UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

TÍTULO: CRANE: Simplificando el despliegue de aplicaciones contenerizadas en entornos locales

AUTORES: Silva Pavón José Miguel, Bellino Franco

DIRECTORAS: Bazán Patricia, Lliteras Alejandra B.

CARRERA: Licenciatura en Sistemas

Resumen

CRANE es una herramienta diseñada para el despliegue local de aplicaciones en contenedores, enfocada en simplificar las pruebas de entornos distribuidos de forma local. CRANE ofrece una solución liviana y de propósito general con capacidades de ruteo, escalado y monitoreo automático. Orientada a estudiantes, docentes y desarrolladores que necesiten crear y desplegar servicios en contenedores simulando las características básicas que ofrece un entorno cloud de plataforma como servicio (PaaS).

Palabras Clave

Docker, escalado, monitorización, API REST, DevOps, PaaS, Cloud

Conclusiones

Podemos concluir que se ha logrado el objetivo de esta tesina, que consistió en crear una herramienta que permita simular las características básicas de un entorno distribuido de forma local. Este proyecto podría facilitar a estudiantes, docentes y desarrolladores profundizar su conocimiento sobre servicios en contenedores de manera más interactiva, mientras aprenden los conceptos básicos de un entorno cloud. Las funciones de la API REST de CRANE abarcan la creación, el despliegue y la administración de aplicaciones en contenedores con enrutamiento,

Trabajos Realizados

- *Análisis e implementación del modelo de CRANE.*
- *Desarrollo de API REST para la creación de servicios en contenedores.*
- *Interacción con el Docker Daemon usando Python.*
- *Integración de base de datos para la gestión de usuarios, roles y aplicaciones.*
- *Implementación de un sistema de seguridad JWT para la autenticación y un modelo RBAC para la autorización.*
- *Integración con sistemas de monitoreo (Prometheus) y alertas (AlertManager).*
- *Integración con un sistema de reglas (OPA).*
- *Servicios de escalado automático basado en reglas y enrutamiento automático (Traefik).*
- *Documentación utilizando Swagger OpenAPI.*

Trabajos Futuros

- *Implementación de un Frontend para mejorar la interacción.*
- *Análisis visual de rendimiento y consumo de los contenedores usando gráficos.*
- *Extender las métricas y alertas permitidas.*
- *Alta, Baja, Consulta y Modificación de Reglas*
- *Mejorar el diseño y creación de los archivos docker-compose.*

Fecha de la presentación: MAYO 2024

Universidad Nacional de La Plata
Facultad de Informática

Tesina de Grado
Licenciatura en Sistemas

CRANE: Simplificando el Despliegue de Aplicaciones Contenerizadas en Entornos Locales

Autores: José Miguel Silva Pavón y Franco Bellino
Directoras: Patricia Bazán y Alejandra B. Lliteras

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

CAPÍTULO 1 - OBJETIVOS

1.1 | Motivación

Los servicios PaaS (Platform as a Service) [1] revolucionaron el paradigma del desarrollo de software en esta era digital al proveer entornos que vienen equipados con herramientas para desarrollar y desplegar código directamente en la nube. Esta flexibilidad y comodidad permite que los programadores se centren en la lógica de aplicación y se abstraigan de las complejidades que trae el manejo de la infraestructura. Pero cuando las necesidades no ameritan el uso de un PaaS y es indispensable trabajar con la infraestructura de forma temprana, empiezan a surgir problemas como:

- Migración del entorno a diferentes plataformas, cada una de estas con diferentes características. Esto implica que haya archivos de configuración diferentes para cada entorno e instalación de dependencias propias donde se vaya a utilizar.
- Dificultad para ejecutar pruebas de rendimiento efectivas en etapas tempranas y construir aplicaciones verdaderamente escalables.
- La migración de aplicaciones entre diferentes hosts es una tarea complicada por las dependencias específicas del sistema, como las versiones de librerías y los distintos sistemas operativos.

A raíz de estos desafíos los contenedores de Docker [2] surgen como una solución prometedora, permitiendo el despliegue de aplicaciones en cualquier host con Docker instalado, independientemente de sus características específicas. Esto lleva a la necesidad de orquestar, medir y escalar estos contenedores de un modo más eficaz, un problema que las soluciones actuales como Kubernetes [3][11] resuelve, pero con un alto costo en recursos computacionales. La configuración de estos contenedores requiere la creación de archivos de configuración estáticos que suelen depender de rutas locales, complicando más el proceso.

Debido a estas necesidades, surgen nuevos roles como el de DevOps (Development and Operations) [4] que une roles que anteriormente estaban separados (desarrollo, operaciones de

TI, ingeniería de la calidad y seguridad) en un solo rol y así tener una visión completa de todo el sistema a desarrollar.

Es por esto, que en este trabajo, se propuso el desarrollo de CRANE [5], el cual aporta una solución de bajo costo, para el despliegue y orquestación de contenedores de forma local.

1.2 | Objetivos Generales

Este trabajo se enfoca en la creación de un sistema que ha sido diseñado para facilitar el despliegue y la gestión autónoma de contenedores en un entorno local. Mediante esta plataforma se permite la implementación de forma rápida y eficiente de servicios en contenedores que son administrados automáticamente por CRANE sin necesidad de intervención manual del desarrollador, hasta en situaciones de bajo rendimiento o errores del sistema host.

Los contenedores están conectados entre sí mediante una red que facilita obtener y analizar métricas en tiempo real. Cualquier descenso en el rendimiento de los contenedores es detectado por este análisis de los indicadores, y se activa un conjunto de políticas preestablecidas para responder de un modo adecuado a estos eventos.

La gestión del sistema es realizada mediante una API REST, que se conecta con el cliente de Docker instalado en el sistema operativo por medio de una librería de Python [6]. Estas herramientas en conjunto, y su organización en la arquitectura de CRANE, permiten realizar las operaciones descritas de forma eficaz y automatizada.

1.3 | Desarrollos Propuestos

Para lograr el objetivo general de este trabajo, se analizó en detalle la arquitectura de CRANE y se propuso implementar los siguientes puntos:

Creación y Gestión de Contenedores:

- Desarrollar un servicio REST para la creación automática de contenedores que facilite la escalabilidad y la gestión rentable de recursos.
- Implementar la generación de archivos de configuración dinámicos y estáticos para el despliegue automático de contenedores que mejoren la adaptabilidad del sistema ante cambios de configuración.
- Establecer buenas configuraciones de seguridad para la gestión de contenedores que aseguren la protección de datos y operaciones.

Utilización de Datos y Bibliotecas:

- Agregar una base de datos para manejar la gestión de usuarios, roles y aplicaciones, centralizando la información y mejorando la seguridad del sistema.
- Utilizar una librería de Python para la interacción programática con Docker, optimizando los procesos de gestión de contenedores.

Integración y Monitoreo:

- Implementar un proxy reverso con Traefik para gestionar de la mejor forma el tráfico de la red y la seguridad en la comunicación.
- Agregar sistemas de gestión de métricas y alertas para el monitoreo en tiempo real y la respuesta rápida ante incidentes.
- Incorporar un sistema de gestión de políticas para automatizar la aplicación de reglas operativas y de seguridad.

Seguridad y Autenticación:

- Implementar un modelo RBAC (Role-Based Access Control) para la autenticación y autorización, reforzando el control de acceso al sistema.

Documentación:

- Documentar utilizando la especificación Swagger OpenAPI, para facilitar la comprensión y el uso del sistema por parte de desarrolladores y nuevos usuarios.

1.4 | Aporte de la Tesina

Al finalizar este trabajo se obtiene una aplicación que permite a los usuarios crear servicios en contenedores de forma fácil y rápida, con monitoreo de métricas, políticas preestablecidas para la seguridad y disponibilidad, y manteniendo las mismas configuraciones en cualquier máquina donde se ejecute.

El usuario se comunica con Crane mediante una API REST y puede crear contenedores y ajustar las reglas de escalado dependiendo de su propio análisis y/o reglas previamente definidas. Se tiene el control de los puertos expuestos de la aplicación, tráfico de los contenedores y se pueden clonar las configuraciones de aplicaciones para generar nuevas instancias de forma más rápida.

Al ser portable y de fácil acceso, se puede desplegar en cualquier sistema simplemente ejecutando un comando. A su vez se tienen a disposición los archivos de Docker Compose generados para poder llevarlos a cualquier otro equipo sin necesidad de tener que sacar a Crane del entorno local de desarrollo.

CAPÍTULO 2 - INTRODUCCIÓN

2.1 | ¿Qué significa DevOps?

DevOps es una metodología que combina el mundo del desarrollo de software (Dev) y operaciones de TI (Ops) con el fin de poder acelerar el ciclo de vida de estas y asegurar la entrega continua de software de calidad. Su objetivo principal se enfoca en automatizar procesos y aumentar la colaboración entre los equipos, promoviendo una cultura de responsabilidad compartida a lo largo de toda la vida útil de la aplicación.

Esta metodología comenzó a utilizarse alrededor del año 2007, cuando las comunidades de desarrollo de software y operaciones de TI se preocuparon por el modelo tradicional de escritura de software donde los desarrolladores escribían el código y trabajaban de forma independiente del equipo de operaciones, quienes se encargaban de implementar y brindar soporte sobre el mismo. Es por esto que el término "DevOps" es una combinación de las palabras desarrollo (Development) y operaciones (operations) porque refleja el proceso de integrar estas disciplinas en un proceso continuo y unificado.

Un equipo de este modelo está constituido por desarrolladores y operaciones de TI trabajando colaborativamente a lo largo del ciclo de vida del producto, con el objetivo de aumentar la velocidad y calidad de la implementación del software. Es una nueva forma de trabajar, un cambio cultural, que tiene implicaciones significativas para los equipos y las organizaciones para las que trabajan.

Bajo este enfoque los equipos de desarrollo y operaciones ya no están separados. En muchos casos, estos equipos se integran en una entidad única en la que los ingenieros colaboran en todas las etapas del ciclo de vida de la aplicación, abarcando desde el desarrollo y la evaluación hasta la implementación y operaciones. Esto generó que los profesionales posean un conjunto diverso de habilidades multidisciplinarias, promoviendo una colaboración más estrecha y eficiente.

Los equipos de desarrollo y operaciones utilizan herramientas para automatizar y acelerar los distintos procesos permitiendo aumentar la confiabilidad. Estas herramientas ayudan a los

equipos a abordar fundamentos importantes, como la integración continua, la entrega continua, la automatización y la colaboración.

Debido a la "naturaleza continua" de esta metodología, los practicantes utilizan el bucle infinito para mostrar cómo se relacionan entre sí las fases del ciclo de vida de DevOps. A pesar de parecer que fluyen de un modo secuencial, el bucle simboliza la necesidad de colaboración constante y mejora iterativa a lo largo de todo el ciclo de vida.

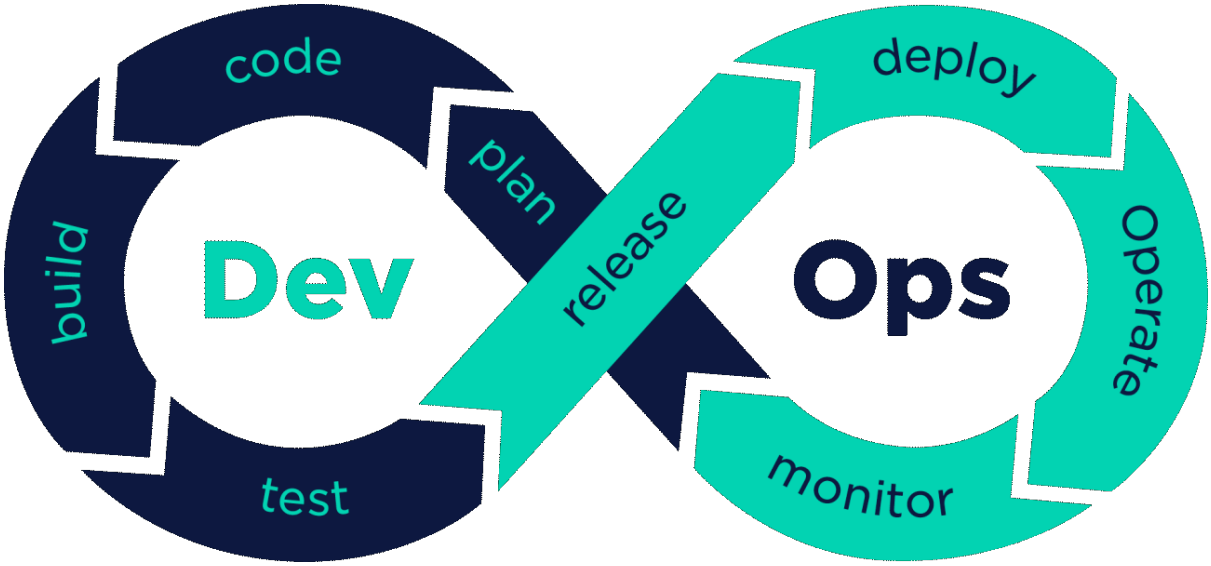


Figura 1. Ciclo de vida de una aplicación utilizando metodologías Devops

2.2 | Dificultades de pruebas en entornos locales

El desarrollo de software en entornos locales, si bien presenta ventajas significativas, también trae una serie de limitaciones que pueden complicar el proceso de escritura e implementación del software. La configuración inicial del entorno de programación requiere que los desarrolladores instalen y configuren todas las herramientas necesarias, un proceso que consume bastante tiempo y esfuerzo. Es muy común además que la variabilidad entre los entornos de desarrollo local y de producción pueda provocar que un software que funciona adecuadamente en el entorno local no se comporte igual en el entorno final de producción.

Este problema se ve exacerbado por la diversidad de sistemas operativos utilizados en los equipos técnicos, como Linux, Mac o Windows [7], cuyas diferencias pueden afectar bastante la ejecución del software, resultando en incompatibilidades que impactan la funcionalidad del programa en diferentes plataformas. La gestión de dependencias introduce otra capa de complejidad porque las variaciones en las versiones de dependencias entre sistemas pueden hacer que el software sea operativo para algunos usuarios, pero no para otros.

Sumado a estos desafíos, la colaboración efectiva en equipos puede verse dificultada en entornos locales debido a que los cambios realizados por un desarrollador pueden no estar disponibles en el momento para el resto del equipo. Aun así, las pruebas en entornos locales son esenciales, en especial por razones económicas.

A pesar de las complicaciones mencionadas, realizar pruebas localmente es mucho más económico si se compara con las pruebas en la nube, donde los costos escalan rápidamente en función del uso de recursos y la duración de las pruebas. Este último punto es muy importante para startups y empresas con presupuesto acotado que necesitan gestionar de forma cuidadosa sus gastos operativos.

Para finalizar, la limitación de recursos en entornos locales, aunque es un desafío, también obliga a los desarrolladores a optimizar el software para que funcione correctamente con recursos limitados. No obstante, si bien el despliegue local puede ser más sencillo, adaptar el software para su óptimo funcionamiento en un entorno productivo necesita modificaciones y trabajo extensivo después de su implementación. Estos factores en conjunto hacen que la escritura de software en entornos locales sea una opción más complicada pero importante para el ciclo de vida del desarrollo de software.

2.3 | El costo de probar aplicaciones en la nube

En el panorama actual de la computación en la nube, hay dos gigantes que se destacan por su capacidad para ofrecer recursos informáticos escalables y flexibles: Amazon Web Services (AWS) y Microsoft Azure. Entre la gran cantidad de servicios que proveen, Amazon Elastic Compute Cloud (Amazon EC2) y Microsoft Azure Virtual Machines son soluciones clave en el espacio de infraestructura como servicio (IaaS) [40].

Razones para Utilizar Servicios en la Nube:

1. **Escalabilidad:** Tanto Amazon EC2 como Azure Virtual Machines permiten escalar la capacidad informática según las necesidades cambiantes de las aplicaciones y los negocios.
2. **Modelo de Pago por Uso:** Ambas plataformas ofrecen un modelo de pago por uso, permitiendo a las empresas pagar solo por los recursos que consumen.

