



DEPARTAMENTO DE ELECTROTECNIA E INFORMÁTICA
INGENIERIA EN INFORMÁTICA

Implementación y despliegue de un asistente virtual especializado en manuales técnicos para operaciones industriales

Yangzi Álvarez Gajardo
yangzi.alvarez@sansano.usm.cl

Gabriel Jara
Profesor Guía

Patricio Santander
Profesor Correferente

Resumen: En la industria de maquinarias, existe una gran cantidad de manuales técnicos, los cuales suelen ser extensos y complejos, generando tiempos de respuestas lentos en terreno e incluso detención de la maquinaria, lo que se traduce en pérdidas económicas. Para enfrentar esta problemática se propone un asistente virtual capaz de responder consultas sobre manuales y ofrecer trazabilidad de estas. A pesar de la situación favorable que presenta el asistente, la implementación inicial enfrentó un problema crítico; al digitalizar la información de manuales extensos, el sistema se bloquea, impidiendo su uso. Este trabajo contribuye a la evolución técnica de este proyecto, donde se rediseñó la arquitectura y se refactorizó en microservicios híbridos. Esto permite separar el servicio de digitalizar manuales, que consume muchos recursos, del servicio de consultas, garantizando la disponibilidad del chat, incluso mientras se cargan manuales, asegurando así su viabilidad en entornos industriales reales.

Palabras Clave: Inteligencia Artificial, RAG, Microservicios Híbridos, Docker, Refactorización por Encapsulación.



CONSTANCIA DE VALIDACIÓN Y CONFIDENCIALIDAD DE MONOGRAFÍA A REPOSITORIO ACADÉMICO

1.- IDENTIFICACIÓN DEL TRABAJO ACADÉMICO

Tipo de monografía (marcar una opción): ☒ Memoria o trabajo de título ☐ Tesis de Postgrado

Título del trabajo: Implementación y despliegue de un asistente virtual especializado en manuales técnicos para operaciones industriales

Nombre del candidato(a): Yangzi Álvarez Gajardo

Carrera / Grado: Ingeniería en Informática

Campus: Viña del mar

Departamento: Electrotecnia e Informática

2.- VALIDACIÓN DEL PROFESOR GUÍA/DIRECTOR DE TESIS

Yo, Gabriel Jara Bulnes, en mi calidad de profesor(a) guía/director(a) del trabajo académico mencionado anteriormente **DEJO CONSTANCIA** que:

- He revisado esta versión del documento y corresponde a la versión final aprobada del trabajo.
- El trabajo cumple con los requisitos académicos y de formato establecidos por la institución.

3.- EVALUACIÓN DE CONFIDENCIALIDAD POR PROPIEDAD INDUSTRIAL (marcar una opción)

☒ El trabajo **NO contiene** información que amerite confidencialidad y puede ser publicado de inmediato en repositorio con acceso abierto.

☐ El trabajo **CONTIENE** información con potenciales implicancias de propiedad industrial o intelectual y requiere un periodo de confidencialidad (**embargo**) por (marcar una opción):

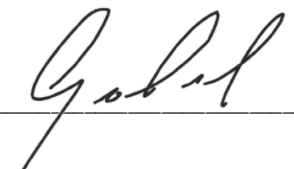
☐ 6 meses ☐ 12 meses ☐ 2 años ☐ 3 años ☐ 5 años ☐ 10 años

Fundamentación de la necesidad de confidencialidad (obligatorio si se solicita embargo):

4.- FIRMAS

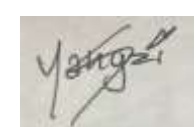
Profesor(a) guía o director(a) de memoria o tesis:

Fecha: 14-10-2026

Firma: 

Estudiante o Candidato(a):

Fecha: 14 de Octubre

Firma: 

Este formulario debe ser insertado como página 2 de la memoria o tesis, completado y firmado por estudiante y profesor(a) antes de la entrega en portal PRISMA de Biblioteca USM.



1 INTRODUCCIÓN

1.1 Contexto y Motivación

En las empresas donde se gestionan operaciones industriales el mantenimiento preventivo y correctivo de la maquinaria depende principalmente del acceso a la información técnica que se encuentra en los manuales: de partes, de uso y de manutención que entrega el fabricante. Estos documentos suelen ser extensos y complejos, ya que utilizan un lenguaje técnico especializado, incluso encontrándose en idiomas diferentes al del operador. Cuando una máquina presenta una detención no programada, los operarios enfrentan una barrera crítica: la búsqueda de información específica dentro de manuales de cientos de páginas PDF es lenta e ineficiente. Este dificultoso acceso a la información crítica se traduce en tiempos de inactividad prolongados, pérdidas económicas y posibles errores humanos por la falta de soporte inmediato.

Inicialmente el asistente virtual, que fue llamado proyecto ATLAS, era un sistema monolito que contenía todos los servicios en el mismo contenedor, servicio de ingesta de documentos y servicio de consultas, donde se identificó un problema crítico: ambos servicios no funcionan simultáneamente, ya que el servicio de ingesta consume gran cantidad de recursos y bloquea los demás servicios mientras se están subiendo documentos.

1.2 Definición del Problema Técnico

La versión inicial del proyecto ATLAS, se desarrolló bajo una arquitectura monolítica, donde todos los servicios coexistían en un único contenedor del backend. Esta decisión de diseño se consideró para la facilidad del desarrollo temprano del proyecto, donde eventualmente se evidenció un problema crítico al someter el sistema a pruebas de carga trabajo real: la incapacidad de ejecutar simultáneamente los servicios de ingesta y de consultas. Esto es un problema de arquitectura y no de funcionalidad, cada servicio ejecutado por si mismo funciona correctamente, es cuando se intentan utilizar ambos servicios que el sistema no está disponible.

