Descargados:** Archivos escritos en el disco local.

3.8. Justificación de Decisiones de Diseño

La arquitectura y tecnología de un proyecto son el resultado de una serie de decisiones deliberadas para resolver un problema específico. A continuación, se justifican las elecciones más importantes en el desarrollo del *Nez-Daemon Watcher*.

3.8.1. Elección de Python y Asyncio

Para un servicio cuya principal función es esperar y gestionar operaciones de entrada/salida (I/O) —como monitorear el sistema de archivos y realizar peticiones de red—, un modelo de concurrencia asíncrono es ideal. Python, a través de su ecosistema `asyncio`, permite manejar miles de operaciones concurrentes con un solo hilo, evitando la sobrecarga de memoria y conmutación de contexto asociada al multithreading tradicional. Esto se traduce en un sistema altamente eficiente y escalable para tareas I/O-bound.

3.8.2. Uso de una Arquitectura Basada en Cola de Tareas

La decisión de usar una cola de tareas (`asyncio.Queue`) como intermediario entre los productores de eventos y los consumidores (trabajadores) es fundamental para la robustez del sistema. Esta arquitectura desacoplada ofrece varias ventajas:

- **Absorción de Picos:** Si se crean cientos de archivos simultáneamente, la cola los almacena de forma segura, permitiendo que los trabajadores los procesen a su propio ritmo sin perder ningún evento.
- **Resiliencia:** Si el proceso de un trabajador falla, solo afecta a la tarea actual. El resto de las tareas permanecen seguras en la cola, listas para ser procesadas.
- **Escalabilidad:** Es fácil aumentar la capacidad del sistema simplemente añadiendo más trabajadores que consuman de la misma cola.

3.8.3. Elección de Socket de Dominio Unix para IPC

Para la comunicación entre procesos (IPC) en la misma máquina, se eligió un socket de dominio Unix en lugar de un socket TCP/IP sobre localhost. Esta decisión se basa en que los sockets Unix son más eficientes y seguros: no tienen la sobrecarga del stack de red TCP/IP (handshakes, cabeceras, etc.) y su acceso puede ser controlado mediante los permisos del sistema de archivos, limitando qué procesos pueden comunicarse con el watcher.

3.9. Seguridad del Sistema

Aunque el watcher opera como un servicio de backend, se implementaron varias medidas para asegurar su operación.

- **Comunicación Segura:** Se asume que la comunicación con el Router de MictlanX se realiza a través de HTTPS, protegiendo los datos en tránsito. La biblioteca `httpx` utilizada soporta TLS de forma nativa.
- **Gestión de la Configuración:** Siguiendo las mejores prácticas, no se almacenan secretos (como tokens de API) en el código. La dirección y posibles credenciales para MictlanX se gestionan a través de variables de entorno, separando la configuración del código.
- **Sanitización de Claves:** La función `sanitize_key` elimina caracteres no alfanuméricos de las rutas de los archivos antes de usarlas como claves en MictlanX. Esto previene ataques de "path traversalz asegura que las claves sean siempre válidas para la API de MictlanX.
- **Permisos del Sistema de Archivos:** El servicio opera con los mínimos privilegios necesarios. Solo requiere permisos de lectura sobre el directorio vigilado y permisos de escritura sobre los directorios de descarga y de cuarentena.

3.10. Configuración y Despliegue

La flexibilidad del sistema se basa en su configuración externa y su empaquetado para un despliegue sencillo.

3.10.1. Variables de Entorno

El comportamiento del watcher se controla mediante las siguientes variables de entorno:

- `LOG_LEVEL`: Define el nivel de detalle de los logs (ej. INFO, DEBUG).
- `WATCH_DIRECTORY`: Ruta absoluta al directorio que será monitoreado.
- `QUARANTINE_DIRECTORY`: Ruta absoluta al directorio donde se moverán los archivos fallidos.
- `MICTLANX_URI`: El URI completo para conectarse al Router de MictlanX, incluyendo protocolo, host, puerto y credenciales si son necesarias.
- `BUCKET_ID`: El nombre del "Bucket.[en] MictlanX donde se almacenarán los archivos.
- `REPLICATION_FACTOR`: Número de réplicas que MictlanX debe crear para cada archivo.
- `MAX_WORKERS`: Número de trabajadores concurrentes para procesar la cola de tareas.