3. **Variedad de Configuraciones:** Tanto AWS como Azure ofrecen una amplia gama de configuraciones de instancias para adaptarse a diversas cargas de trabajo.

Figura 2. Cuadrante Mágico de Gartner para Infraestructura en la nube

Presentación de Costos:

En esta sección se presenta una tabla que detalla los costos del uso de Amazon EC2, Azure Virtual Machines y Google Compute Engine al momento de esta redacción. Esta comparación permite tener una visión más general de cómo los PaaS se alinean en relación a los precios, ayudando a las empresas a tomar decisiones informadas sobre su infraestructura en la nube [8][9][10].

Tabla 1: Precios Amazon AWS (EC2)

Nombre de la Instancia	$ Hora Bajo demanda	vCPU	Memoria	Costo total 30 Dias de Uso
t4g.large	$0.0672	2	8 GiB	$ 48,384
t4g.xlarge	$0.1344	4	16 GiB	$ 96,768
t4g.2xlarge	$0.2688	8	32 GiB	$ 193,536

Tabla 2: Precios Azure Virtual Machines

Nombre de la Instancia	$ Hora Bajo demanda	vCPU	Memoria	Costo total 30 Dias de Uso
B2as	$0.075	2	8 GiB	$ 54
B4as	$0.150	4	16 GiB	$ 108
B8as	$0.301	8	32 GiB	$ 216,72

Tabla 3: Precios Google Compute Engine

Nombre de la Instancia	$ Hora Bajo demanda	vCPU	Memoria	Costo total 30 Dias de Uso
e2-standard-2	$0.0670	2	8 GiB	$ 48,24
e2-standard-4	$0.1340	4	16 GiB	$ 96,48
e2-standard-8	$0.2680	8	32 GiB	$ 192,96

2.4 | Conclusiones

Se considera que elegir entre el desarrollo de software en un entorno local en comparación con la nube implica evaluar detenidamente varios factores críticos como los costos, la escalabilidad, el rendimiento y la seguridad. El desarrollo local puede dar un control más detallado sobre la infraestructura y, en algunos casos, menores costos iniciales. Por otro lado, optar por la nube ofrece beneficios como mayor escalabilidad y flexibilidad, además de brindar acceso a tecnologías avanzadas que pueden no estar disponibles o ser viables localmente.

Al no contar con una solución universal que se ajuste perfectamente a todos los proyectos, hay que considerar las necesidades específicas de cada iniciativa. En este sentido, se puede investigar la posibilidad de integrar las ventajas de ambos enfoques. Implementando tecnologías como la virtualización, la automatización y la orquestación de recursos, es posible crear un entorno local que imita las capacidades de la nube, ofreciendo así una infraestructura que combine lo mejor de ambos mundos.

Por eso se concluye que desarrollar una herramienta personalizada que facilite la escalabilidad en entornos locales y emule las características típicamente asociadas con la nube resulta más beneficioso. Esta solución innovadora debe brindar a los desarrolladores las ventajas de la escalabilidad y la flexibilidad de la nube, mientras se mantiene un control exhaustivo sobre la infraestructura y se gestionan los costos de forma más rentable.

CAPÍTULO 3 - TECNOLOGÍAS UTILIZADAS

3.1 | Docker

3.1.1 | Definición

Docker [12] es una plataforma abierta diseñada para automatizar el despliegue de aplicaciones en contenedores de manera eficiente. Facilita la separación de las aplicaciones de la infraestructura subyacente, permitiendo así una entrega de software más rápida.

Con Docker, es posible gestionar la infraestructura con la misma flexibilidad y agilidad que se manejan las aplicaciones. Esto se logra mediante el uso de contenedores, que son entornos ligeros y portables que encapsulan todo lo requerido para que una aplicación funcione, independientemente del entorno local o de producción.

Al adoptar las metodologías de Docker, que facilitan la integración continua y el despliegue continuo (CI/CD), se reduce ampliamente el tiempo entre la escritura del código y su ejecución en un entorno de producción. Esta capacidad transforma radicalmente los ciclos de desarrollo de software, ofreciendo ventajas respecto a la eficiencia y competitividad.

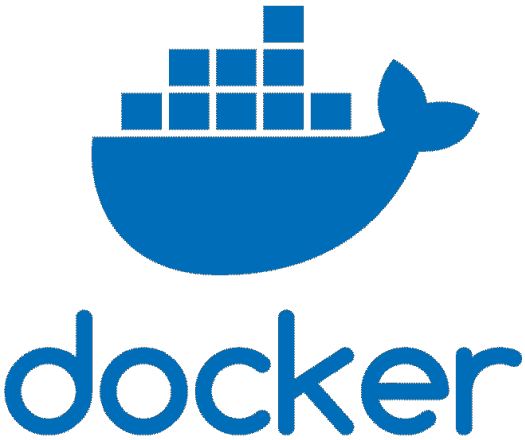


Figura 3. Logo oficial de Docker

3.1.2 | Docker y el despliegue de aplicaciones

En el ámbito del desarrollo y la implementación de software moderno, Docker se ha consolidado como una herramienta muy importante por su capacidad para encapsular aplicaciones y sus dependencias dentro de contenedores virtualizados. Esta tecnología no solo facilita la portabilidad de las aplicaciones por medio de diferentes entornos, sino que también asegura la consistencia operativa desde desarrollo hasta producción.

Se explican a continuación las distintas ventajas y desventajas de utilizar Docker, dando una visión equilibrada para que se entienda el motivo por el cual se decidió utilizar esta herramienta como el gestor de servicios en Crane

Ventajas

1. Facilita la generación de entornos de desarrollo y pruebas, brindando consistencia en la ejecución de aplicaciones en distintas fases del ciclo de vida del desarrollo.

2. Optimiza el flujo de CI/CD al entregar un entorno aislado y reproducible para la construcción y despliegue de aplicaciones.

3. Promueve la adopción de arquitecturas de microservicios al posibilitar la creación de contenedores independientes y escalables, cada uno ejecutando una parte específica de la aplicación.

4. Se emplea en sistemas de orquestación como Kubernetes y Docker Swarm para simplificar la gestión, escalabilidad y orquestación de contenedores en entornos de producción.

5. Es invaluable en despliegues multinube gracias a su capacidad para contenerizar aplicaciones y sus dependencias, permite la portabilidad y consistencia en distintos entornos. Esto posibilita una distribución flexible de contenedores en múltiples proveedores de servicios en la nube, optimizando costos, rendimiento y disponibilidad.

6. Es una solución eficaz para la implementación de aplicaciones en entornos de producción, mejorando la portabilidad y consistencia en diversas infraestructuras.

7. Garantiza el aislamiento de recursos, en otras palabras, las aplicaciones en contenedores no interfieren entre sí, ofreciendo además una capa adicional de seguridad al encapsular aplicaciones.

8. Facilita a los desarrolladores trabajar en entornos heterogéneos porque los contenedores pueden ejecutarse de manera coherente en diferentes sistemas operativos y configuraciones.

Desventajas

1. Las imágenes de Docker pueden volverse bastante grandes, y más si incluyen todas las dependencias necesarias. Esto puede generar un consumo grande de espacio de almacenamiento y ancho de banda para transferir imágenes.

2. Aunque Docker ofrece aislamiento, si no se configura bien, los contenedores pueden presentar riesgos de seguridad. Se deben seguir las mejores prácticas de seguridad y mantener actualizadas las imágenes y dependencias para mitigar posibles vulnerabilidades.

3. El rendimiento de los contenedores puede ser afectado por la virtualización subyacente y el sistema operativo anfitrión y el rendimiento de las aplicaciones dentro del contenedor también puede ser inferior comparado con su ejecución directa en el sistema operativo anfitrión.

4. A medida que las implementaciones aumentan su complejidad, la orquestación de contenedores se vuelve mucho más complicada.

5. Docker es compatible con la mayoría de los sistemas operativos actuales pero algunos aspectos pueden variar entre plataformas y se debe entender cuáles son esos aspectos.

6. La persistencia de datos en contenedores puede ser compleja y más cuando se trata de almacenamiento a largo plazo. Gestionar datos persistentes y bases de datos dentro de contenedores requiere de conocimiento y consideraciones sobre la disponibilidad.

7. Algunas imágenes de Docker tienen software con licencia que se tienen que respetar para evitar problemas legales.

8. Para los que recién empiezan con Docker, puede haber una alta curva de aprendizaje para entender los conceptos de contenedores, imágenes, volúmenes y su adecuada configuración.

Para mitigar el impacto de estas desventajas se deben aplicar buenas prácticas en el manejo de los contenedores, las configuraciones más convenientes dependiendo el tipo de servicio que se desea desplegar y herramientas auxiliares cuando sea necesario

El uso de Docker en este proyecto se justificó por su simplicidad y eficacia en la gestión de aplicaciones. Usando la API REST de Crane, se puede establecer comunicación con el cliente de Docker instalado en el sistema operativo, mejorando el despliegue y control de aplicaciones. La tecnología de contenedorización de Docker encapsula cada aplicación con sus dependencias en contenedores individuales, permite la portabilidad y consistencia entre diferentes entornos operativos. A su vez, se brinda un entorno de desarrollo y pruebas coherente, lo cual es esencial para la creación y el manejo de aplicaciones de forma independiente del sistema operativo. Esta funcionalidad de Docker, vital para la gestión de aplicaciones en el contexto del motor de Crane, fue decisiva para su elección, resaltando su papel imprescindible en la eficiencia y la versatilidad del despliegue de aplicaciones.

3.1.3 | Docker Swarm

Docker Swarm [13] es una herramienta de orquestación de contenedores implementada en Docker. Actualmente se la denomina Swarm mode y no se debe confundir con Docker Classic Swarm [14], el cual ha sido deprecado.

Con Swarm Mode se disponen de las siguientes características:

- **Escalabilidad automática**: Permite escalar automáticamente el número de réplicas del contenedor para adaptarse a la demanda.
- **Alta disponibilidad**: Asegura que la aplicación siga funcionando en caso de que falle algún nodo.
- **Facilidad de uso**: Está integrado con las herramientas de Docker, por lo tanto, su uso una vez se maneja Docker, no es complejo.

- **Seguridad**: Ofrece opciones de seguridad para proteger los datos y comunicación entre nodos.

Swarm Mode es una herramienta interesante para poder gestionar de forma fácil y rápida un stack de contenedores, pero fue descartado para el desarrollo de este trabajo por los siguientes motivos:

- Necesidad de control total de la infraestructura y personalización: Tiene limitaciones para personalizarlo y para el desarrollo de este trabajo se necesitó un control total de las piezas.
- Puede no ser compatible con sistemas 3rd party y en este caso de uso se debe poder conectar aplicaciones de terceros sin estas restricciones.
- Docker Swarm Classic está discontinuado y dio paso a Swarm mode. Nada respalda que Docker deje de dar soporte a Swarm mode en favor de otros sistemas como Kubernetes.
- Swarm mode implementa gran parte de las características que se pretenden desarrollar en este trabajo, se decidió no usarlo para visualizar las capas de más bajo nivel, entender cómo funcionan y que sirviese con fines educativos.
- Para sistemas con necesidades de escalabilidad extrema, es mejor usar Kubernetes, por ende, bajo esta premisa, Swarm mode está limitado.

3.1.4 | Kubernetes

Un aspecto importante en el desarrollo de aplicaciones, es el poder estimar si la aplicación necesita escalar, es decir, si necesita ajustar su capacidad para manejar la carga de trabajo requerida. CRANE nace con la necesidad de servir como herramienta para que el proceso de escalado sea lo más automático posible, por lo tanto, después de descartar Swarm mode, se analizó el uso de otra herramienta que brinde esta característica y entre estas, una de las más conocidas es Kubernetes.

Kubernetes [11] es una plataforma de orquestación de contenedores de código abierto que facilita la implementación, la escalabilidad y la gestión de aplicaciones contenerizadas en entornos de producción. Fue desarrollado por Google y luego donado a la Cloud Native Computing Foundation (CNCF). Kubernetes brinda un conjunto de herramientas para automatizar el despliegue, la escalabilidad, la administración y la operación de aplicaciones en contenedores. En

pocas palabras, es una herramienta perfecta para cualquier DevOps y se necesito compararlo con Docker para entender sus principales similitudes y diferencias.

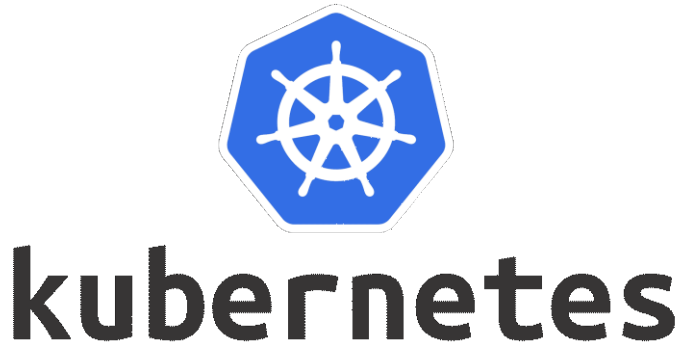


Figura 4. Logo oficial de Kubernetes

Tabla 4: Comparación entre Docker y Kubernetes

	DOCKER	KUBERNETES
Naturaleza	Plataforma para desarrollar, enviar y ejecutar aplicaciones en contenedores.	Plataforma de orquestación que administra y escala contenedores, compatible con diversos motores de contenedores.
Funcionalidad	Se centra en la creación y ejecución de contenedores	Centrada en la orquestación de múltiples contenedores, coordinando su implementación, escalado, monitoreo y mantenimiento.
Escalabilidad	Ofrece funciones básicas de escalado.	Está centrada en el escalado de aplicaciones y servicios, facilitando la administración de clústeres de contenedores.

	DOCKER	KUBERNETES
Orquestación	Orquestación básica con Docker SWARM	Conjunto completo de características para orquestar contenedores, como la gestión de la implementación, actualización, escalado y distribución de cargas de trabajo.
Portabilidad	Portabilidad mediante las imágenes de los contenedores.	Facilita la portabilidad de las aplicaciones al brindar una capa de abstracción sobre la infraestructura subyacente, permitiendo ejecutar aplicaciones en diferentes entornos de la nube y locales.

Luego de analizar esta tabla, se concluyó que Kubernetes es una solución más completa y avanzada para el manejo de contenedores, integra todas las herramientas que se buscan automatizar en este trabajo. Es por esto que se realizó un segundo análisis para determinar si Kubernetes cumplía con los objetivos de CRANE, entre ellos: simplicidad, bajo consumo de recursos, simular escalabilidad en entornos locales no clusterizados, enfoque, documentación, facilidad de implementación, costos, mantenimiento y curva de aprendizaje.

Tabla 5: Comparación realizada para decidir la tecnología a utilizar

Característica	Docker	Kubernetes
Simplicidad	Accesible y fácil de instalar y configurar.	Requiere configuraciones más complejas y conocimiento previo.
Recursos	Funciona bien en configuraciones de hardware limitado, es ideal para uso personal.	En general requiere un entorno de servidor o nube, más costoso y complejo.

Característica	Docker	Kubernetes
Escalabilidad	Adecuada para demostraciones y proyectos pequeños, permite entender la expansión básica.	Diseñado para escalar automáticamente y manejar grandes volúmenes de tráfico y servicios.
Enfoque	Centrado en contenedores individuales, ideal para aprender los principios básicos de los contenedores.	Enfocado en la orquestación a nivel de sistema, maneja complejidades de interacciones entre contenedores.
Comunidad y soporte	Gran cantidad de tutoriales, guías y comunidades en línea que facilitan el aprendizaje autónomo.	Comunidad enfocada en operaciones de escala empresarial y optimización de despliegues complejos.
Implementación	Ideal para desarrollo local y pruebas rápidas.	Mejor utilizado en entornos de producción o donde se necesite alta disponibilidad.
Costo y mantenimiento	Menor costo operativo inicial, menos mantenimiento debido a su simplicidad.	Más costoso con relación a la operación y mantenimiento debido a su escala y complejidad.
Curva de aprendizaje	Más suave, permite a los usuarios concentrarse en aprender sin distracciones adicionales.	Más empinada, puede ser abrumadora para los nuevos usuarios debido a sus múltiples componentes y capacidades.