El servicio de ingesta es el que más recursos demanda, ya que su función es procesar los documentos y generar vectores, es una tarea intensiva en recursos computacionales, tanto de CPU como de memoria RAM. Por otro lado, el servicio de consultas es, en comparación, mucho más liviano, por lo que es el servicio de ingesta el cual saturaba por completo el sistema. Es posible ver este problema cuando en las pruebas se sube un manual extenso (superior a 200 páginas), donde el sistema dejaba de responder las consultas, devolviendo errores del tipo 504 (Gateway Timeout) en la interfaz. Por esto, en el proceso de ingesta de este tipo de documentos, se compromete el servicio de consultas, invalidando el uso en entornos operativos reales, ya que la parte crítica del sistema es el asistente virtual, no la carga de documentos a este.

1.3 Solución Propuesta

Para superar estas limitaciones que se encontraron, se propone refactorizar la arquitectura del proyecto ATLAS, el cual incluye los servicios previamente mencionados: un servicio de consultas basado en generación aumentada por recuperación (RAG), un servicio de ingesta

de manuales con procesamiento de imágenes (OCR) y vectorización, además de la trazabilidad de los chat y las autenticaciones para acceder a la página. Este trabajo se enfoca en refactorizar este sistema en una arquitectura basada en microservicios híbridos. Esta arquitectura desacopla los servicios, separando en contenedores diferentes: uno para el proceso más pesado de ingesta de documentos donde se extrae el texto y se vectoriza, otro para el proceso liviano de consultas donde se generan las respuestas a partir de los vectores. Estos servicios se orquestan mediante Docker Compose y comparten volúmenes de persistencia, tanto el índice vectorial y la base de datos, logrando con esto un aislamiento de recursos sin la complejidad operativa de tener una sincronización de los datos distribuidos. La transición desde la arquitectura inicial monolítica se realiza con una estrategia de refactorización por encapsulamiento.

1.4 Objetivos

1.4.1 Objetivo General

Como objetivo general planteado para el proyecto ATLAS tenemos: "Implementar y desplegar un asistente virtual especializado en manuales técnicos de maquinaria industrial, que permita agilizar el soporte técnico, reduciendo los tiempos de detención de producción y mejorando la autonomía de operarios y técnicos mediante respuestas rápidas, precisas y trazables". Al momento de mencionar los objetivos de este trabajo en específico, se resumen como: "Desplegar y validar una arquitectura de microservicios híbridos para el asistente virtual ATLAS que garantice la capacidad de responder consultas de manera fluida, sin bloqueos operativos ni pérdidas de disponibilidad durante la ingesta de documentos".

1.4.2 Objetivos Específicos

- Establecer la línea base, generando contenedores para la versión monolítica desde la cual se trabaja.
- Diseñar arquitectura de microservicios híbridos para desacoplar el módulo de ingesta del módulo de consultas, conectando mediante APIs.
- Implementar la refactorización creando wrappers de API alrededor del código existente y generar contenedores Docker especializados para cada servicio.
- Configurar la orquestación, persistencia compartida y seguridad mediante Docker Compose, desde el cual gestionar volúmenes para el índice vectorial FAISS y la base de datos SQLite.
- Validar la arquitectura mediante pruebas proyectadas, comparando la disponibilidad del sistema y el consumo de recursos entre la arquitectura monolítica y la refactorización en microservicios híbridos.

1.5 Alcances y Limitaciones

El alcance del proyecto ATLAS incluye el procesamiento de manuales técnicos en formato PDF, ya sea texto nativo o en imágenes con texto, con soporte para múltiples idiomas, incluye autenticación básica para acceder a todos los servicios, incluye el servicio de consultas que mediante el uso de un índice vectorial conecta con un modelo LLM para generar la respuesta aumentada (usando RAG), además el sistema está enfocado en el despliegue en una infraestructura local mediante Docker Compose. Como limitaciones se encuentra que no se



pueden procesar aún planos de ingeniería complejos en los cuales se necesite interpretación, asimismo con diagramas complejos de mecánica.

1.6 Estructura del Documento

El presente documento se estructura según los siguientes capítulos:

- **Capítulo 1:** Describe el contexto del problema operativo y técnico, se establece la motivación, la solución propuesta y los objetivos principales del proyecto.
 - **Capítulo 2:** Proporciona los fundamentos teóricos necesarios para comprender el funcionamiento del sistema.
 - **Capítulo 3:** Presenta un análisis de las soluciones actuales en la industria, sus limitaciones y justifica el proyecto como solución.
 - **Capítulo 4:** Detalla la evolución arquitectónica del sistema, explicando el rediseño desde el monolito hacia los microservicios híbridos.
 - **Capítulo 5:** Explica la estrategia técnica utilizada para refactorizar el sistema profundizando en la encapsulación del código existente.
 - **Capítulo 6:** Define la metodología empleada para validar la nueva arquitectura, detallando las pruebas y documentando resultados.
 - **Capítulo 7:** Expone las conclusiones sobre el impacto de la refactorización, evaluando el cumplimiento de los objetivos y planteando trabajo futuro.
-

2 MARCO CONCEPTUAL

Se presenta a continuación los conceptos clave y las tecnologías más importantes para el desarrollo de este trabajo.

2.1 Fundamentos de Inteligencia Artificial

La inteligencia artificial (IA) es el pilar base de la solución propuesta, puesto que posee las capacidades de procesar, interpretar y generar información útil. En este contexto, se enfoca principalmente en las ramas de la IA donde se interactúa entre hombre y máquina mediante lenguaje natural y el procesamiento de texto extenso y no estructurado. A continuación se definen los conceptos clave que aportan esto.