3.10.2. Despliegue con Docker

El ‘Dockerfile‘ del proyecto define el proceso de empaquetado. Los pasos más relevantes son:

1. Se utiliza una imagen base oficial de Python (`python:3.12-slim`) para minimizar el tamaño.
2. Se copia y se instala primero la dependencia local `mictlanx-client`, ya que es un prerrequisito.
3. Se instalan el resto de las dependencias listadas en `requirements.txt`.
4. Finalmente, se copia el código fuente del watcher y se define el comando de inicio (`CMD`).

Para ejecutar el servicio, se utilizaría un comando `docker run` que monte los directorios locales como volúmenes y pase la configuración a través de variables de entorno.

3.11. Manejo de Errores y Resiliencia

Un aspecto clave del diseño es la capacidad del sistema para operar de forma desatendida y recuperarse de problemas comunes.

- **Comprobación de Salud Inicial:** Antes de iniciar el bucle principal, el watcher realiza una llamada de "health check.ªl Router de MictlanX. Si el servicio de almacenamiento no está disponible, el watcher termina con un error, evitando operar en un estado degradado.
- **Reintentos con Retroceso Exponencial:** Todas las llamadas a la API de MictlanX están envueltas en una lógica de reintentos automáticos. Si una llamada falla por un problema de red transitorio, el sistema esperará un tiempo (que aumenta exponencialmente con cada fallo) antes de reintentar, hasta un máximo de intentos configurado.
- **Mecanismo de Cuarentena:** Si un archivo falla todos los reintentos, se considera "venenoso"(poison pill). En lugar de bloquear la cola para siempre, el archivo es movido a un directorio de cuarentena. Esto libera al sistema para que continúe procesando otras tareas y permite que un administrador investigue el problema de forma manual.

3.12. Algoritmos Principales

A continuación se describen los algoritmos que definen el flujo de control y la lógica de negocio del Watcher.

3.12.1. Algoritmo del Trabajador Principal

Este algoritmo representa el bucle de eventos de cada trabajador, responsable de consumir tareas y despacharlas a la lógica apropiada.

3.12.2. Justificación y Utilidad del Patrón Worker

Este algoritmo, basado en el patrón de productor-consumidor, es el motor que impulsa toda la lógica de negocio del *Nez-Daemon Watcher*. Su diseño responde a la necesidad de gestionar eventos que ocurren en momentos impredecibles y concurrentes, como la creación de archivos o las peticiones por socket. Un diseño secuencial simple se bloquearía con la primera tarea de larga duración, volviéndose incapaz de responder a nuevos eventos. La solución implementada resuelve esto mediante un bucle de eventos con una cola de tareas. El bucle infinito (‘While el sistema está activo‘) establece al trabajador como un proceso persistente, siempre listo para operar. El punto más crítico es la línea ‘tarea <- esperar y obtener de ColaDeTareas‘, donde la cola actúa como un búfer que absorbe los picos de trabajo y desacopla la detección de eventos de su procesamiento. Los productores (el observador de archivos y el servidor de socket) simplemente añaden tareas a la cola sin esperar a que se completen. El bloque ‘If/ElsIf‘ funciona como un enrutador que despacha cada tipo de tarea a la lógica correspondiente, permitiendo que el mismo pool de trabajadores maneje distintas operaciones. Al ejecutar múltiples instancias de este algoritmo en paralelo, el sistema logra la concurrencia, maximizando el uso de recursos. Este patrón transforma un flujo de eventos caótico en un proceso de trabajo ordenado, concurrente y robusto.

Algoritmo 1 Procesamiento de Tareas del Trabajador

```
1:  while el sistema está activo do
2:      tarea ← esperar y obtener de ColaDeTareas
3:      if tarea es un evento de archivo then
4:          ruta_archivo ← obtener ruta de tarea
5:          if ruta_archivo es una solicitud de descarga then
6:              procesarSolicitudDeDescarga(ruta_archivo)
7:          else
8:              esperarEstabilidadDeArchivo(ruta_archivo)
9:              if archivoYaExisteEnMictlanX(ruta_archivo) then
10:                 registrar .^Archivo ya existe, omitiendo."
11:                 continue
12:             end if
13:             if subirArchivoAMictlanX(ruta_archivo) es exitoso then
14:                 registrar .^Archivo subido con éxito."
15:             else
16:                 registrar "Fallo en la carga del archivo."
17:                 moverArchivoACuarentena(ruta_archivo)
18:             end if
19:         end if
20:     else if tarea es una solicitud de socket then
21:         ruta_descarga ← obtener ruta de tarea
22:         procesarDescargaDesdeSocket(ruta_descarga)
23:     end if
24: end while
```