Dado el caso de uso al que apunta Crane, se concluyó que usar Kubernetes sería mucho más complejo, costoso y difícil de utilizar para entornos locales no clusterizados, por lo tanto, Docker fue elegido como la herramienta de virtualización de Crane.

3.1.5 | Docker Compose

Docker Compose [15] es una herramienta para construir y ejecutar aplicaciones multi-contenedor. Simplifica el control de toda la pila de aplicaciones y facilita la gestión de servicios, redes y volúmenes en un único y comprensible archivo de configuración YAML [16], que es un formato de serialización de datos legible por humanos que se utiliza comúnmente para la configuración y representación de datos estructurados.

Docker Compose permite ejecutar archivos con un solo comando, creando y poniendo en marcha todos los servicios según la configuración definida en los YAML. Funciona en todos los entornos: producción, desarrollo, pruebas, como también en flujos de trabajo de integración continua (CI). También cuenta con comandos para gestionar todo el ciclo de vida de la aplicación:

- **Iniciar, detener y reconstruir servicios**
- **Ver el estado de los servicios en ejecución**
- **Transmitir la salida de registro de los servicios en ejecución**
- **Ejecutar un comando único en un servicio**

Los archivos de configuración son fáciles de compartir, permitiendo que otra persona pueda replicar el entorno en su sistema. En lo que respecta al uso de recursos del sistema, se cachea la configuración usada para crear un contenedor. Cuando se reinicia un servicio que no ha cambiado, se reutilizan los contenedores existentes, ayudando a hacer cambios en el entorno mucho más rápido y con menos costo.

Para su implementación en distintos entornos una de sus características más importantes es que se admiten variables en el archivo YAML. Por lo tanto, se puede personalizar la composición para diferentes entornos o usuarios

Para el desarrollo de este trabajo, se necesitó poder instanciar “servicios”, esto quiere decir, crear contenedores, iniciarlos y asignarles una configuración que los interconecte entre sí permitiendo que cada uno cumpla una función y entre ellos cooperen para lograr sinergia.

Se analizaron diferentes alternativas para solventar esto:

- **Uso Swarm mode o Kubernetes**: Estos poseen alternativas para escalar, instanciar, eliminar contenedores. Pero se descartó su uso previamente.
- **Mediante la interfaz de línea de comandos:** Sería indispensable introducir manualmente el comando de Docker, lo cual no es práctico dado que se busca automatización. Una alternativa sería desarrollar un script en bash que incluya los detalles del contenedor a implementar y su configuración, pero esto estaría restringido al uso en sistemas Unix limitando el alcance de CRANE a este sistema operativo.
- **Por último, Docker Compose** al ser un archivo de configuración de Docker, no importa el sistema operativo donde se ejecute, solamente necesita tener Docker instalado y para interactuar con el solo se necesita ejecutar el comando `docker-compose up`, únicamente sería necesario automatizar en qué momento se ejecuta ese comando.

3.1.5.1 | Estructura de un archivo docker-compose.yml

Como se visualizó previamente, la configuración de los contenedores usando Docker Compose se realiza en el docker-compose.yml, con este archivo se puede definir, qué imágenes de Docker utilizar, con qué versiones, que puertos exponer, que volúmenes montar y como conectarlos entre sí.

```yaml
version: '3.8'

services:
  web:
    image: nginx:latest
    ports:
      - "8080:80"
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf
    networks:
      - mynetwork

  db:
    image: mysql:5.7
    environment:
      MYSQL_ROOT_PASSWORD: example
      MYSQL_DATABASE: mydatabase
    volumes:
      - db_data:/var/lib/mysql
    networks:
      - mynetwork

volumes:
  db_data:

networks:
  mynetwork:
    driver: bridge
```

Figura 5. Ejemplo de un archivo docker-compose.yml

Este archivo de configuración de ejemplo se compone de las siguientes partes:

- **Version:** Indica la versión del formato de Docker Compose que se utiliza en el archivo y especifica qué características y sintaxis son compatibles y como se debe interpretar el archivo.
- **Services**: Cada servicio define un contenedor que forma parte de la aplicación. En el ejemplo hay 2 servicios “web” y “db” que se componen de **image**, que es la imagen a usar, **ports**, que son los puertos a mapear, **volumes**, que son los volúmenes que monta esa imagen y **environment** qué son las variables de entorno, como en este caso las credenciales del servicio db.
- **Volumes**: Esta sección define las diferentes fuentes de almacenamiento que pueden ser usadas por los servicios, ya sea para montarse, guardar datos y compartirlos.
- **Networks**: Se definen las redes que utilizarán los servicios, en este caso se define una red llamada mynetwork en modo bridge, este modo crea una red privada interna en el host donde se ejecutan los contenedores y cada contenedor que se inicia en esta red obtiene una IP en ese rango, para así poder comunicarse con los otros contenedores de la misma red.

Configurar contenedores usando docker-compose es mucho más simple que hacerlo de la forma tradicional por línea de comandos, por lo tanto, es el modo de instanciar contenedores que se implementó a lo largo de este trabajo.

3.1.6 | Uso de servicios en Docker

Un servicio en Docker se refiere a un conjunto de contenedores en ejecución que comparten la misma configuración y son manejados como una entidad única. Esta gestión centralizada de los contenedores facilita su despliegue, actualización y escalamiento de un modo más coordinado y automático.

La configuración de los servicios se realiza mediante archivos docker-compose.yml o utilizando la API de Docker Swarm, donde se especifican las propiedades deseadas del servicio y luego Docker se encarga de mantener el estado del mismo según lo declarado en la configuración.

Los servicios pueden ser escalados fácilmente incrementando o disminuyendo el número de réplicas. Esto es vital si se quiere lograr una mayor disponibilidad y capacidad de respuesta de la aplicación ante cambios en la demanda, además su capacidad de balanceo automático se encarga de distribuir las solicitudes de los usuarios entre las distintas réplicas del servicio para optimizar el uso de los recursos y mejorar el rendimiento de la aplicación.

Para la comunicación entre estos servicios Docker brinda la posibilidad de crear redes virtuales que permitan interconectarlos entre sí con más seguridad y eficiencia en la comunicación de los diferentes servicios dentro de una misma red.

Ventajas de los Servicios:

- Configurar y gestionar servicios usando Docker es simple gracias a las herramientas, comandos y documentación que brinda.
- La capacidad de actualizar y escalar servicios automáticamente reduce la necesidad de intervención manual y facilita el mantenimiento continuo.
- Se asegura que cada instancia del contenedor se ejecute en un ambiente idéntico, minimizando los problemas relacionados con diferencias en los entornos de desarrollo, staging y producción.

Desventajas de los Servicios:

- Si bien Docker promueve la portabilidad, ciertas características y funcionalidades pueden depender de la configuración específica del sistema operativo que lo contiene o de la infraestructura.
- La gestión de los servicios puede ser difícil en sistemas grandes, en especial sin las herramientas de orquestación correctas.
- La compartición del kernel del sistema operativo entre contenedores puede traer riesgos de seguridad si no se gestionan de forma correcta los distintos permisos y accesos.

En este trabajo se decidió su implementación por la simplicidad de uso y efectividad, es más accesible y gestionable para los fines educativos que se proponen. Hace posible centrarse en el aprendizaje y la experimentación dentro del ámbito de las aplicaciones contenerizadas sin la sobrecarga de tener que manejar configuraciones complejas y estructuras de orquestaciones más avanzadas.

3.1.7 | Seguridad en contenedores

La seguridad en contenedores es un aspecto muy importante porque comparten el mismo kernel del sistema operativo y recursos con el host y otros contenedores. Al momento de crear nuevos contenedores se debe tener en cuenta distintas consideraciones y buenas prácticas para asegurarlos porque si no se pueden ingresar vulnerabilidades, violaciones de datos u otros problemas críticos de seguridad en el equipo y demás contenedores que se encuentren en el sistema. Las siguientes recomendaciones ayudan a mitigar muchos de estos problemas y se aplicaron para realizar la parte práctica de este trabajo con el fin de se cumplan los objetivos de seguridad propuestos:

1. Utilizar imágenes oficiales o de confianza desde repositorios verificados. Siempre se tiene que verificar que las mismas estén actualizadas y que provengan de fuentes confiables.

2. Utilizar herramientas de escaneo de vulnerabilidades para analizar las imágenes en busca de posibles problemas de seguridad. Herramientas como Clair, Trivy (utilizado para la parte práctica de este trabajo) o Anchore Engine pueden ayudar a identificar vulnerabilidades conocidas.

3. Utilizar versiones “Latest” preferentemente para mantener las imágenes y contenedores actualizados con las últimas correcciones de seguridad, siempre y cuando no generen algún tipo de incompatibilidad en el stack de servicios.

4. Aplicar el principio de menor privilegio, ejecutando contenedores con los permisos mínimos necesarios para realizar sus funciones. Restringiendo el acceso a recursos y privilegios que no se necesitan.

5. Utilizar límites de recursos para evitar que un contenedor consuma demasiados recursos del sistema y así evitar inconsistencia por parte del Host

6. Asegurar que los contenedores estén bien aislados entre sí y del sistema host, limitando las interacciones entre estos y utilizando políticas de red adecuadas.

3.2 | Proxy Reverso

3.2.1 | ¿Qué es un Proxy Reverso?

Un proxy reverso [17] es un tipo de servidor proxy que actúa como intermediario entre los clientes y uno o más servidores web. A diferencia de un proxy regular que se posiciona entre el cliente y el servidor destino, un proxy reverso se sitúa entre el cliente y uno o más servidores web.

Cuando un cliente hace una solicitud, esta se dirige primero al proxy reverso. Este es quien determina a qué servidor web debe enviarse la solicitud en función de diversos criterios, como las reglas de enrutamiento, la carga del servidor, la disponibilidad, etc. Una vez que se toma esta decisión, el proxy reverso reenvía la solicitud al servidor web correspondiente, obtiene la respuesta del servidor y luego la envía de vuelta al cliente.

Herramientas como Swarm mode y Kubernetes implementan la funcionalidad de Proxy que redirige las peticiones, pero al no usar ninguno de estos, surge la necesidad de usar un proxy reverso que provea los siguientes beneficios:

1. Distribuir el tráfico entre varios contenedores para mejorar el rendimiento y la disponibilidad.
2. Almacenar en caché las respuestas del contenedor para reducir la carga en los servidores y mejorar los tiempos de respuesta para las solicitudes repetidas.
3. Actuar como un punto de defensa frente a ataques, ocultando la estructura interna de la red y filtrando las solicitudes maliciosas.
4. Enrutar el tráfico en función de diversos criterios, como la URL, el encabezado HTTP, la geolocalización, etc.

Existen muchas opciones actualmente cuando se necesita elegir un Proxy Reverso, que varían dependiendo de las necesidades de uso y por los que se tuvo que realizar un análisis detallado para determinar cuál es el más conveniente para este caso de uso, para entender un poco mejor esto se repasaron los proxys más populares, para luego decidir por uno de ellos en particular:

- **Traefik:**
    - **Principal Característica:** Integración nativa con contenedores.
    - **Descripción:** Traefik es un proxy reverso moderno que está diseñado específicamente para entornos de contenedores y microservicios. Su mayor ventaja es la facilidad de uso y la integración automática con plataformas de orquestación como Docker y Kubernetes, haciendo más simple la gestión de rutas y servicios en sistemas dinámicos. Fue elegido para este trabajo por su integración nativa con los contenedores y su facilidad para enrutar nuevas instancias y por lo tanto se observa más en detalle su funcionalidad más adelante.

- **Nginx:**
    - **Principal característica:** Eficiencia en la gestión de conexiones concurrentes.
    - **Descripción:** NGINX destaca no sólo como un servidor web de gran rendimiento, sino también como un proxy reverso ampliamente utilizado, gracias a su arquitectura basada en eventos. Esta estructura le capacita para gestionar numerosas conexiones simultáneas en un modelo de baja utilización de memoria, convirtiéndolo en una opción ideal para sitios web de alto tráfico y aplicaciones en tiempo real.

- **Apache HTTP Server:**
    - **Principal característica:** Flexibilidad en la configuración.
    - **Descripción:** Aunque es más conocido como servidor web, Apache también puede funcionar eficazmente como proxy reverso. Su fortaleza radica en la amplia modularidad y la capacidad de personalización, que permite a los usuarios configurar un entorno de servidor web adaptado a sus necesidades particulares.

- **HAProxy:**
    - **Principal Característica:** Alta disponibilidad y fiabilidad en balanceo de carga**.**
    - **Descripción:** HAProxy es un software de balanceo de carga y proxy TCP/HTTP de código abierto que se destaca por su solidez y estabilidad. Es ampliamente utilizado en entornos críticos donde el rendimiento y la escalabilidad son factores necesarios, como en aplicaciones de alta demanda y entornos de comercio electrónico.

- **Squid:**
    - **Principal Característica:** Capacidades avanzadas de cacheo.
    - **Descripción:** Squid es un proxy de caché de código abierto que también puede funcionar como proxy reverso. Es conocido por sus eficaces capacidades de cacheo, que ayudan a reducir el tiempo de respuesta y la carga en los servidores de origen. También ofrece opciones robustas para el filtrado de contenido y el control de acceso para mejorar la seguridad y la gestión del tráfico web.

3.2.2 | Traefik, el proxy de Crane

Traefik [18] es un moderno proxy reverso y balanceador de carga diseñado principalmente para entornos de contenedores y microservicios. Su función principal es dirigir el tráfico de red entre los clientes y los servicios que se ejecutan en una infraestructura distribuida.

La principal característica que distingue a Traefik es su capacidad para adaptarse automáticamente a los servicios en ejecución en un entorno. Esto significa que, en lugar de requerir una configuración estática, Traefik puede detectar dinámicamente los servicios disponibles y enrutar el tráfico hacia ellos sin necesidad de intervención manual.

Figura 6. Logo oficial de Traefik

Durante la búsqueda de un Proxy Reverso para el caso de uso de este trabajo, se decidió usar Traefik principalmente por estos 5 puntos:

1. Se integra de forma mucho más simple con plataformas populares de orquestación de contenedores como Docker, Kubernetes y Swarm, haciendo más fácil su implementación en estos entornos.

2. Puede enrutar el tráfico basándose en diversas reglas dinámicas, como la URL, el encabezado HTTP o las etiquetas Docker/Kubernetes.

3. Distribuye el tráfico entre múltiples instancias de un servicio para mejorar la disponibilidad y el rendimiento.

4. Puede encargarse de forma automática de gestionar los certificados SSL/TLS requeridos para habilitar el cifrado de extremo a extremo en las comunicaciones.

5. Está diseñado para ser escalable y de alto rendimiento, haciendo que sea adecuado para entornos de producción con altos volúmenes de tráfico.

Traefik utiliza 4 conceptos que se deben de comprender antes de continuar avanzando en la parte práctica, estos conceptos son *EntryPoints, Routers, Middlewares y Services*. Se explicará brevemente el significado de cada uno:

- **EntryPoints**: Son los puntos de entrada de red en Traefik. Definen el puerto que recibirá los paquetes y si escucharán para el protocolo TCP o UDP.

- **Routers**: Se encarga de conectar las solicitudes entrantes a los servicios que pueden manejarlas.

- **Middlewares**: Junto a los routers, los middlewares pueden modificar las solicitudes o respuestas antes de que se envíen al destino.