2.1.1 Procesamiento de Lenguaje Natural (NLP)

Campo de la IA que permite que las máquinas puedan interpretar y generar lenguaje humano[1]. En el contexto del proyecto ATLAS, el procesamiento de lenguaje natural juega un rol fundamental, otorgando al sistema la capacidad de procesar textos extensos y altamente técnicos. Gracias al NLP el asistente no sólo realiza búsquedas literales, ya que comprende la semántica de la consulta y esto permite la extracción y recuperación precisa de la información no estructurada. Esto es vital en el asistente virtual.



2.1.2 Modelos de Lenguaje Grande (LLM)

Modelos de IA entrenados con una gran cantidad de volúmenes de texto, pueden generar respuestas coherentes y contextuales a partir de una entrada por lenguaje natural[2]. Estos modelos utiliza arquitecturas basadas en Transformers[3], que permite manejar dependencias de largo plazo en el texto.

2.2 Procesamiento y Gestión de Documentos

La gestión de documentos en entornos industriales enfrenta el desafío de digitalizar grandes volúmenes de información, la cual suele estar en formatos no editables como PDF o imágenes escaneadas. Para que el asistente virtual pueda acceder y utilizar esta información, es necesario transformar los documentos no estructurados en un formato utilizable mediante procesos de digitalización, extracción y representación numérica.

2.2.1 Reconocimiento Óptico de Caracteres (OCR)

Tecnología para convertir imágenes de texto en datos textuales[4]. Esto es vital para procesar manuales y convertirlos en una base de conocimientos. Existen herramientas como Tesseract[5], las cuales destacan en el campo por su precisión y soporte multi idioma.

2.2.2 Bases de Datos Vectoriales y Embeddings

Una vez que se extrae el texto, este debe ser indexado para su búsqueda. Los *embeddings* son representaciones vectoriales del texto, las cuales capturan su significado semántico. Las bases de datos vectoriales, por ejemplo FAISS[6] que se utiliza en este proyecto, almacenan estos vectores y permiten realizar búsquedas por similitud semántica, una gran ventaja comparado con búsquedas por palabras clave.

2.3 Arquitectura RAG (Retrieval-Augmented Generation)

Arquitectura que combina recuperación de información con generación de respuestas[7]. Primero revisa la base de conocimientos en busca de fragmentos relevantes y estos se usan como contexto en la consulta a través de un LLM, generando respuestas precisas y trazables. Esta arquitectura reduce significativamente lo que se conoce como “alucinaciones” del modelo.

2.4 Ingeniería de Software y Despliegue

El despliegue de soluciones de IA en entornos productivos requiere una arquitectura de software robusta que soporte cargas de trabajo intensivas y garantice la disponibilidad del servicio. Más que los algoritmos de IA, es necesario definir el diseño para poder aislar y orquestar los servicios del sistema para que este funcione de manera eficiente en una infraestructura local.

2.4.1 Contenedores Docker

Tecnología para empaquetar aplicaciones y sus dependencias en contenedores aislados, garantizando consistencia entre los entornos de desarrollo y producción[8]. Para la orquestación de múltiples contenedores se utiliza, en este caso, Docker Compose, una herramienta que permite definir y ejecutar aplicaciones multi-contenedor configurando redes internas seguras. Un pilar fundamental de esta tecnología en este proyecto es el uso de volúmenes compartidos, los cuales permiten que el almacenamiento de datos, tanto la base de datos relacional como los índices vectoriales, sea persistente, manteniendo a salvo de los ciclos de vida de los contenedores y permitiendo que distintos contenedores compartan la misma información.

2.4.2 Arquitectura de Microservicios Híbridos

Arquitectura adoptada en esta tesina para equilibrar rendimiento y simplicidad[9], como puntos a destacar:

- **Aislamiento a nivel de ejecución:** Los servicios se ejecutan en contenedores Docker aislados, evitando problemas de consumo de recursos para mantener la disponibilidad.
- **Acoplamiento a nivel de datos:** Los servicios comparten volúmenes de almacenamiento persistente, tanto para el índice vectorial como la base de datos relacional. Esto está pensado para evitar la complejidad operativa de mantener consistencia de un sistema puro de microservicios.

2.5 Patrón Wrapper

El patrón wrapper o de envoltorios es un patrón de diseño de ingeniería de software[18] que permite envolver una lógica de negocio antigua o acoplada para exponerlas a través de una nueva interfaz. En el contexto de la refactorización de sistemas, este patrón es efectivo ya que posibilita tomar funciones internas, como los códigos del monolito original, y envolverlos utilizando APIs, en este caso FastAPI. De esta manera se logra aislar y encapsular los servicios para ser ejecutados de manera independiente sin la necesidad de reescribir el código fuente, facilitando una transición segura a microservicios.

3 ESTADO DEL ARTE

3.1 Soluciones Existentes y sus Limitaciones

En esta área de consultas técnicas especializadas, existen diversas tecnologías que se aplican. Sin embargo, la mayoría presentan limitaciones importantes respecto a la necesidad específica del proyecto, entre estas tenemos:

- **Sistemas de documentación industrial tradicionales:** encontramos plataformas como Siemens Teamcenter[11] o ABB Ability, que ofrecen gestión de documentos robusta, pero no integran capacidades de inteligencia artificial para entablar una

conversación con el servicio. Se enfocan en la organización jerárquica de documentos, sin comprensión del texto.