3.12.3. Algoritmo de Carga de Archivo a MictlanX

Este algoritmo es crucial, pues su diseño de dos pasos es una consecuencia directa de la arquitectura desacoplada de MictlanX. La separación del registro de metadatos y la carga de datos permite al Router coordinar la replicación y preparar a los Peers antes de recibir el contenido, aumentando la escalabilidad y robustez del VSS.

Algoritmo 2 Carga de Archivo a MictlanX

```
1:  procedure SUBIRARCHIVOAMICTLANX(ruta_archivo)
2:      checksum ← CALCULARSHA256(ruta_archivo)
3:      tamaño ← OBTENERTAMAÑO(ruta_archivo)
4:      clave ← SANITIZARRUTARELATIVA(ruta_archivo)
5:      metadatos ← crear estructura con bucket_id, clave, checksum, tamaño,
    replication_factor
6:      // Paso 1: Registrar metadatos en el Router
7:      respuesta_meta ← POST(/api/v4/buckets/metadata", metadatos)
8:      if respuesta_meta es un error then
9:          return falso
10:     end if
11:     id_tarea ← OBTENERID(respuesta_meta)
12:     // Paso 2: Subir contenido de datos al Router usando el ID de tarea
13:     respuesta_datos ← POST(/api/v4/buckets/data/id_tarea", ruta_archivo)
14:     if respuesta_datos es un error then
15:         return falso
16:     end if
17:     return verdadero
```

3.12.4. Justificación y Utilidad del Proceso de Carga en Dos Pasos

A primera vista, el algoritmo para cargar un archivo podría parecer innecesariamente complejo, ya que una simple petición HTTP PUT con el contenido del archivo parecería suficiente. Sin embargo, este diseño de dos pasos (primero metadatos, luego datos) es una decisión deliberada y crucial que resuelve problemas fundamentales inherentes a los sistemas de almacenamiento distribuido como MictlanX. El problema principal a resolver es cómo cargar un archivo de forma fiable a un sistema complejo donde el destino final no se conoce de antemano y los errores de red son comunes. La solución implementada es un protocolo que separa la "negociación"(plano de control) de la "transferencia"(plano de datos). La primera fase, el registro de metadatos, actúa como el plano de control. La llamada a la API (‘POST /api/v4/buckets/metadata‘) no transfiere el archivo, sino que comunica la *intención* de subirlo con ciertas características (nombre, tamaño, checksum, factor de replicación). Esta negociación permite al Router de MictlanX realizar operaciones críticas de forma anticipada, como la validación de permisos y la planificación del almacenamiento, donde consulta al SPM para seleccionar los Peers más adecuados. Una ventaja clave de este paso es la capacidad de **fallar rápido (fail-fast)**. Si alguna condición previa no se cumple, el Router devuelve un error inmediato, y el Watcher evita el coste de transferir un archivo que sería rechazado, ahorrando tiempo y ancho de banda. La segunda fase, la carga de datos, representa el

plano de datos. Solo si el primer paso tiene éxito, el Router devuelve un ÏD de tarea"que representa una transacción de carga aprobada. La segunda llamada a la API (‘POST /api/v4/buckets/data/{id_tarea}‘) utiliza este identificador para transferir el contenido binario. Este enfoque dota al proceso de una atomicidad simplificada, ya que los datos solo se envían después de que el sistema ha confirmado formalmente que está listo para recibirlos, previniendo estados inconsistentes. Además, esta separación de responsabilidades es lo que permite la escalabilidad del ecosistema MictlanX, optimizando la gestión de metadatos y la transferencia de datos de forma independiente.

Índice de figuras

Índice de cuadros

Índice de algoritmos

Referencias

[1] Martin Sudmanns et al. «Big Earth Observation Data-A Review of the Challenges and Opportunities for Big Data Processing in the Copernicus Era». En: *Remote Sensing* 11.21 (2019), pág. 2577. DOI: 10.3390/rs11212577.