- **Services**: Son responsables de configurar cómo llegar a los servicios reales que eventualmente manejarán las solicitudes entrantes.

Router de borde o Edge Router

Traefik es un enrutador de borde, esto significa que es la puerta de entrada a tu plataforma y que intercepta y enruta cada solicitud entrante: conoce toda la lógica y cada regla que determina qué servicios manejan qué solicitudes (basado en la ruta, el host, encabezados, etc…).

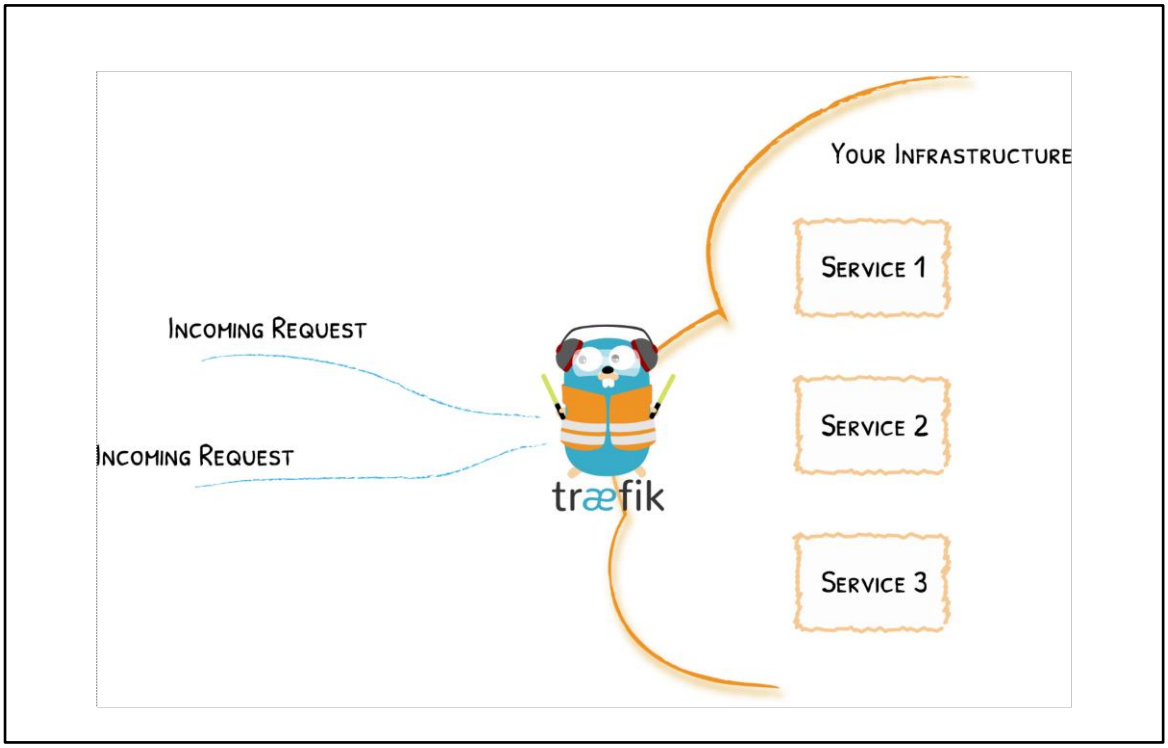


Figura 7. Infraestructura con Traefik. [18]

Servicio de Auto descubrimiento.

A diferencia de los enrutadores de borde tradicionales o proxies reversos, que requieren un archivo de configuración estático con todas las rutas posibles hacia los servicios, Traefik automatiza este proceso extrayendo la información directamente de los servicios en despliegue. Al momento de implementar los servicios, se adjunta metadatos específicos que indican a Traefik las características de las solicitudes que cada servicio está capacitado para gestionar.

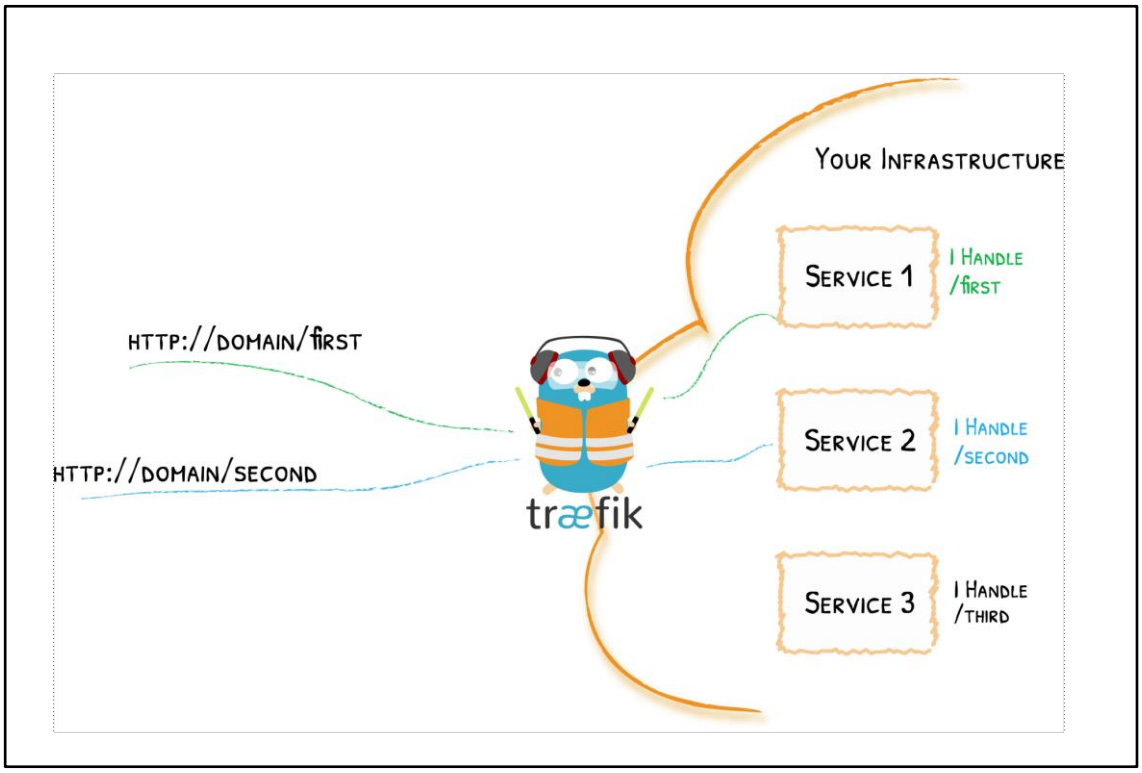


Figura 8. Configuración descentralizada con Traefik. [18]

3.2.3 | Desventajas del uso de Traefik

1. **Complejidad en entornos no contenerizados**: Al estar optimizado para entornos de contenedores, puede ser más complejo de configurar en entornos no contenerizados y puede ser menos adecuado para organizaciones que no utilizan estas tecnologías.

2. **Curva de aprendizaje inicial**: Si bien Traefik es más fácil de configurar que otros proxies populares, puede requerir tiempo para familiarizarse con sus características y funcionamiento.

3.2.4 | Comparación con NGINX

Como se mencionó en puntos anteriores, Nginx es otro de los proxies más populares que se usan actualmente, de hecho, la versión inicial de Crane se utilizó este proxy para el enrutamiento, pero se terminó decantando por Traefik al analizar las diferentes características que se mencionan en la tabla siguiente.

Tabla 6: Comparación entre NGINX y Traefik

Característica	NGINX	Traefik
Propósito principal	Servidor web de alto rendimiento y proxy reverso para contenido estático y balanceo de carga.	Proxy reverso y balanceador de carga, optimizado para microservicios y contenedores.
Configuración	Requiere configuración manual usando archivos, proceso que puede ser complejo y propenso a errores.	Configuración dinámica y automática utilizando metadatos y etiquetas de contenedores, aumentando la escalabilidad y simplificando la gestión.
Automatización	No posee capacidades de automatización integrada para el manejo de servicios en contenedores.	Automatización avanzada para el descubrimiento y enrutamiento de servicios en entornos de contenedores.
Integración con contenedores	Integración manual necesaria con plataformas como Docker o Kubernetes.	Diseñado para integrarse naturalmente con plataformas de orquestación de contenedores como Docker, Kubernetes y Swarm.
Características específicas	Amplia gama de funcionalidades como el balanceo de carga, proxy reverso, caché y soporte para múltiples protocolos.	Enfoque en características relevantes para contenedores, como enrutamiento dinámico, integración con servicios de descubrimiento y gestión automática de certificados SSL/TLS.

3.2.5 | Configuración de Traefik

Configurar Traefik es un proceso bastante sencillo. Como se vio anteriormente en el apartado donde se explica el formato del archivo docker-compose.yml, todo lo que se necesita hacer es

definir una serie de atributos que se toman como configuración de los servicios. Estos atributos permiten especificar cómo se quiere que Traefik maneje el tráfico entrante y cómo interactúa con los servicios en contenedores.

```
versión: '3'
services:
  reverse-proxy:
    # The official v2 Traefik docker image
    image: traefik:v2.11
    # Enables the web UI and tells Traefik to listen to docker
    command: --api.insecure=true --providers.docker
    ports:
      # The HTTP port
      - "80:80"
      # The Web UI (enabled by --api.insecure=true)
      - "8080:8080"
    volumes:
      # So that Traefik can listen to the Docker events
      - /var/run/docker.sock:/var/run/docker.sock
```

Figura 9. Ejemplo de configuración de Traefik en un docker-compose.yml

En el siguiente ejemplo, se tiene un archivo docker-compose.yml con la configuración mínima para implementar Traefik

- **Services**: Especifica qué servicios se instancian.
  - **Reverse-proxy:** nombre asignado al servicio
  - **Image**: imagen de Traefik versión 2.11.
  - **Command:** especifica los argumentos que se le pasarán al contenedor cuando se inicia, en este ejemplo: `--api.insecure=true` habilita la interfaz web y `--providers.docker` para que escuche eventos de Docker.
  - **Ports:** mapea los puertos del host contenedor. En este caso el puerto 80 del host, se mapea al puerto 80 del contenedor (utilizado para el tráfico http) y el puerto 8080 del host se mapea al puerto 8080 del contenedor (para el acceso a la interfaz web de Traefik).
  - **Volumes:** monta un volumen en el contenedor. En este caso se monta el socket de Docker `/var/run/docker.sock:/var/run/docker.sock` en el contenedor para que Traefik pueda escuchar eventos de Docker y ajustar su configuración en tiempo real según los cambios en los contenedores.

Esta es una configuración básica de Traefik, más adelante se analiza la configuración que se usó en CRANE y de qué servicios se compone.

3.3 | Python

3.3.1 | Definición y características del lenguaje

Python es un lenguaje de programación de alto nivel, interpretado y multipropósito, desarrollado por Guido van Rossum y lanzado por primera vez en 1991. Este lenguaje es muy querido principalmente en la comunidad académica y en la industria tecnológica por su sintaxis clara y legible, características que lo hacen particularmente accesible para principiantes, y, además, permite a los profesionales escribir código de forma rápida y eficiente.

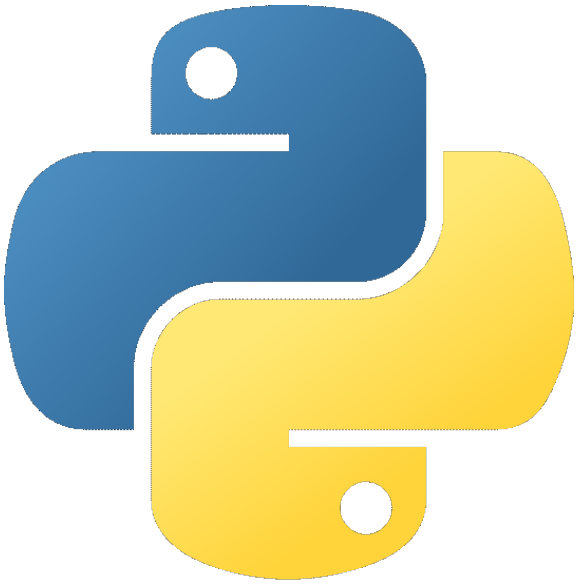


Figura 10. Logo oficial de Python

Principales características de Python:

- **Sintaxis simple y legible**: Tiene una sintaxis sencilla que se asemeja al lenguaje natural, facilitando así la comprensión y el aprendizaje del código, más aún para quienes se inician en la programación.
- **Interpretado e interactivo**: Es un lenguaje interpretado, ejecuta el código línea por línea mediante un intérprete. Esta característica favorece un ciclo de desarrollo rápido, permitiendo la prueba y depuración interactiva del código directamente en la consola o mediante un entorno de desarrollo integrado (IDE).
- **Multipropósito**: Se emplea en una diversidad de campos, entre ellos desarrollo web, ciencia de datos, inteligencia artificial, automatización de tareas, scripting y desarrollo de juegos, demostrando su versatilidad.
- **Gran comunidad y ecosistema de bibliotecas**: Tiene una extensa comunidad de desarrolladores y un buen ecosistema de bibliotecas y frameworks que simplifican la ejecución de múltiples tareas sin necesidad de reinventar la rueda.
- **Portabilidad**: Es compatible con múltiples plataformas como Windows, macOS y Linux, es ideal para el desarrollo de aplicaciones multiplataforma.

3.3.2 | Python contra otros lenguajes

Al momento de seleccionar un lenguaje de programación para el desarrollo de este trabajo, se tuvieron en cuenta múltiples factores cruciales. Se realizó un análisis exhaustivo para determinar las funcionalidades esenciales que el sistema requeriría. Los criterios clave que se consideraron son:

1. **Backend**: El núcleo de CRANE implica la integración de una lógica compleja por medio de una API REST. Dada nuestra experiencia previa con Python y TypeScript, Python se perfilaba como una opción ventajosa para facilitar el desarrollo de esta sección crucial.
2. **Interacción con Docker**: Es esencial comunicarse eficazmente con Docker para realizar tareas como escalar, desescalar e instanciar contenedores. Al investigar las diferentes formas de lograr esta integración mediante lenguajes de programación, se encontró que

Docker ofrece un SDK (Software Development Kit) compatible tanto con Go como con Python.

3. **Manejo de archivos de configuración**: La gestión adecuada de archivos de configuración es esencial para el funcionamiento del sistema. Python no solo demostró ser una opción robusta para conectarse con Docker, también es conocido por su eficacia en el manejo de archivos haciéndolo una elección idónea para esta tarea.

Teniendo en cuenta estos criterios fundamentales y nuestras competencias previas, Python fue seleccionado como el lenguaje principal para el desarrollo del proyecto. Esta decisión no solo capitaliza nuestra experiencia existente, sino que también maximiza la eficiencia del desarrollo gracias a las capacidades integradas de Python y su compatibilidad con las tecnologías involucradas.

3.3.3 | Frameworks para el desarrollo web

En el ámbito del desarrollo web, el uso de frameworks [19] se ha convertido en un elemento casi indispensable por los múltiples beneficios que ofrecen. Estas librerías brindan herramientas predefinidas que permiten crear aplicaciones más rápido y con mayor eficiencia. Funciones como el **enrutamiento**, la gestión de sesiones y la validación de formularios vienen integrados generalmente de forma predeterminada, achicando la necesidad de desarrollar código desde cero y aumentando la productividad. Muchas librerías y **frameworks** promueven el uso de **patrones de diseño** y **buenas prácticas de programación**. Esto no solo ayuda a tener un código limpio y organizado, sino que también facilita la colaboración entre equipos y mejora la mantenibilidad y escalabilidad de las aplicaciones a largo plazo.