- **Chatbots genéricos con IA:** encontramos soluciones como ChatGPT Enterprise[12] o Claude, con interfaces avanzadas para las conversaciones, pero están diseñadas como asistentes generales. No incorporan procesamiento de manuales ni trazabilidad sobre las respuestas, limitando su aplicación en entornos industriales reales, donde la precisión es vital.
- **Framework RAG de código abierto:** encontramos herramientas como Haystack de Deepset[13] y Langchain que permiten construir sistemas de conversación basados en recuperación de información. Sin embargo, estas soluciones requieren un nivel de manejo importante para su optimización e implementación en entornos productivos.
- **Soluciones OCR especializadas:** encontramos servicios en la nube como Google Document AI[14] y AWS Textract que ofrecen capacidades avanzadas en el reconocimiento de caracteres en documentos, pero funcionan como componentes aislados que no integran flujos completos de generación de respuestas contextuales como lo que se busca.

3.2 Análisis Comparativo y Brecha Identificada

El proyecto ATLAS se distingue de las soluciones existentes al realizar la integración completa de las tecnologías involucradas en un único flujo: procesar manuales mediante OCR, consultar la base de conocimientos especializada con RAG y mantener la trazabilidad.

La principal ventaja que brinda este proyecto es la especialización industrial combinada con accesibilidad técnica, pues esta solución es adaptable y escalable.

Sin embargo, la brecha crítica que se encuentra no es solo funcional, sino en este caso, arquitectónica. Las soluciones existentes no muestran cómo desplegar los modelos pertinentes en servidores locales modernos sin comprometer la disponibilidad del servicio, es aquí donde se ubica el corazón de este trabajo en particular, ya que no solo se integran las tecnologías en un único flujo de trabajo, sino que se refactoriza la arquitectura en microservicios híbridos para garantizar la disponibilidad del sistema completo, haciendo que el proyecto sea viable operativamente.

4 DISEÑO DE LA ARQUITECTURA Y SOLUCIÓN

4.1 Evolución del Diseño Arquitectónico

La arquitectura del sistema ATLAS se alcanza luego de varias iteraciones, donde se fueron encontrando distintas limitaciones técnicas al momento de hacer pruebas. A continuación, se detalla la transición desde la base monolítica hasta la solución final planteada.

4.1.1 Fase 1: Prototipo Monolítico

En la etapa temprana del proyecto el sistema se desarrolló en una arquitectura monolítica, donde todas las funcionalidades (servidor web, lógica de negocio, gestión de usuarios, pipeline

OCR y generación de respuestas RAG) compartían el mismo contenedor del backend con django, el cual se conectaba con docker compose a un contenedor del frontend con Unicorn. Si bien en primera instancia esta arquitectura simplificó el desarrollo temprano del proyecto, las pruebas mostraron un problema crítico: el bloqueo de recursos. El procesamiento de documentos con OCR y vectorización es una tarea que consume gran cantidad de CPU, mientras que el chat con RAG es una tarea ligera, pero al coexistir en el mismo proceso, la carga de un manual pesado (superior a 200 páginas) saturaba el servidor, por lo que el chat dejaba de responder, devolviendo errores de tiempos de espera, "Error 504 Gateway Timeout". Este problema de disponibilidad valida el plan de transición a una arquitectura de microservicios en teoría.

4.1.2 Fase 2: Arquitectura de Microservicios Híbridos

Para poder solucionar la problemática de disponibilidad se diseña una arquitectura de microservicios híbridos, la idea base es desacoplar los servicios según el consumo de recursos. La solución consiste en refactorizar la arquitectura monolítica, separando los módulos intensivos en servicios semi independientes:

- **Servicio de ingesta:** Servicio dedicado exclusivamente al procesamiento de los manuales, es el servicio más pesado del sistema al utilizar reconocimiento de caracteres (OCR) además de vectorizar. Siendo un microservicio semi independiente, su caída o saturación no afecta al resto del sistema.
- **Servicio de QA:** Servicio dedicado a las consultas de los usuarios, utiliza generación por recuperación aumentada (RAG), es más liviano que el servicio de ingesta y se espera que siempre esté disponible. Mantiene los modelos relacionales del backend utilizados en el monolito.
- **Orquestador:** El monolito original también mejora, ya que se han desacoplado servicios, con lo cual se aligera, actuando como gestor de tráfico y seguridad.

Se denominan microservicios "Híbridos" debido a que, a pesar de que la ejecución es aislada según los contenedores distintos, se mantiene el acoplamiento a nivel datos para evitar complejidad al momento de gestionar las bases de datos distribuidas y mantener la consistencia de los datos.

4.2 Arquitectura de Despliegue Detallada

Para lograr implementar la solución se utiliza Docker Compose[10] como orquestador, definiendo una topología de red que priorice el aislamiento y la seguridad. La arquitectura se compone de cuatro contenedores fundamentales:

1. Contenedor de Frontend (Capa de presentación):

- **Tecnología:** Nginx single page application en React.
- **Puerto:** Se expone el puerto 80 del host.
- **Función:** Maneja la interacción entre el usuario y la aplicación. Envía las peticiones hacia el Gateway.

2. Contenedor de Backend Gateway (Orquestador):

- **Tecnología:** Django (python).
- **Puerto:** Se expone el puerto 8000 del host.
- **Función:** Es el único punto en entrada a los servicios. Valida sesiones, gestiona usuarios y enruta internamente las peticiones a los servicios.

3. Contenedor servicio de ingesta (Procesamiento):

- **Tecnología:** FastAPI[17], Tesseract OCR y PyMuPDF
- **Puerto:** Escucha el puerto 8001.
- **Seguridad:** Se configura en modo Internal-Only. Este puerto no se mapea al host, para hacerlo inaccesible desde el exterior. Solo puede acceder desde el Gateway.