[2] Geert Litjens et al. «A survey on deep learning in medical image analysis». En: *Medical Image Analysis* 42 (2017), págs. 60-88. DOI: 10.1016/j.media.2017.07.005.

[3] Kennedy O. Ondimu y Geoffrey M. Muketha. «Challenges in Achieving Interoperability in Distributed Systems: a Survey of Literature». En: *International Journal of Computer Applications* 168.10 (2017), págs. 33-39. DOI: 10.5120/ijca2017914539.

[4] Carole A. Goble y David De Roure. «Scientific workflow systems–for whom and for what?» En: *Journal of Grid Computing* 5.3 (2007), págs. 249-255. DOI: 10.1007/s10723-007-9078-8.

[5] J. L. González-Compeán et al. *Muyal-Nez, una plataforma de construcción de sistemas de ciencias de datos médicos para procesos de toma de decisiones en el sector salud.* 2020. URL: https://contactos.izt.uam.mx/index.php/contactos/article/view/297/169.

[6] Lee Calcote y Zack Butcher. *Istio: Up and Running: Using a Service Mesh to Connect, Secure, Control, and Observe*. Sebastopol, CA, USA: O'Reilly Media, Inc., 2019.

[7] Andrew S. Tanenbaum y Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Pearson Education, 2014.

[8] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2021.

[9] Chris Richardson. *Microservices Patterns: With Examples in Java*. Manning Publications, 2018.

[10] Martin Fowler. *Microservices*. Accedido: 27-10-2023. 2014. URL: https://martinfowler.com/articles/microservices.html.

[11] Pierre Gancarski y Luc Bouge. «A survey of service mesh solutions». En: *arXiv preprint arXiv:2104.06406* (2021).

[12] Gregor Hohpe y Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2003.

[13] Dante D Sánchez-Gallegos et al. «An efficient pattern-based approach for workflow supporting large-scale science: The DagOnStar experience». En: *Future Generation Computer Systems* 122 (2021), págs. 187-203.

[14] Dante D Sánchez-Gallegos et al. «PuzzleMesh: A puzzle model to build mesh of agnostic services for edge-fog-cloud». En: *IEEE Transactions on Services Computing* (2022).

[15] E. G. R. Communications. *Middleware for Communications*. Wiley, 2008.

[16] Philip A Bernstein. «Middleware: a model for distributed system services». En: *Communications of the ACM* 39.2 (1996), págs. 86-98.

[17] Charity Majors, Liz Fong-Jones y George Miranda. *Observability Engineering: Achieving Production Excellence*. O'Reilly Media, 2022.

[18] Sebastian Pape, Christoph Mezger y Bernd Freisleben. «Object storage on anthill: a client-side approach to decentralization». En: *SIGOPS Oper. Syst. Rev.* 45.2 (2011), págs. 72-76.

[19] Martin Kleppmann. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. O'Reilly Media, 2017.

[20] W. Richard Stevens y Stephen A. Rago. *Advanced programming in the UNIX environment*. Addison-Wesley, 2013.

[21] James Sanders. *Daemons in Linux*. Accedido: 27-10-2023. 2005. URL: https://www.linuxjournal.com/article/8042.

[22] Michael Kerrisk. *The Linux programming interface: a Linux and UNIX system programming handbook*. No Starch Press, 2010.

[23] Robert Love. «Inotify: A new kernel filesystem monitoring API». En: *Linux Journal* 2005.135 (2005), pág. 2.

[24] The Watchdog Authors. *Watchdog: Python API and shell utilities to monitor file system events*. Accedido: 27-10-2023. 2023. URL: https://python-watchdog.readthedocs.io/en/stable/.

[25] Python Software Foundation. *Python 3.12.0 documentation*. Accedido: 27-10-2023. 2023. URL: https://docs.python.org/3/.

[26] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Accedido: 27-10-2023. 2021. URL: https://martinfowler.com/books/refactoring.html.

[27] Dirk Merkel. «Docker: lightweight linux containers for consistent development and deployment». En: *Linux journal* 2014.239 (2014), pág. 2.

[28] Nigel Poulton. *Docker Deep Dive*. Nigel Poulton, 2020.

[29] National Institute of Standards and Technology. *FIPS PUB 180-4: Secure Hash Standard (SHS)*. 2015. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf.

[30] William Stallings. *Cryptography and network security: principles and practice*. Pearson, 2017.