La **seguridad** es otro aspecto necesario. Algunos frameworks integran funcionalidades de seguridad que protegen contra ataques comunes como CSRF, XSS y la inyección SQL, mitigando los riesgos y simplificando la implementación de medidas de protección. Con respecto a la **escalabilidad**, los frameworks y librerías están diseñados para facilitar el crecimiento de las aplicaciones tanto en funcionalidad como en tráfico de usuarios sin afectar el rendimiento. Se pueden encontrar herramientas como el almacenamiento en caché y la gestión de sesiones que ayudan a manejar las aplicaciones en entornos de alta demanda de forma más eficiente.

La **comunidad activa** que rodea a estos recursos es un pilar importantísimo. El soporte continuo, sumado a una gran cantidad de documentación, tutoriales y soluciones a problemas comunes proporcionados por otros desarrolladores, acelera el proceso de desarrollo y reduce el tiempo de inactividad, haciendo que las librerías de desarrollo web sean una elección estratégica para nuevos proyectos.

Los frameworks más populares para el desarrollo web en Python son:

1. **Django**
2. **Flask**
3. **FastApi**

Para decidir cual se debería usar en este trabajo, se tomó como punto de partida las necesidades del proyecto, es decir, ¿qué se precisa desarrollar a nivel Backend?

Para realizar toda la operativa con contenedores se requirió una API REST que sirva los endpoints a los que apuntar. Por este motivo, se buscó una librería que pudiese acelerar el desarrollo de una API REST sin grandes complicaciones. Se examinó que podían ofrecer las librerías anteriormente mencionadas y se hizo la comparación de estas.

En primera instancia, quedó descartado Django por ser un Framework que aportaba muchas más herramientas de las que se necesitaban, como por ejemplo:

- Una interfaz de administración que permite a los desarrolladores gestionar los modelos de base de datos de forma sencilla. Esta interfaz brinda una GUI predefinida para realizar tareas comunes como agregar, editar y eliminar registros de la base de datos.

- Un sistema de formularios que facilita la creación, validación y procesamiento de formularios web.

- Django template language (DTL) qué consiste en un sistema de templates para el Backend.

  En este caso de uso no se requirió ningún motor de templates porque estos endpoints serán consumidos a futuro por un componente Frontend el cual se desarrollará independiente de la API, ni tampoco sistemas de formularios e interfaces GUI para administrar los modelos. Descartado Django, solo quedaban dos opciones Flask y FastAPI, dos frameworks muy interesantes por los cuales se tuvo que realizar una comparación detallada.

Los puntos principales que se posicionaron a FastAPI como el framework Web de Crane:

- **Rendimiento**: Se diseñó para ofrecer un rendimiento excepcionalmente rápido, más que nada comparado con Flask.
- **Tipado y validación de datos**: Integra Pydantic, una biblioteca de serialización y validación de datos, para definir modelos de datos y realizar validaciones automáticas de datos de entrada y salida en las rutas. Esto hace que sea más fácil y seguro manipular datos comparado con Flask, donde los desarrolladores tienen que diseñar la validación de datos manualmente o utilizar bibliotecas externas.
- **Asincronía**: Se diseñó desde cero para admitir operaciones asincrónicas, permitiendo manejar solicitudes HTTP de un modo eficiente y escalable mediante el uso de corutinas asincrónicas de Python. Flask, por otro lado, no tiene un soporte asincrónico nativo y puede requerir extensiones o bibliotecas adicionales para admitir operaciones asincrónicas.
- **Tipo de API**: Se diseñó para la creación de APIs RESTful, ofreciendo características y herramientas específicas para este propósito, como la generación automática de documentación interactiva (OpenAPI / Swagger).

3.3.4 | Fastapi, el framework de Crane

FastAPI es un web framework moderno y rápido (de alto rendimiento) para construir APIs con Python 3.8+ basado en las anotaciones de tipos estándar de Python.

Sus características principales [20] son:

- **Rapidez**: Alto rendimiento, a la par con **Nodos** y **Go**.
- **Rápido de programar**: Incrementa la velocidad de desarrollo entre 200% y 300%.
- **Menos errores**: Reduce los errores humanos (de programador) aproximadamente un 40%.
- **Intuitivo**: Gran soporte en los editores con auto completado en todas partes. Gasta menos tiempo debugging.
- **Fácil**: Está diseñado para ser fácil de usar y aprender. Gastando menos tiempo leyendo documentación.
- **Corto**: Minimiza la duplicación de código. Múltiples funcionalidades con cada declaración de parámetros. Menos errores.

- **Robusto**: Crea código listo para producción con documentación automática interactiva.
- **Basado en estándares**: Basado y totalmente compatible con los estándares abiertos para APIs: OpenAPI [21] (conocido previamente como Swagger) y JSON Schema [22].

3.3.5 | Uvicorn

Uvicorn es un servidor **ASGI (Asynchronous Server Gateway Interface)** [23] de alto rendimiento para Python. Fue creado por el mismo equipo que desarrolló FastAPI y está diseñado para ser utilizado como el servidor web subyacente para aplicaciones FastAPI, aunque también puede utilizarse con otros marcos web ASGI.

Principales características:

- Está diseñado para ser muy rápido y escalable, aprovechando la asincronía y la concurrencia de Python para manejar grandes volúmenes de solicitudes de forma eficiente.
- Es compatible con el estándar ASGI, permitiendo que pueda integrarse de forma fácil con cualquier framework web o aplicación que sea compatible con ASGI, como FastAPI, Starlette, Django Channels, entre otros.
- Aprovecha el procesamiento asíncrono para gestionar múltiples solicitudes de la mejor manera y lo convierte en una buena opción para aplicaciones que demandan alto rendimiento y escalabilidad.
- Se puede configurar de forma fácil utilizando la línea de comandos o mediante un archivo de configuración Python, facilitando su uso y adaptación dependiendo los requisitos del proyecto.

3.3.6 | Python on Whales para interactuar con Docker

"Python on Whales" es una biblioteca de Python que facilita la interacción con Docker y la que se eligió en este trabajo

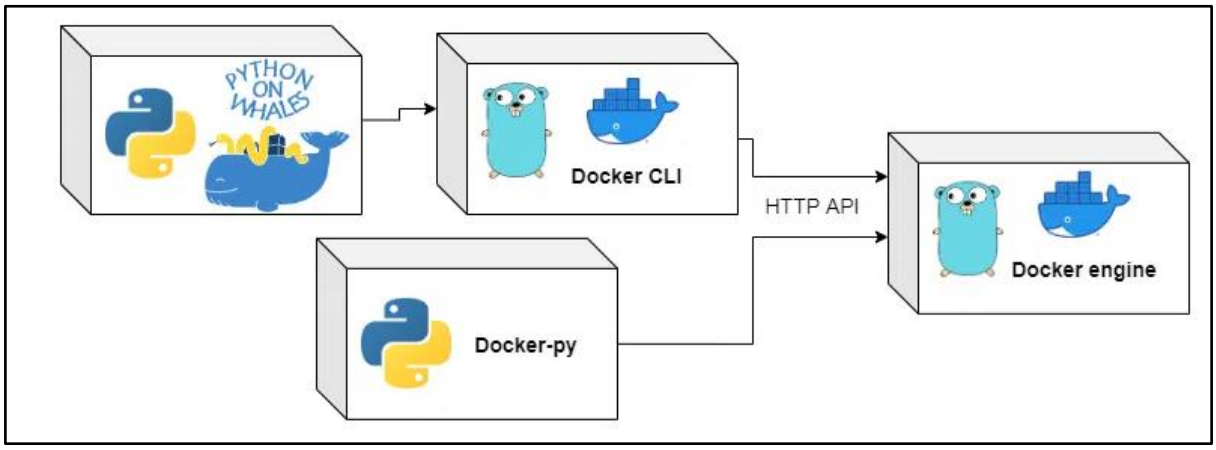


Figura 11. Interacción de Python on Whales con Docker-CLI. [24]

Como se puede ver en la figura 11, existen 2 librerías nativas que interactúan con el Docker Engine, Docker CLI programado en GO, en el que se apoya Python on Whales y Docker-py que está programado directamente en Python.

Al utilizar "Python on Whales", se puede aprovechar las capacidades de Docker de un modo más simple y programático, esto facilitó realizar tareas como crear y administrar contenedores, administrar volúmenes y redes, y orquestar servicios directamente desde el código escrito en Python.

Docker-py es otra biblioteca que ofrece una API para conectarse con el demonio de Docker y realizar operaciones como crear, administrar y supervisar contenedores desde aplicaciones Python. Aunque tanto Docker-py como "Python on Whales" tienen funcionalidades similares al interactuar con Docker, existen algunas razones por las cuales se decidió utilizar “Python on Whales" (más adelante mencionado como POW) en vez de docker-py.

Primero que nada, POW, se diseñó para tener una sintaxis más intuitiva y expresiva, permitiendo tener un código más limpio y legible al comunicarse con Docker, además de obtener una capa de abstracción de más alto nivel que puede simplificar tareas comunes y reducir la cantidad de código que se necesita para lograr ciertos objetivos. Si bien Docker-py es un proyecto con mayor cantidad de colaboradores, se encontró en POW una librería muy prometedora y que permite la interacción con archivos Docker Compose, aspecto fundamental en este proyecto, además de que se actualiza constantemente aplicando nuevas funcionalidades y mejoras. Si bien al momento del

desarrollo de este trabajo aun no llego a su primera versión estable, durante el funcionamiento y las pruebas de Crane no se tuvo ninguna limitación ni inconveniente para el caso de uso que requerido y por lo que se apostó a esta opción para continuar con el desarrollo.

3.3.7 | Pydantic para la definición y control de esquemas

Pydantic es una biblioteca de Python que brinda herramientas para la validación de datos y la serialización de modelos. Fue creada para facilitar la validación de datos de entrada en aplicaciones Python y para ayudar en la creación de clases de datos Pythonic que sean claras, concisas y fáciles de entender.

Características de Pydantic:

1. Permite definir modelos con tipos de datos y restricciones, y luego utilizar estos modelos para validar y deserializar los datos de entrada. Esto asegura que los mismos cumplan con ciertos criterios antes de ser procesados por la aplicación.
2. Puede ser utilizado para serializar modelos de datos en diferentes formatos, como JSON, YAML, o mensajes de texto, haciendo más simple el intercambio de los mismos entre diferentes sistemas y aplicaciones.
3. Es compatible con los tipos de datos nativos de Python, pero también con tipos de datos personalizados definidos por el usuario, obteniendo una gran flexibilidad al definir modelos.
4. Se integra de forma más simple con frameworks web como FastAPI y con Object-Relational Mapping (ORM) como SQLAlchemy, facilitando la validación de datos en aplicaciones web y BBDD.

3.3.8 | PyYaml para la creación de archivos YAML

PyYAML es una biblioteca de Python que permite leer y escribir archivos en formato YAML (YAML Ain't Markup Language) [25], el cual es un formato de serialización de datos legible por humanos

que se utiliza comúnmente para la configuración, la serialización y el intercambio de información estructurada entre diferentes aplicaciones.

Muchas aplicaciones y herramientas utilizan archivos de configuración en formato YAML debido a su legibilidad y simplicidad, es especial Docker, por lo tanto, esta librería permite, desde la API REST de Crane, la personalización y la gestión de los archivos docker-compose desde el código Python. Más adelante se observa cómo Crane genera estos archivos y cómo realiza PyYAML [26] el volcado de información en ellos.

3.4 | Autenticación mediante JWT

3.4.1 | ¿Qué es JWT?

JSON Web Token (JWT) [27][28] es un estándar abierto (RFC 7519) que define una forma compacta y autónoma de transmitir información de un modo seguro entre partes como un objeto JSON. Esta información puede ser verificada y confiable porque está firmada digitalmente. Los JWT pueden ser firmados utilizando un secreto (con el algoritmo HMAC) o un par de claves pública/privada utilizando RSA o ECDSA.

Aunque los JWT pueden ser encriptados para proporcionar también secreto entre las partes, se decidió usar simplemente tokens firmados. Los tokens firmados pueden verificar la integridad de las afirmaciones contenidas en él, mientras que los tokens encriptados ocultan esas afirmaciones a otras partes. Cuando los tokens son firmados utilizando pares de claves pública/privada, la firma también certifica que solo la parte que posee la clave privada es la que lo firmó.

JWT se usa en los siguientes escenarios:

- **Autorización**: Este es el escenario más común para el uso de JWT. Una vez que el usuario ha iniciado sesión, cada solicitud posterior incluirá el JWT y esto le permitirá al usuario

acceder a rutas, servicios y recursos que están permitidos con ese token. El inicio de sesión único (Single Sign On) es una característica que utiliza ampliamente JWT en la actualidad, debido a su pequeña sobrecarga y su capacidad de ser fácilmente utilizado en diferentes dominios.

- **Intercambio de información**: Los JSON Web Tokens son una buena alternativa para transmitir información de un modo más seguro entre partes. Debido a que los JWT pueden ser firmados utilizando pares de claves pública/privada se puede estar seguro de que los remitentes son quienes dicen ser. Además, como la firma se calcula utilizando el encabezado (**header**) y la carga útil (**payload**) permite verificar que el contenido no haya sido manipulado.

JWT se compone de tres partes separadas por puntos:

1) HEADER

El Header se compone de 2 partes. El tipo de Token que es JWT en este caso y el algoritmo de firmado

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Luego esta parte se codifica en base 64 para formar la primera parte del JWT.

2) PAYLOAD

La segunda parte del token es el payload, que contiene las afirmaciones. Estas son declaraciones sobre una entidad (típicamente, el usuario) y datos adicionales. Hay tres tipos de afirmaciones: afirmaciones registradas, públicas y privadas.

- **Afirmaciones registradas**: son un conjunto de campos estándar que no son obligatorios, pero se recomiendan para proporcionar un conjunto de afirmaciones útiles e interoperables. Algunas de ellas son: iss (emisor), exp (tiempo de expiración), sub (sujeto), aud (audiencia), y otras.

```
{
  "iss": "https://miaplicacion.com",
  "sub": "usuario123",
  "exp": 1613039600
}
```

- **Afirmaciones públicas**: Estas pueden ser definidas a voluntad por aquellos que utilizan JWTs. Sin embargo, para evitar colisiones, deberían ser definidas en el Registro de Tokens JSON Web de la IANA [29] o ser definidas como un URI que contenga un espacio de nombres resistente a colisiones.

```
{
  "iss": "https://miaplicacion.com",
  "sub": "usuario123",
  "exp": 1613039600,
  "name": "John Doe",
  "email": "john.doe@example.com"
}
```

- **Afirmaciones privadas**: son las afirmaciones personalizadas creadas para compartir información entre partes que acuerdan usarlas y que no son ni afirmaciones registradas ni públicas.

```
{
  "iss": "https://miaplicacion.com",
  "sub": "usuario123",
  "exp": 1613039600,
  "role": "admin",
  "custom_data": {
    "id": 123456,
    "email": "usuario@example.com"
  }
}
```

Luego esta parte se codifica en base 64 para formar la primera parte del JWT.

3) SIGNATURE

Para crear la parte de la firma, se debe tomar el encabezado codificado, la carga útil codificada, un secreto, el algoritmo especificado en el encabezado y firmar eso.