4. Contenedor servicio de consultas (Chat):

- **Tecnología:** FastAPI, SentenceTransformers y FAISS.
- **Puerto:** Escucha el puerto 8002
- **Función:** Se configura en modo Internal-Only. Este puerto no se mapea al host, para hacerlo inaccesible desde el exterior. Solo puede acceder desde el Gateway.

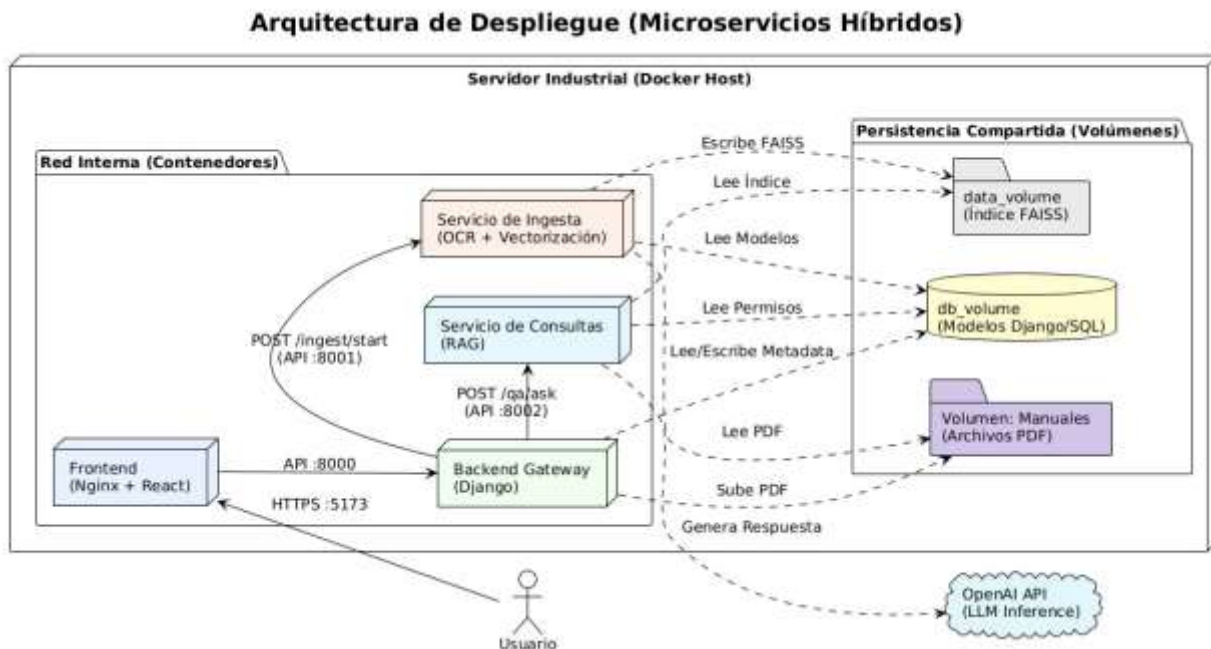


Figura 1: Diagrama de despliegue de la arquitectura de microservicios híbridos.

Fuente: Elaboración propia.

4.3 Diseño de la Persistencia Compartida

Es parte base de este trabajo para ser viable el despliegue el uso de volúmenes Docker para compartir el estado entre servicios sin necesidad de bases de datos distribuidas. Los volúmenes se diseñan de la siguiente forma:

- **Volumen /data** (Índice Vectorial): este volumen almacena los índices FAISS.
 - El servicio de ingesta tiene permisos de Escritura para crear o actualizar los índices luego de procesar un manual.
 - El servicio de consultas solo tiene permisos de lectura para realizar búsquedas, garantizando que la consulta utilice los datos del índice vectorial generado por el servicio de ingesta.
- **Volumen /db** (Metadatos Relacionales): este volumen aloja la base de datos SQLite.
 - Para evitar riesgos de corrupción típicos en SQLite, se habilita el modo WAL[19] (Write-Ahead Logging).
 - Este modo permite que múltiples procesos lean la base de datos simultáneamente mientras uno escribe, garantizando consistencia.

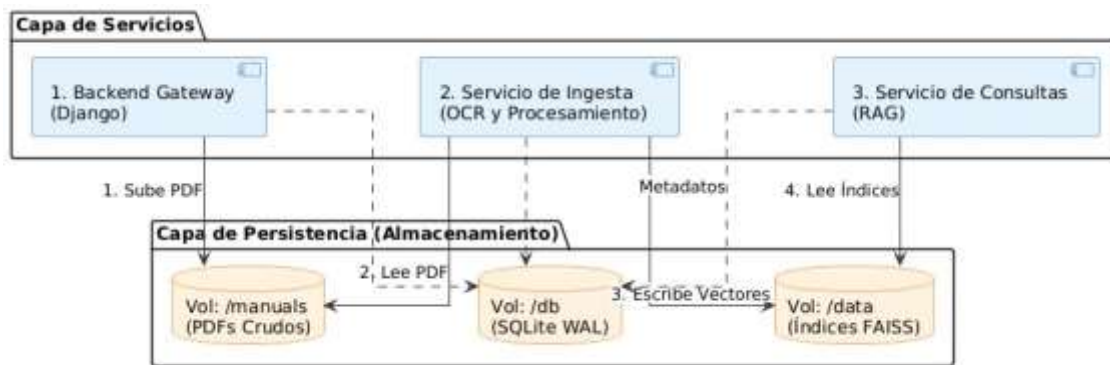


Figura 2: Diagrama de persistencia compartida y acceso de servicios.

Fuente: Elaboración propia.