Por ejemplo, si se quiere usar el algoritmo HMAC SHA256, la firma se creará del siguiente modo:

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret
)
```

La firma se utiliza para verificar que el mensaje no fue cambiado en el camino y, en el caso de tokens firmados con una clave privada, también puede verificar que el remitente del JWT es quien dice ser.

La salida son tres cadenas Base64-URL separadas por puntos que pueden ser fácilmente transmitidas en entornos HTML y HTTP, siendo más compactas en comparación con estándares basados en XML como SAML.

La figura 12 muestra un JWT que tiene el encabezado y la carga útil previamente codificados, y está firmado con un secreto.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiw
iaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

Figura 12. Estructura de un JWT

3.4.2 | Autenticación sin estado

Que JWT es "stateless" significa que el servidor que emite el token y valida su autenticidad no necesita mantener ningún estado de sesión para el usuario. En lugar de eso, toda la información necesaria para autorizar y autenticar al usuario se encuentra dentro del propio token JWT. Esto contrasta con los sistemas basados en sesiones, donde el servidor necesita mantener información de sesión para cada usuario que interactúa con él.

En un sistema stateless con JWT:

- El servidor emite un token JWT después de que el usuario se autentica exitosamente.
- El token JWT contiene toda la información necesaria sobre el usuario y sus permisos.
- En cada solicitud posterior, el cliente (generalmente un navegador web o una aplicación móvil) incluye este token JWT en la cabecera de la solicitud, usando el esquema Bearer.

- El servidor valida el token JWT utilizando una clave secreta conocida sólo por el servidor. No necesita consultar una base de datos centralizada o mantener información de sesión para el usuario porque toda la información necesaria está contenida en el token.

- Esto hace que el servidor sea "stateless" porque no necesita mantener ningún estado de sesión para el usuario.

3.4.3 | Ventajas y desventajas de usar JWT

Las ventajas de usar JWT como mecanismo de seguridad son múltiples:

- **Más compactos:** JSON es menos verboso que XML, por ende, cuando se codifica, un JWT es más pequeño que un token SAML. Esto permite que JWT sea una buena opción para ser transmitido en entornos HTTP.

- **Más seguro:** Pueden ser firmados y encriptados para verificar la integridad y la confidencialidad de los datos transmitidos. Además, al no depender de cookies, son menos vulnerables a ciertos ataques de seguridad como Cross-Site Request Forgery (CSRF). Los algoritmos de firma son:

    a. **RS256:** Es un algoritmo asimétrico que utiliza funciones criptográficas RSA con SHA-256. En estos algoritmos asimétricos hay dos claves: una clave pública y una clave privada que debe mantenerse en secreto. Tiene sus variantes con SHA-384 y SHA-512

    b. **ES256**: También es un algoritmo asimétrico al igual que el anterior, pero utiliza funciones de hash ECDSA basadas en la curva elíptica P-256 junto con SHA-256, es más eficiente en cuanto a computación y tamaño de clave comparado con RSA. Tiene sus variantes con P-384 y SHA-384 y P-521 y SHA-512

    c. **HS256**: Es un algoritmo simétrico, es decir, solo hay una clave privada que debe mantenerse en secreto, y esta se comparte entre las dos partes. Se debe tener

cuidado para evitar que la clave sea comprometida. Utiliza funciones criptográficas HMAC con SHA-256. Tiene sus variantes con SHA-384 y SHA-512

- **Autenticación y Autorización sin Estado:** JWT permite la autenticación y la autorización sin la necesidad de mantener el estado de la sesión en el servidor. Esto significa que no es indispensable almacenar información de sesión en el servidor, haciéndolo mucho más escalable y eficiente.

- **Portabilidad y Flexibilidad:** Debido a que los tokens JWT son compactos y autocontenidos, son portátiles y pueden ser fácilmente compartidos entre diferentes sistemas y plataformas. Esto los hace ideales para entornos distribuidos y microservicios.

- **Interoperabilidad:** JWT es un estándar abierto que es ampliamente adoptado y compatible con una diversidad de lenguajes de programación y frameworks.

Las desventajas:

- **Tamaño del Token:** Un exceso de información en el token JWT puede provocar su incremento considerable, generando una carga mayor en la red y mayores demandas de almacenamiento tanto en el cliente como en el servidor.

- **Vulnerabilidad a Ataques de Fuerza Bruta:** En caso de robo o interceptación del token JWT, un atacante podría intentar descifrarlo o falsificar su firma. Por esto, resulta crucial adoptar buenas prácticas de seguridad para proteger y transmitir los tokens JWT de forma segura.

- **Falta de Invalidación Dinámica:** Una vez que se emite un token JWT y se firma, no se puede invalidar antes de que expire naturalmente. Esto significa que, si un token es comprometido, el atacante puede tener acceso a recursos protegidos hasta que el token expire.

- **Complejidad en la Gestión de Claves:** La gestión de claves para firmar y validar tokens JWT puede ser compleja, en especial para entornos distribuidos con múltiples servicios y aplicaciones.

3.5 | Prometheus

3.5.1 | ¿Qué es Prometheus?

Prometheus [30] es un conjunto de herramientas de monitoreo y alerta de sistemas de código abierto creado originalmente en SoundCloud. Desde su inicio en 2012, muchas empresas y organizaciones adoptaron el uso de Prometheus y el proyecto cuenta con una comunidad de desarrolladores y usuarios muy grande. Ahora es un proyecto de código abierto y se mantiene de forma independiente de cualquier empresa. Para enfatizar esto, y para aclarar la estructura de gobierno del proyecto, Prometheus se unió a la Cloud Native Computing Foundation en 2016 como el segundo proyecto alojado, después de Kubernetes.

Prometheus recopila y almacena sus métricas como datos de series temporales, es decir, la información de las métricas se almacena con la marca de tiempo en la que fue registrada, junto con pares clave-valor opcionales llamados etiquetas.

Figura 13. Logo oficial de Prometheus

Características Principales:

1. **Modelo de Datos Multidimensional:** Utiliza un modelo de datos multidimensional basado en pares clave-valor para almacenar series temporales de datos de monitoreo. Esto

permite etiquetar cada métrica con metadatos adicionales y facilitar la consulta y el análisis de datos.

2. **Recolección de Métricas:** Tiene un mecanismo de recolección de métricas altamente productivo que puede integrarse de forma fácil con sistemas y aplicaciones existentes por medio de clientes específicos para diversos lenguajes de programación y plataformas.
3. **Lenguaje de Consulta PromQL:** Integra un lenguaje de consulta poderoso llamado PromQL, que permite a los usuarios realizar consultas flexibles y avanzadas sobre los datos de monitoreo almacenados en su base de datos.
4. **Alertas y Notificaciones:** Además de la recolección y el almacenamiento de métricas, Prometheus permite definir reglas para monitorear estas métricas y disparar alertas en caso de que se cumpla alguna.

3.5.2 | Arquitectura y componentes principales de Prometheus

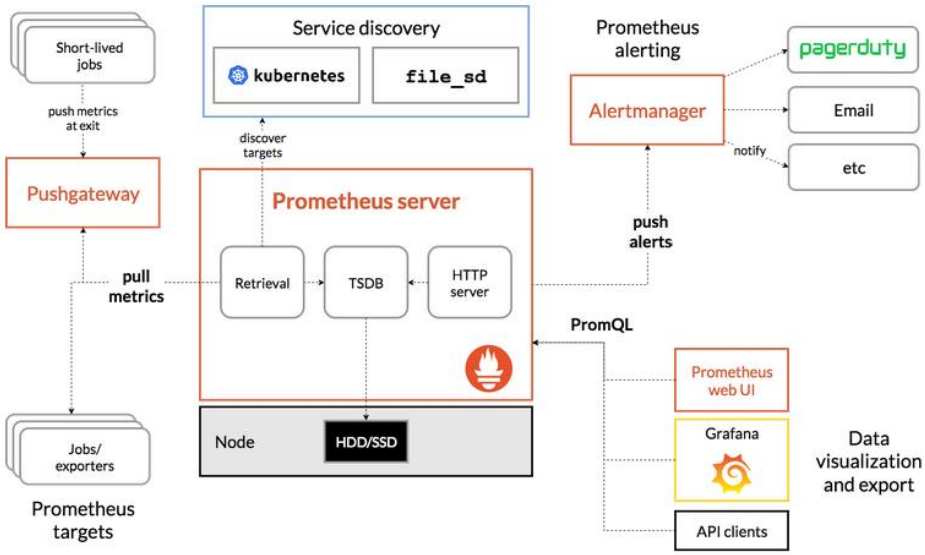


Figura 14. Diagrama de la arquitectura de Prometheus

El ecosistema de Prometheus abarca varios componentes, muchos de estos son opcionales. Sin embargo, en este trabajo, solo se utilizaron los siguientes:

- El servidor principal de Prometheus, encargado de recopilar y almacenar datos de series temporales.

- Un gestor de alertas para manejar alertas.

3.5.3 | Configuración y recolección de métricas

La configuración de Prometheus está basada en un fichero llamado prometheus.yml, en el cual se definen los intervalos y la recopilación de métricas. Este archivo yml se divide en varias partes y cada una hace referencia a cada bloque de la configuración.

```yaml
global:
  scrape_interval:     15s
  evaluation_interval: 15s

scrape_configs:
  - job_name: 'node_exporter'
    static_configs:
      - targets: ['localhost:9100']

  - job_name: 'my_service'
    static_configs:
      - targets: ['my-service:8080']

alerting:
  alertmanagers:
  - static_configs:
    - targets: ['alertmanager:9093']
```

Figura 15. Ejemplo de un fichero prometheus.yml básico

- La sección global especifica las configuraciones que se aplicarán a todas las métricas que Prometheus recopile.

- scrape_interval es el intervalo en el cual Prometheus recopila métricas de los objetivos (15 seg).
- evaluation_interval define cada cuanto tiempo se evalúan las reglas de alerta (15 seg).

- La sección scrape_configs define qué métricas recopilar y de dónde. Cada job_name representa un objetivo con las mismas configuraciones de scrape. En concreto, el job_name node_exporter recopilará métricas de localhost en el puerto 9100 donde se estará ejecutando un servicio node_exporter.
- La sección alerting especifica cómo se manejan las alertas generadas en Prometheus.
  - Se define que las alertas deben enviarse a un administrador de alertas (alertmanager) que está escuchando en el puerto 9093. Esto quiere decir que cuando se activa una regla de alerta en Prometheus, enviará esa alerta al servicio de Alertmanager que se está ejecutando en 'alertmanager' en el puerto 9093.

3.6 | Alert Manager

3.6.1 | Definición y control de alertas

Alertmanager [31] es un componente de Prometheus que maneja las alertas generadas y permite aplicar ciertas acciones. Para este proyecto fue fundamental porque permite accionar ante cualquier variación del rendimiento o disponibilidad de los servicios activos. La forma de funcionamiento que tiene Alert Manager depende de las reglas definidas en Prometheus, esté en pocas palabras es quien se encarga analizar si se cumple alguno de los umbrales definidos en el rules.yaml y accionar con la configuración que le provee el usuario. Para este caso de uso se utilizó una comunicación mediante webhook en la que Alert Manager le avisa a Crane mediante un llamado al endpoint receptor de Alertas cuando detecta alguna anomalía.

La integración de Prometheus con este sistema trajo muchos beneficios porque permite abstraerse de varias complicaciones al momento de definir un sistema de alertas, a continuación, se mencionan las principales ventajas por las cuales se decidió hacer uso de este complemento en la arquitectura de Crane:

- Recibe alertas de Prometheus u otros sistemas de monitoreo que están configurados para enviar alertas a través de él.
- Puede descartar alertas idénticas y agrupar alertas similares en notificaciones únicas. Esto ayuda a reducir el ruido generado por múltiples alertas similares.
- Puede enviar alertas a receptores específicos basados en ciertos criterios como etiquetas de alerta, prioridades, o configuraciones definidas por el usuario. Esto permite enviar notificaciones de alerta a los equipos de operaciones correctos o canales de comunicación adecuados.
- Puede enviar notificaciones de alerta a diferentes canales de comunicación, como correos electrónicos, sistemas de chat (Slack, Microsoft Teams), sistemas de ticketing (PagerDuty, JIRA), y otros servicios de notificación mediante integraciones customizadas.
- Ofrece la capacidad de silenciar temporalmente o suprimir ciertas alertas para evitar la sobrecarga de notificaciones durante períodos de mantenimiento planificados o incidentes conocidos.

3.6.2 | Configuración mediante YAML

La configuración de Alertmanager sigue una estructura similar a la que usa Prometheus. Ambas herramientas utilizan archivos YAML para definir las reglas, receptores y demás configuraciones que se tienen disponibles.

La similitud en el formato de configuración entre estas herramientas no solo simplifica su implementación y mantenimiento, sino que también, permitió una integración más fluida dentro del ecosistema de monitoreo. Usando esta configuración, se pueden definir reglas específicas para el manejo de alertas, establecer cómo y a quién se deben enviar las notificaciones, y

personalizar otros aspectos del sistema de alerta según las necesidades específicas de cada entorno.

En la figura 16 se presenta un ejemplo de configuración de Alertmanager y se analiza cómo se definen las reglas de alerta, cómo se configuran los receptores para las notificaciones y cómo esta configuración se diferencia de la de Prometheus, esto permite tener una comprensión más detallada de cómo Alertmanager contribuye al proceso de monitoreo y gestión de sistemas críticos [32].

```
global:
  resolve_timeout: 5m

route:
  group_by: ['alertname', 'severity']
  group_wait: 10s
  group_interval: 5m
  repeat_interval: 3h
  receiver: 'email_notifications'

receivers:
- name: 'email_notifications'
  email_configs:
  - to: 'tu_correo@example.com'
    from: 'alertmanager@example.com'
    smarthost: 'smtp.example.com:25'
    auth_username: 'tu_usuario'
    auth_password: 'tu_contraseña'