4.4 Stack Tecnológico

La selección de las tecnologías a utilizar prioriza el rendimiento en entornos locales y el uso de software de código abierto.

- **Backend:** Desarrollado en Python 3.12 utilizando Django 5.2[16] para la gestión de seguridad y Gunicorn 22.0 como servidor de aplicaciones WSGI.
- **IA y Procesamiento:** El módulo RAG se gestiona con LangChain 0.3[15], utilizando FAISS para la búsqueda de vectores de manera eficiente y OpenAI v2.1 como motor de generación. Para el procesamiento de imágenes se utiliza Tesseract 5 y OpenCV Headless, optimizado para servidores sin interfaz gráfica, ahorrando el consumo de recursos por la interfaz. La incorporación de FastAPI para estos servicios se justifica por su manejo asíncrono nativo, característica para no bloquear los hilos operativos del servidor.
- **Frontend:** Construido en Node.js 20 y servido con Nginx Stable sobre Alpine Linux, para mantener un bajo consumo de recursos por parte del frontend. Node.js se utilizó por su entorno optimizado para empaquetar aplicaciones ligeras.
- **Persistencia:** Se utiliza SQLite habilitando el modo WAL en lugar de migrar a PostgreSQL para mantener la coherencia con el principio de simplicidad de despliegue local. Agregar PostgreSQL requeriría un contenedor extra.

Se le priorizó el desacoplamiento del sistema y del proveedor del modelo LLM. La arquitectura modular del servicio de consultas permite que sustituir el motor de generación se pueda hacer



sencillamente, solo cambiando las variables de entorno y ajustar los llamados específicos desde el módulo de consultas.

5 IMPLEMENTACIÓN Y DESPLIEGUE

Esta fase del proyecto es la materialización del diseño arquitectónico definido. El desafío principal no se centra en la creación de más código, sino en migrar la lógica existente del monolito hacia contenedores aislados sin romper la integridad de los datos ni la lógica de negocio.

5.1 Estrategia de Implementación: Refactorización por Encapsulamiento

Para evitar riesgos de regresión escribiendo código nuevo desde cero, se opta por el uso de "Wrappers"(Envoltorios). La ventaja principal de esto es que permite exponer funciones internas como servicios independientes.

- **Paso 1: Gestión de dependencias:** Se identificó que la tarea crítica correspondía al procesamiento de documentos, por lo que, considerando que cada servicio tendrá su propio contenedor, las librerías y dependencias pesadas que utiliza el servicios de ingesta no necesitan ser cargadas en los otros contenedores. Se generan archivos de requerimientos para cada contenedor, haciendo el backend gateway más liviano al quitar las dependencias pesadas del servicio de ingesta o del servicio de consultas.
- **Paso 2: Creación de Wrappers con FastAPI:** Se implementan dos nuevas aplicaciones usando el framework FastAPI para envolver la lógica. Para enfrentar el desafío de sincronizar los datos a los que acceden los servicios se inicializa el módulo de configuración de Django, permitiendo que el código acceda al ORM de Django y puedan realizar consultas a la base de datos compartida, reutilizando el código de los modelos que ya fueron validados en la versión monolítica.
- **Paso 3: Comunicación entre servicios:** Se modificó el Backend Gateway eliminando las llamadas a funciones locales. Fueron remplazadas por llamadas HTTP asíncrona usando la librería httpx, así no es bloqueante en memoria la solicitud del servicio de ingesta.

5.2 Gestión de Configuración y Secretos

Se implementa un sistema de inyección de variables de entorno para mantener la seguridad, evitando que las variables queden en el código expuestas.

- Orquestación de variables: utilizando docker compose se centralizó la configuración mediante archivos de entorno, lo que permite gestionar:
 - Credenciales: API Keys de servicios externos (como OpenAI).
 - Rutas de infraestructura: Rutas de la ubicación de los volúmenes persistentes compartidos por los contenedores.



- Topología de red: Las URLs de los servicios internos se definen como variables, lo que permite cambiar la arquitectura de red de manera más sencilla.

5.3 Contenerización Especializada

A continuación, se describen las dependencias clave que incluye cada contenedor.

1. **Contenedor de Ingesta:** Este contenedor aísla la carga computacional más alta y agrupa las dependencias necesarias para procesar los documentos y convertirlos en una base vectorial.
 - **Imagen base:** python:3.12-slim.
 - **Puerto:** Escucha el puerto interno 8001, no expuesto al host.
 - **Dependencias del sistema:** se instalan paquetes apt-get críticos:
 - tesseract-ocr: El motor OCR.
 - poppler-utils: Necesario para que la librería pdf2image convierta los archivos PDF a imágenes.
 - opencv-python-headless: Se utiliza versión headless para evitar la instalación de librerías gráficas pesadas, reduciendo el peso de la imagen.
 - **Resultados:** Una imagen robusta capaz de ejecutar las tareas de procesamiento intensivo sin comprometer el resto del sistema.
2. **Contenedor de Consultas:** Este contenedor está diseñado para la inferencia y la interacción con los modelos LLM, priorizando la alta disponibilidad.
 - **Imagen base:** python:3.12:slim.
 - **Puerto:** Escucha el puerto interno 8002, no expuesto al host.
 - **Optimización:** Se instalan únicamente librerías de inteligencia artificial y manejo vectorial, excluyendo todas las librerías de herramientas OCR y manipulación de archivos PDFs.
 - **Resultados:** Una imagen ligera que arranca en segundos, optimiza el uso de memoria RAM para mantener los índices vectoriales y ofrecer alta disponibilidad del chat.
3. **Contenedor Backend Gateway:** Este contenedor es el punto de entrada y gestor de tráfico, liberado de tareas pesadas.
 - **Puerto:** Expuesto al host en el puerto 8000.
 - **Limpieza:** A partir del dockerfile del monolito, se eliminan instalaciones de las dependencias del módulo OCR y RAG. Se reduce en entorno estrictamente a lo necesario para servir peticiones HTTP, Django, Django REST Framework y Unicorn.
 - **Resultado:** El consumo de memoria del contenedor de Django se redujo notablemente, ya que ahora delega las funciones clave del sistema y se encarga solo del enrutamiento, la autenticación y la gestión de sesiones.
4. **Contenedor Frontend:** Este contenedor es el encargado de servir la interfaz de usuario eficientemente y desacoplado de la lógica del backend.
 - **Puerto:** Expuesto internamente al puerto 80, mapeado al puerto 5173 del host.
 - **Construcción:** Se configura un proceso de construcción en dos etapas para minimizar el tamaño.
 - Etapa de compilación: se utiliza una imagen temporal con node:20-alpine para instalar las dependencias y compilar el código React, generando archivos estáticos optimizados.
 - Etapa de ejecución: utiliza nginx:stable-alpine y utiliza solo los archivos generados en la etapa anterior.
 - **Resultados:** Una imagen pequeña y segura, sin contener código fuente ni dependencias de desarrollo, así obteniendo una carga rápida al acceso a la aplicación.