inhibit_rules:
- source_match:
    severity: 'critical'
  target_match:
    severity: 'warning'
  equal: ['alertname', 'dev', 'instance']
```

Figura 16. Ejemplo de configuración de Alertmanager

- La sección **global** especifica las configuraciones que se afectan al comportamiento global de alertmanager.

- **resolve_timeout** establece el tiempo máximo que Alertmanager esperará para resolver una alerta antes de que expire. Por lo tanto, si en 5 min la alerta no se resuelve, alertmanager la marcará como no resuelta.

- La sección **route** define reglas de enrutamiento para dirigir las alertas a receptores específicos.
  - **Group_by**: agrupa las alertas que comparten las mismas etiquetas, en este caso se agruparan por nombre de alerta y gravedad.
  - **Group_wait**: Es el tiempo de espera antes de enviar una alerta después de recibir la primera instancia de esa alerta.
  - **Group_interval**: Es el tiempo durante el cual se agruparán las alertas. Después de este intervalo, se enviará una notificación para cada grupo.
  - **Repeat_interval**: Define cada cuanto tiempo se repetirá una notificación para una alerta no resuelta.
  - Receiver: Es el receptor al que se enviarán las alertas que coincidan con esa ruta.
- La sección **receivers** define los receptores a los que se pueden enviar las alertas.
  - **name**: Es el nombre del receptor.
  - **email_configs**: Define la configuración para enviar alertas por correo electrónico.
- **Inhibit_rules**: Define reglas de inhibición que previenen que unas alertas disparen otras alertas.
  - **Source_match**: son las condiciones que deben cumplir las alertas para ser consideradas como fuente de inhibición.
  - **Target_match**: Son las condiciones que deben cumplir las alertas que se están evaluando para ser inhibidas.
  - **equal**: Especifica las etiquetas que deben coincidir exactamente entre la alerta fuente y la alerta objetivo para que se aplique la inhibición. En este caso las alertas

con una severidad “critical” inhibirán alertas de severidad de advertencia si comparten las mismas etiquetas de “alertname”, “dev”, “instance”.

3.6.2 | Comunicación mediante Webhook

Uno de los principales desafíos que se enfrentó durante el desarrollo de este trabajo es la posibilidad de poder comunicar el contenedor donde se ejecuta Alertmanager contra la API de Crane, la cual se encuentra alojada un nivel más arriba en el sistema Host y la cual en principio no es accesible desde el contenedor de Alertmanager por su encapsulamiento propio de la tecnología Docker. Para este problema docker ofrece el hostname ***host.docker.internal*** el cual permite, especificando el puerto donde corre Crane establecer una comunicación Container->Host, esto abrió un abanico de posibilidades, pero principalmente permitió lanzar las alertas y que Crane reciba la misma para aplicar las acciones necesarias.

Ahora bien, Alertmanager dispone de muchas maneras de notificar alertas y se detalla cada una a continuación:

- **Servicios de mensajería instantánea** como Slack, Microsoft Teams, Discord, etc. Esto permite que se puedan recibir notificaciones en tiempo real en diversos canales de comunicación.
- **Sistemas de gestión de incidencias** como JIRA, Service Now, PagerDuty, entre otros. Esto permite que las alertas generen automáticamente tickets de incidencia, facilitando el seguimiento y la resolución de problemas.
- **Sistemas de monitoreo y visualización** como Grafana, Prometheus, etc. Esto permite crear paneles de control y alertas personalizadas en función de los datos de monitoreo que se obtienen en tiempo real.
- **Correo electrónico**, se puede especificar la dirección del destinatario, y además otros detalles de configuración como el servidor SMTP, el puerto y las credenciales de autenticación si es necesario.

- **Solicitudes HTTP** a una URL específica utilizando webhooks. Esto permite integrar Alertmanager con servicios y aplicaciones que pueden recibir y atender solicitudes HTTP.

Para el caso de uso de este trabajo, se decidió que **CRANE** use los Webhooks como forma de notificación.

Cuando una alerta alcanza cierto nivel de gravedad o cumple ciertos criterios definidos por Crane, Alertmanager genera una solicitud HTTP POST a la URL del webhook especificada en la configuración. Esta solicitud tiene la información detallada sobre la alerta, como su nombre, gravedad, etiquetas asociadas y cualquier otra información que sea relevante.

El receptor del webhook puede ser cualquier servicio capaz de atender solicitudes HTTP, en este caso Crane, quien recibe y procesa la solicitud, realizando acciones como escalar, desescalar o reiniciar el host, además se puede extender a todas las funcionalidades mencionadas más arriba.

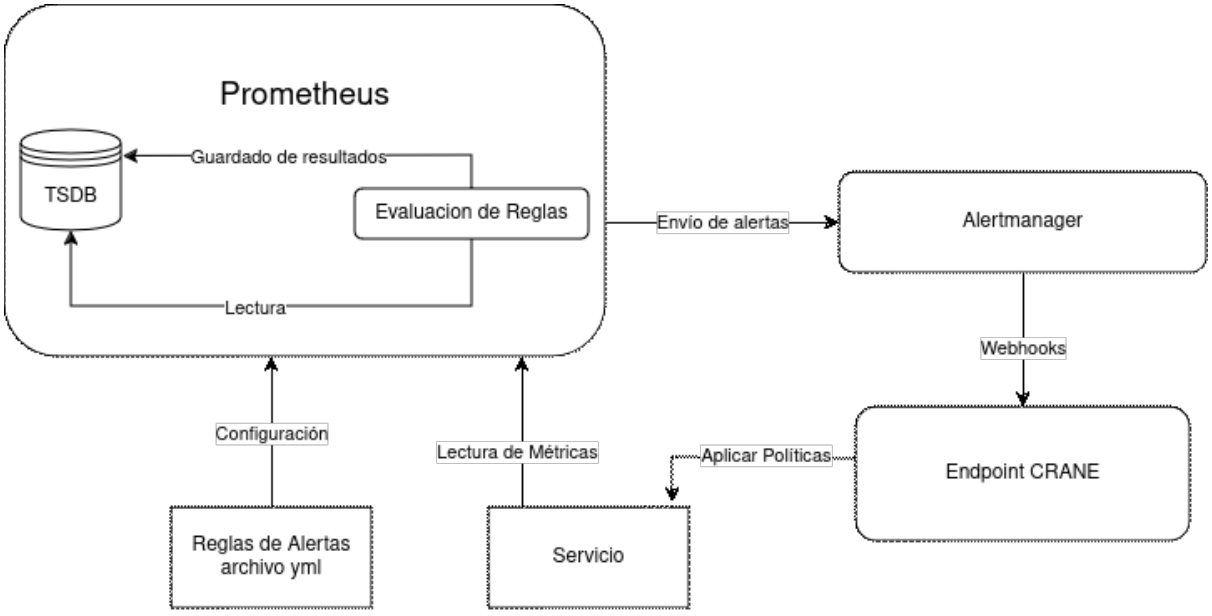


Figura 17. Diagrama del estado de una alerta

3.6.2 | Configuración de Alertmanager usando Webhooks

La forma de configurar Webhooks en alertmanager es la siguiente:

```
receivers:
- name: 'webhook_notifications'
  webhook_configs:
  - url: 'http://tu_webhook.com/alertas'
```

Figura 18. Ejemplo de configuración de webhooks en Alertmanager

Se define un receptor que utilice el tipo webhook_configs y se configura el endpoint que recibirá la alerta.

Cuando una alerta alcanza el estado en el que debe ser notificada (por ejemplo, supera un umbral definido), Alertmanager la procesa según las reglas de enrutamiento y la envía al receptor configurado.

Una vez que Alertmanager determina que una alerta debe ser enviada al receptor de webhook, genera una solicitud HTTP POST a la URL especificada en la configuración del webhook. Esta solicitud tiene información sobre la alerta en el cuerpo del mensaje, generalmente en formato JSON.

Más adelante se detalla el comportamiento de CRANE al recibir una alerta proveniente del Webhook de Alert Manager

3.7 | Open Policy Agent

3.7.1 | Definición y evaluación de políticas

Open Policy Agent (OPA) es un proyecto de código abierto que brinda una plataforma para la autorización y evaluación de políticas. OPA está diseñado para ayudar a los equipos de desarrollo a diseñar y aplicar políticas de seguridad, acceso y otros tipos de políticas en sus aplicaciones y servicios.

OPA se basa en la definición de unas políticas en un lenguaje declarativo llamado Rego. Estas políticas pueden abordar muchos casos de uso, como el control de acceso, validación de datos, autorización de red y todo se lleva a cabo mediante una api expuesta a la aplicación, que se puede

consultar para verificar, entre varias funciones, si el usuario tiene permisos para acceder a un recurso.

Casos de uso de OPA:

1. Control de acceso basado en políticas (RBAC, ABAC).
2. Validación de solicitudes de API y validación de datos.
3. Implementación de políticas de seguridad como política de red y política de cifrado.
4. Validación y aplicación de políticas de cumplimiento y gobierno.

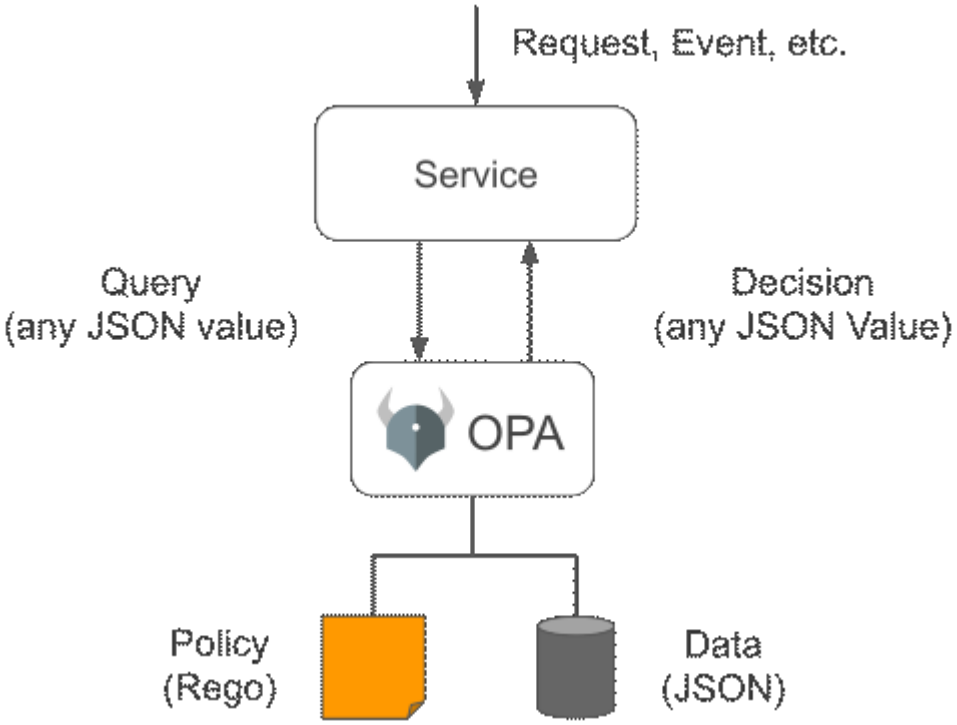


Figura 19. Diagrama de validación de políticas.

En este caso de uso, se utilizó OPA para validar si un usuario posee permisos de acceso a un recurso y también para decidir qué acción ejecutar ante una alerta proveniente de Alertmanager, la interacción de Crane con el servidor OPA es la siguiente:

- Hay dos desencadenantes:
    a. Se recibe una solicitud HTTP de un usuario intentando acceder a un recurso
    b. Se recibe una alerta proveniente del Webhook de Alertmanager

- Crane consulta al servidor de OPA enviando el nombre de la política que debe evaluar y los parámetros de la misma:
    a. Para el caso del acceso, se envían los roles del usuario, el recurso al que intenta acceder y el método HTTP.
    b. Para el caso de la alerta, se envía el tipo de alerta y la severidad
- El servidor de OPA consulta las políticas y verifica que respuesta dar antes los parámetros que recibe.
- El servidor de OPA envía una respuesta al servicio para que pueda accionar.

3.7.2 | RBAC mediante políticas

El modelo de control de acceso conocido como RBAC (Role Based Access Control) [33] estructura y administra los permisos de los usuarios según los roles que desempeñan en una determinada aplicación o sistema.

En el modelo RBAC los permisos se asignan a los roles específicos y luego a los usuarios se les asignan roles. Esto simplifica mucho la gestión de permisos porque en lugar de asignar permisos a cada usuario, solo se le asigna un rol, que puede tener varios permisos.

Componentes del modelo RBAC:

- **Roles**: Son un conjunto de actividades o funciones que un usuario puede desarrollar en un sistema. ej. Supervisor, analista.
- **Permisos**: Son las acciones o actividades específicas que un usuario puede realizar en el sistema. Estos permisos se asignan a roles. ej. Lectura, modificación, eliminación.
- **Asignación de Roles**: Los roles como se describió anteriormente se asignan a los usuarios. Un usuario puede tener más de un rol. ej. El usuario Pepe tiene el rol de administrador y este rol posee los permisos de lectura, modificación y eliminación.
- **Política de acceso**: Define las reglas que gobiernan qué roles tienen acceso a qué recursos y que acciones pueden realizar. Asegurando que los usuarios solo tengan acceso a los recursos y datos necesarios para realizar sus funciones. ej. Una política de acceso sería

que, solo pudiesen acceder a los datos personales de un usuario de la plataforma, los usuarios con rol supervisor.

Para el desarrollo de este trabajo se usó este modelo como mecanismo de protección para ciertas acciones que puede realizar el usuario, entre ellas, que solo un administrador pueda eliminar contenedores. Combinando este modelo con Open Policy Agent, Crane resuelve el acceso de los usuarios a cada recurso.

3.8 | SQLite

3.8.1 | Qué es SQLite y sus características principales

SQLite [34] es un motor de base de datos relacional [35] ligero, autónomo y de código abierto. Es ampliamente utilizado en aplicaciones móviles, sistemas embebidos y aplicaciones de escritorio debido a su simplicidad, eficiencia y portabilidad. Su principal diferencia con sistemas de gestión de bases de datos (DBMS) como MySQL o PostgreSQL, es que SQLite no opera como un servidor independiente, sino que se incorpora directamente en la aplicación que lo utiliza. Esto significa que no requiere configuración ni administración de un proceso de servidor separado.

Figura 20. Logo oficial de SQLite

Características de SQLite:

- La base de datos SQLite se almacena en un solo archivo, facilitando su transporte y respaldo.
- Admite transacciones ACID [36] (Atomicidad, Consistencia, Aislamiento y Durabilidad), garantizando integridad en los datos como en los casos de fallas del sistema.
- No requiere un servidor independiente para funcionar.
- No hay necesidad de configurar usuarios, privilegios u otros ajustes de seguridad, los controles de acceso se implementan a nivel de archivo.
- Soporta la mayoría de las características del lenguaje SQL estándar, permite realizar consultas complejas y operaciones de manipulación de datos.

Tabla 7: Comparativa entre los DBMS más usados

Características	SQLite	MariaDB	PostgreSQL
Arquitectura	Sin servidor	Cliente-Servidor	Cliente-Servidor
Lenguaje SQL	Completo	Completo	Completo
Soporta transacciones	Sí	Sí	Sí
ACID	Sí	Sí	Sí
Multiusuario	No	Sí	Sí
Escalabilidad	Limitada	Alta	Alta
Conexiones Concurrentes	Limitada a solo lectura	Alta	Alta
Instalación/configuración	Fácil	Moderada	Moderada
Soporte ORM	Si	Si	Si
Licencia	Dominio público	GPLv2	PostgreSQL (BSD, MIT)

3.8.2 | Bases de datos ligeras basadas en archivos

Como se ha mencionado previamente, hay varios tipos de bases de datos con características distintivas y para casos de uso diferente. Al seleccionar la base de datos para este trabajo, se consideraron varios puntos que llevaron a elegir SQLITE como el motor de BBDD de Crane:

1. **No necesita un proceso independiente en ejecución para operar**. Su base de datos se guarda en un solo archivo, haciendo más sencilla su integración y distribución con la aplicación que emplea Docker. En este caso, no se requirió instalar ningún software adicional ni lanzar otro contenedor con una base de datos, como sería necesario con, por ejemplo, MariaDB.
2. SQLite es una **base de datos basada en un solo archivo**, esto la hace muy portable. Se puede integrar la base de datos directamente en la imagen Docker de la aplicación, haciendo más fácil la implementación y garantizando que la aplicación y su base de datos estén siempre sincronizadas y sean fácilmente desplegables en diferentes entornos. Durante el ciclo de pruebas, solo con enviar el archivo de BBDD, se puede replicar los datos exactos en otro entorno.
3. Está diseñado para ser **ligero y eficiente en recursos**. Tiene un bajo consumo de memoria y CPU comparado con otras bases de datos más robustas. Factor crítico para el desarrollo de este trabajo es que fuera eficaz en el consumo de los recursos.
4. Es **fácil de configurar y usar**. No requiere configuraciones complicadas ni administración especializada.
5. Es **compatible con la mayoría de los sistemas operativos y plataformas**, otro punto importante porque el propósito de Crane es que sea multiplataforma.

3.8.2 | Integración mediante SQLAlchemy

SQLAlchemy [37] es un **ORM** [38] distribuido mediante una biblioteca de Python que permite una abstracción de alto nivel para trabajar con bases de datos relacionales. Es compatible con una gran cantidad de motores de bases de datos, entre estos SQLite.

Figura 21. Logo de SQLAlchemy

Para Integrar SQLAlchemy con SQLite se realizaron los siguientes pasos:

1. **Instalación:** Es necesario instalar SQLAlchemy y el módulo de SQLite usando el gestor de paquetes de Python pip

```
pip install SQLAlchemy
```

2. **Configuración:** La conexión a una base de datos SQLite se configura utilizando una URL de conexión. Esta URL indica a SQLAlchemy dónde se encuentra el archivo de la base de datos SQLite en el sistema de archivos. Una vez que se arma la URL de conexión, se puede usar para crear un objeto Engine, que se encargará de manejar la conexión con la base de datos.

```
from sqlalchemy import create_engine

# Path archivo de la base de datos SQLite
database_path = 'sqlite:///ruta/a/tu/base/de/datos.db'

# Creacion objeto Engine para conectarse a la base de datos SQLite
engine = create_engine(database_path)
```

3. **Definición de modelos:** Los modelos se definen utilizando la clase declarative_base y clases Python que heredan de esta.