6 PLAN DE VALIDACIÓN Y RESULTADOS PROYECTADOS

Dado que el objetivo central de este trabajo es resolver el problema crítico sobre la disponibilidad operativa, la validación no se enfoca en medir la calidad de las respuestas del asistente, sino de cuantificar la estabilidad y robustez de la infraestructura en condiciones de carga intensa. Se diseñó un experimento comparativo teórico entre la arquitectura monolítica y la arquitectura de microservicios híbridos.

6.1 Metodología de Validación

Para poder cuantificar las mejoras, se definen escenarios de prueba que someten al sistema a distintos niveles de carga.

- **Herramientas de medición:**
 - **Generación de carga:** uso de scripts automatizados para simular peticiones.
 - **Monitoreo de recursos:** Registro del consumo de recursos, CPU y RAM, en tiempo real por contenedor mediante docker stats.
 - **Análisis de Disponibilidad:** Detección de códigos de error en los logs del servidor Gateway.
- **Definición de escenarios:**
 - **Escenario A** (Base - Carga ligera)
 - **Acción:** Un usuario sube un manual pequeño, no más de 5 páginas y con texto nativo, mientras otro usuario realiza consultas en el chat.
 - **Objetivo:** Verificar el funcionamiento correcto en condiciones de carga.
 - **Escenario B** (Caso intensivo - Carga pesada)
 - **Acción:** Un usuario inicia el procesamiento de un manual pesado, más de 300 páginas, que requiere un trabajo intensivo del OCR, mientras otro usuario realiza consultas concurrentes.
 - **Objetivo:** Determinar si el servicio de chat sigue respondiendo mientras el servidor procesa la ingesta de documentos.

6.2 Resultados Proyectos vs Observados

El análisis permite proyectar los siguientes comportamientos:

1. Disponibilidad del servicio de chat.

- **Arquitectura monolito:** Se proyecta una disponibilidad aproximada del 0% en el escenario B, ya que el proceso de OCR satura el servidor, las peticiones de chat entrarán en cola hasta expirar, resultando en errores 504 Gateway Timeout.
- **Arquitectura microservicios híbridos:** Se proyecta una disponibilidad aproximada del 100%, al estar el proceso de ingesta aislado en su contenedor, los otros del servicio de consultas y del backend gateway mantienen sus recursos libres para

responder las consultas del chat, independiente de la carga en segundo plano de archivos procesados.

2. Aislamiento de fallos.

- **Hipótesis de fallo:** Se simula un error fatal en la librería de procesamiento de imágenes que cause el cierre del proceso.
- **Impacto en monolito:** El cierre del proceso detiene todo el servidor web, desconectando usuarios y perdiendo sesiones.
- **Impacto en microservicios:** El cierre del proceso solo afecta al contenedor de ingesta, el cual se detiene o reinicia automáticamente, no afectando al servicio de consultas, no se propaga el fallo a otros módulos.

3. Consumo de recursos.

- **Análisis:** Se anticipa un incremento en el consumo de memoria RAM en la arquitectura de microservicios. Se estima que el incremento de consumo de memoria total del sistema sea de aproximadamente 40% más.
- **Causa:** La arquitectura híbrida implica la ejecución simultánea de múltiples instancias del intérprete de Python, una por cada contenedor activo, y la duplicación de librerías base del sistema por cada imagen docker, consumiendo más capacidad operativa que no existe en el proceso del sistema monolito.
- **Justificación:** Este costo adicional de memoria es aceptable, pues se intercambia una menor eficiencia de memoria RAM por estabilidad y aislamiento de los procesos. En un entorno industrial es importante garantizar que el servicio de consultas esté siempre disponible.

Para contrastar estos resultados con los observados se utiliza el comando `docker stats`, ajustado para que muestre los estados cada 2 segundos. A partir de estos se pueden observar los máximos consumos de recursos en cada prueba, además de observar desde otra terminal por los errores, ya sea por los bloqueos en el monolito o errores de timeout en el mismo. Se muestra en la siguiente tabla la comparación de los resultados en el monolito, la proyección de la nueva arquitectura y los resultados obtenidos en la nueva arquitectura.

Tabla 1: Comparativa resultados y proyección.

Escenario	Métrica Crítica	Monolito	Proyecciones	Microservicios
B – Manual Extenso	Disponibilidad Chat	~0% (Timeouts)	~100% (Estable)	~90% (Pequeños errores)
B – Manual Extenso	Consumo de procesamiento (CPU)	~97.01% CPU valor máximo en pruebas (Contenedor backend)	Estable (Límites Docker)	~49.72% CPU valor máximo en pruebas (Contenedor ingesta)



B – Fallo OCR	Impacto Sistema	Caída Total	Fallo Aislado	Chat disponible, errores por ausencia de contenedor ingesta
------------------	--------------------	-------------	---------------	---

Fuente: Elaboración propia.

7 CONCLUSIONES

7.1 Conclusiones Generales

En este trabajo se abordó una problemática de la digitalización industrial: la inviabilidad operativa al desplegar modelos de inteligencia artificial en arquitecturas monolíticas. Se demuestra que, aunque las tecnologías de RAG y OCR son capaces de automatizar el acceso al conocimiento técnico, su implementación en una arquitectura monolítica es propensa a bloqueos que comprometen la disponibilidad del servicio, un requisito no funcional clave en las operaciones de terreno. La solución basada en esta arquitectura de microservicios híbridos resuelve el conflicto de disponibilidad al separar y aislar las cargas de trabajo, los principales logros de esta implementación fueron:

- El desacoplamiento efectivo de los servicios, separando en uno de trabajo intensivo, de ingesta y procesamiento de documentos, y otro de consultas, dedicado a tomar la base de conocimientos para generar respuestas lo más precisas y confiables posible. Todo esto gracias a la refactorización por encapsulamiento en contenedores Docker.
- El aislamiento de recursos y fallos garantiza que algún tipo de saturación o errores en el módulo de ingesta no propague su impacto a los demás módulos, manteniendo la disponibilidad del servicio de consultas.
- La validación de un modelo de persistencia compartida con volúmenes Docker y SQLite en modo WAL, permite la comunicación de los datos entre servicios sin agregar la complejidad de bases de datos distribuidas.

La principal contribución técnica de este trabajo va más allá del desarrollo de un asistente virtual especializado, se centra en demostrar la viabilidad de desplegar un sistema de IA complejo en una infraestructura local. Se valida que mediante una arquitectura de microservicios es posible ejecutar tareas complejas como OCR, procesamiento y RAG, directamente en el servidor local, sin depender de la nube. Esto garantiza la privacidad de los datos y la continuidad operativa, confirmando que el proyecto ATLAS es una solución técnicamente robusta para entornos productivos reales. Finalmente se reconoce el intercambio de recursos en el diseño: la arquitectura distribuida trae consigo una mayor demanda en cuanto a memoria RAM, pero es más robusto y estable respecto al monolito. Este costo se considera aceptable en el contexto industrial, priorizando la disponibilidad del sistema.

Los resultados que se observaron fueron satisfactorios, si bien se encuentran otros pequeños problemas que afectan la disponibilidad, se alcanzan valores de disponibilidad similares a lo esperado, los consumos de procesamiento se mantuvieron en niveles notablemente menores en la arquitectura híbrida comparada a la arquitectura monolítica, validando la solución propuesta de re-estructurar en una arquitectura de microservicios híbridos que asegure la disponibilidad y garantice el funcionamiento del chat.

7.2 Trabajo Futuro

Para poder alcanzar un producto final más avanzado, escalable y robusto, se identifican los siguientes posibles avances:

- **Desacoplamiento total de la persistencia:** Para desacoplar totalmente necesitamos migrar la base de datos SQLite a una como PostgreSQL y eliminar la dependencia del ORM de Django en los microservicios, esto implica reemplazar todas las llamadas a django por APIs REST bien definidas, consiguiendo la independencia y escalabilidad de cada servicio.
- **Orquestación y Gestión de producción:** Avanzar de Docker Compose a un orquestador como Kubernetes, habilitando la capacidad de auto-escalado, auto-reparación, gestión de secretos y un despliegue más robusto.

Referencias

- [1] Jurafsky, D. & Martin, J. H. *Speech and Language Processing*, 3rd Ed. draft, Stanford University, 2024.
- [2] Brown, T. et al. *Language Models are Few-Shot Learners*, OpenAI, 2020.
- [3] Vaswani, A. et al. *Attention is All You Need*, Google Research, 2017.
- [4] Smith, R. *An Overview of the Tesseract OCR Engine*, Google Research, 2007.
- [5] Google Developers. *Tesseract OCR Documentation*, 2024.
- [6] Johnson, J., Douze, M., & Jégou, H. *Billion-scale Similarity Search with GPUs*, Facebook AI Research, 2017.
- [7] Lewis, P. et al. *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*, Facebook AI, 2020.
- [8] Merkel, D. *Docker: Lightweight Linux Containers for Consistent Development and Deployment*, Linux Journal, 2014.
- [9] Newman, S. *Building Microservices*, O'Reilly Media, 2015.
- [10] Docker Inc. *Docker Compose Documentation*, 2024.



- [11] Siemens. *Teamcenter: Industrial Knowledge Management Platform*, 2023.
- [12] OpenAI. *ChatGPT Enterprise: Security and Privacy for Enterprise*, 2023.
- [13] Deepset AI. *Haystack: Open Source Framework for RAG Systems*, 2024.
- [14] Google Cloud. *Document AI Overview*, 2024.
- [15] LangChain AI. *LangChain Python Documentation: Building applications with LLMs*, 2024.
- [16] Django Software Foundation. *Django Documentation Release 5.2*, 2024.
- [17] Ramírez, S. *FastAPI: Modern, fast (high-performance), web framework for building APIs with Python*, 2024.
- [18] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [19] SQLite Consortium. *Write-Ahead Logging. SQLite Documentation*, 2024.