```
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Integer, String

Base = declarative_base()
```

```
class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(String)
    age = Column(Integer)
```

Cada vez que se define una clase, esta se corresponde con una tabla, en la cual se detalla cada una de las columnas y tipos.

4. **Crear o acceder a la sesión:** SQLAlchemy usa sesiones para conectarse con la base de datos. Por lo tanto, para crear una sesión se debe realizar lo siguiente:

```
from sqlalchemy.orm import sessionmaker

Session = sessionmaker(bind=engine)
session = Session()
```

5. **Realizar consultas y operaciones CRUD**: Con la sesión configurada, se pueden realizar consultas y operaciones de creación, lectura, actualización y eliminación en la base de datos. Por ejemplo:

```
# Insertar un nuevo usuario
new_user = User(name='John', age=30)
session.add(new_user)
session.commit()

# Consultar todos los usuarios
users = session.query(User).all()
for user in users:
    print(user.name, user.age)

# Actualizar un usuario
user = session.query(User).filter_by(name='John').first()
user.age = 31
session.commit()

# Eliminar un usuario
session.delete(user)
session.commit()
```

CAPÍTULO 5 - Crane y su implementación

5.1 | Arquitectura de Crane

En las últimas décadas, el ecosistema del desarrollo de software ha sufrido cambios para escribir, testear y desplegar aplicaciones, lo que ha provocado que cada una de estas características se transforme en un dominio especializado por sí mismo. En respuesta a esto, han surgido tendencias como DevOps, que buscan disminuir la distancia entre el desarrollo y el despliegue.

Basándose en esta premisa, nace CRANE, un sistema que se enfoca en mejorar el proceso de despliegue de aplicaciones contenerizadas, pero también su monitoreo y escalado, haciendo énfasis en la transparencia y facilidad de uso para el usuario.

CRANE se basa en la implementación de dos componentes:

- Un **Back-End** para la creación y despliegue de contenedores Docker.
- Un **Front-End** destinado al usuario final.

El componente de Back-End, utiliza Python y FastAPI para generar una API que permite la gestión de servicios Docker. Entre ellos la creación, despliegue y monitoreo de contenedores, al igual que la definición de políticas de escalado y la gestión de alertas.

El componente de Front-End, por medio de la API de CRANE, permite al usuario interactuar con la herramienta de forma transparente e independiente del lugar donde se ejecute. La interacción con el usuario se realiza mediante una interfaz web intuitiva y fácil de usar, diseñada para simplificar la experiencia del usuario y reducir la curva de aprendizaje. Este último componente queda por fuera del alcance de este trabajo y será abordado en trabajos futuros.

La infraestructura del backend de Crane se compone de varios componentes interconectados que trabajan conjuntamente para ofrecer una alta disponibilidad y una respuesta eficaz ante eventos críticos.

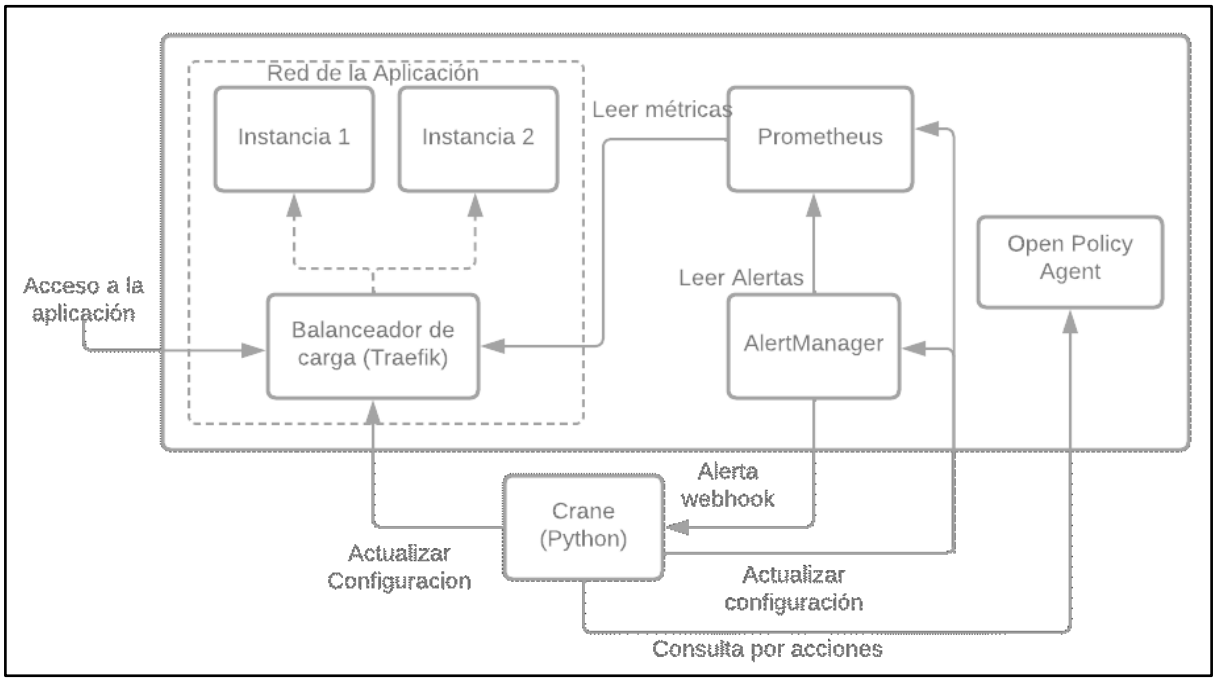


Figura 22. Diagrama de CRANE y sus componentes

- **CRANE**: Es el componente de esta arquitectura, escrito en Python, se encarga de coordinar y gestionar diversas funciones vitales. Actúa como el vínculo principal con la API de Python, mediante la librería Python on Whales para llevar a cabo operaciones de creación, lectura, actualización y eliminación (CRUD) de aplicaciones en contenedores. Además de ser el generador de las instancias que se visualizan en las figuras, CRANE también despliega su capacidad al recibir alertas y gestionarlas de forma eficiente mediante sus complementos mencionados más abajo. No se limita a estas funciones, sino que también valida y consulta los permisos de los usuarios, entre otras tareas cruciales delineadas en el diagrama.
- **Balanceador de Carga (Traefik)**: El acceso a la aplicación se regula mediante Traefik, que actúa como un balanceador de carga. Traefik distribuye de forma eficaz las solicitudes entrantes entre las instancias de la aplicación, asegurando un reparto equitativo de la carga y manteniendo la disponibilidad del servicio aun en los picos de demanda.
- **Red de la Aplicación:** Las instancias de la aplicación, denominadas Instancia 1 e Instancia 2, están interconectadas dentro de una red privada virtual. Esto permite una comunicación segura y aislada entre las instancias de la aplicación. A su vez, el router de la aplicación se comunica con Prometheus usando una red global para brindarle acceso a las métricas de red del contenedor.

- **Prometheus:** Actúa como el sistema de monitoreo, responsable de leer y recopilar métricas de las instancias de la aplicación. Estas métricas ofrecen información crucial sobre el estado y el rendimiento de la aplicación, esenciales para la toma automatizada de decisiones.
- **Alertmanager:** Este componente se integra con Prometheus para leer las alertas generadas basadas en las métricas recolectadas. Alertmanager procesa y gestiona estas alertas, y si se cumplen ciertas condiciones, desencadena una alerta por medio del webhook hacia CRANE.
    - **Alerta Webhook**: CRANE recibe alertas por webhooks, que son solicitudes HTTP desencadenadas por Alertmanager cuando se detectan condiciones específicas que requieren atención. Esto activa un proceso en CRANE para consultar posibles acciones y actualizar la configuración según sea necesario.
- **Open Policy Agent (OPA):** actúa como una capa de autorización y control de políticas, que recibe consultas de CRANE y determina las acciones permitidas de acuerdo con las políticas definidas. Este mecanismo asegura que cualquier cambio o respuesta automatizada esté alineada con las reglas de negocio y los requisitos de seguridad.

La interacción entre estos componentes permite tener una gestión de la aplicación mucho más automatizada y basada en políticas, desde la distribución de carga hasta la respuesta ante incidentes.

Uno de los principales desafíos, luego de diseñar esta arquitectura, fue encontrar un modelo de aplicación adecuado que permita representar el tipo de aplicación que se pretende crear usando la API de Crane. Este modelo debería permitirle al usuario definir qué servicios quiere crear, con qué configuraciones y además la cantidad de instancias que puede gestionar.

Después de muchas pruebas y modificaciones, se llegó al siguiente modelo de aplicación presentado en la tabla n°8:

Tabla 8: Modelo de aplicación en CRANE

CLASE	ATRIBUTO	TIPO DE DATO	DESCRIPCIÓN	OPCIONAL
App	**id**	int	Identificador único de la aplicación	Opcional
	name	str	Nombre de la aplicación	No
	services	list of Service	Lista de servicios asociados a la aplicación	No
	hosts	list	Lista de hosts asociados a la aplicación	No
	min_scale	int	Escala mínima de instancias de la aplicación	No. Default 1
	current_scale	int	Escala actual de instancias de la aplicación	No. Default 1
	max_scale	int	Escala máxima permitida de instancias de la aplicación	No. Default 2
	force_stop	bool	Indicador de detención forzada de la aplicación	No. Default False
	created_at	datetime	Fecha y hora de creación de la aplicación	Se auto-genera al crear el registro
	updated_at	datetime	Fecha y hora de la última actualización	Se auto-genera al modificar el registro
	deleted_at	datetime	Fecha y hora de eliminación de la aplicación	Se auto-genera al eliminar el registro
	user_id	int	Identificador del usuario propietario de la aplicación	No

Tabla 9: Modelo de servicio en CRANE

CLASE	ATRIBUTO	TIPO DE DATO	DESCRIPCIÓN	OPCIONAL
Service	**name**	str	Nombre del servicio	No
	image	str	Imagen de Docker utilizada por el servicio	No
	command	str	Comando para ejecutar cuando se inicia el servicio	Opcional
	ports	list	Puertos que utiliza el servicio	Opcional
	volumes	list	Volúmenes de datos para el servicio	Opcional
	networks	list	Redes a las que se conecta el servicio	Opcional
	labels	list	Etiquetas asociadas al servicio	Opcional

Con la arquitectura y el modelo ya definidos, lo siguiente que se debe entender es el paso a paso que se diseñó para poner en marcha la primera aplicación utilizando la API de Crane. Seguidamente se detallan las acciones que ejecuta el usuario y, por otro lado, lo que hace Crane Internamente:

Acciones ejecutadas por el usuario:

El usuario envía una solicitud HTTP conforme al modelo especificado en la tabla anterior. Como resultado de esta acción, se obtiene el ID de la aplicación junto con sus datos de acceso. Adicionalmente, se observa que los contenedores asociados a la aplicación se encuentran ya en ejecución.

Procesos internos ejecutados por CRANE:

1. **Autenticación y registro:** CRANE recibe la solicitud y extrae el user_id del token incluido, integrándolo al objeto de la aplicación.
2. **Almacenamiento:** La aplicación se registra en la base de datos.
3. **Gestión de nombres:** Se genera un nombre único para la aplicación, combinando el nombre suministrado por el usuario con un ID autoincremental, para prevenir colisiones de nombres en el host.
4. **Configuración de contenedores:** Se crea un archivo docker-compose.yml con la configuración predeterminada del proxy y las especificaciones brindadas por el usuario. Este archivo se almacena temporalmente.
5. **Despliegue de contenedores:** Utilizando la biblioteca Python on Whales, se procede a construir y levantar el entorno con Docker Compose.
6. **Integración con Prometheus:** Una vez en ejecución el contenedor, se obtiene el puerto asignado al router en la red de Prometheus y se actualiza el scrape para permitir la lectura de métricas de la nueva aplicación.
7. **Reinicio del sistema de monitoreo:** Se reinicia el stack de monitoreo para aplicar y refrescar los cambios de configuración.
8. **Gestión de archivos temporales:** Finalmente, si la constante REMOVE_TEMP_FILES está activada, se elimina el archivo temporal de Docker Compose creado.

El usuario tiene a disposición todas las configuraciones que acepta docker actualmente, no hay limitación para crear aplicaciones, siempre y cuando Python on Whales se mantenga actualizado con las últimas características de Docker. Además, una vez generado cualquier servicio, se hace uso del CRUD que ofrece la API de Crane para consultar, modificar y eliminar las aplicaciones generadas. A su vez, los usuarios con rol de administrador tienen a disposición las configuraciones de métricas, alertas y políticas para evolucionar la API y adaptarla a sus necesidades específicas.

Con el servicio ya corriendo en el sistema, se aplicaron ciertas pruebas de carga y rendimiento, prender y apagar los distintos contenedores para probar el funcionamiento del sistema de Alertas que tiene Crane. Posteriormente se explica como es el paso a paso desde que la API de Crane recibe una alerta, hasta que decide la acción a ejecutar.

Gestión de alertas por CRANE

En caso de recibir alertas por el webhook de Alert Manager, CRANE procede de la siguiente manera:

1. **Determinación de severidad:** Identifica si la alerta indica una condición **FIRING** o **RESOLVED**.
2. **Análisis de servicio afectado:** Localiza en la base de datos el id del servicio que ha generado la alerta.
3. **Consulta a Open Policy Agent (OPA):**
    - **Entrada**: CRANE envía al OPA el tipo de alerta y su severidad, sumando el nombre del servicio afectado.
    - **Proceso**: OPA evalúa las políticas correspondientes basadas en las entradas recibidas.
    - **Decisión**: OPA retorna la acción a ejecutar (escalar, desescalar, reiniciar contenedor, etc.).
4. **Ejecución de Acción:** CRANE ejecuta la acción determinada por OPA, que podría incluir ajustar la escala de servicios o reiniciar un contenedor.

5.2 | Preparación del entorno de trabajo

Para comenzar a desarrollar funcionalidades que extiendan y/o mejoren el comportamiento CRANE o simplemente poner en marcha la API para desplegar servicios, se necesitan, como primera medida, las siguientes herramientas: **Docker y Python**

Estas herramientas se encuentran disponibles para la mayoría de los sistemas operativos, pero en este trabajo se hizo hincapié en solo 3 de ellos y en los que se realizaron las pruebas de portabilidad de Crane:

- **Ubuntu**
- **Arch Linux**
- **Windows**
- **MacOS**

Antes de comenzar debemos clonar el repositorio de Crane del siguiente enlace: https://github.com/JoseMix/crane-rest-api

En el siguiente instructivo se explica cómo proceder en la instalación dependiendo el sistema utilizado:

UBUNTU:

```
$ sudo apt update
$ sudo apt upgrade
$ sudo apt-get install curl apt-transport-https ca-certificates
$ sudo apt-get install software-properties-common
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
$ sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu focal stable"
$ sudo apt update
$ apt-cache policy docker-ce
$ sudo apt install docker-ce
$ apt install python3-pip
$ apt install python
```

ARCH LINUX:

```
$ sudo pacman -S docker
$ sudo pacman -S python
```

Para los sistemas operativos basados en Linux se debe activar e iniciar el demonio de docker:

```
$ sudo systemctl enable docker.service
$ sudo systemctl start docker.service
```

WINDOWS y MAC:

1. Descargar e instalar el Docker Desktop desde la página oficial
    a. https://docs.docker.com/get-docker/
2. Descargar e instalar el cliente de Python desde la página oficial
    a. https://www.python.org/downloads/
3. Iniciar el Docker Desktop para encender el motor de docker

Una vez que se ha instalado Docker y Python en el sistema operativo se procede a abrir un terminal en el directorio de Crane y ejecutar el siguiente comando para instalar las dependencias del proyecto:

```
$ pip install -r requirements.txt
```

Nota: Se puede utilizar VENV el cual es un módulo de la biblioteca estándar de Python que se utiliza para crear entornos virtuales ligeros y específicos para proyectos particulares

Por último, se procede a iniciar el servidor de CRANE utilizando el comando

```
$ uvicorn app:app –reload
```

5.2.1 | Docker Daemon

Para que CRANE funcione adecuadamente, es imprescindible que el daemon de Docker esté activo y en funcionamiento. Este componente es esencial para la creación, ejecución y gestión de contenedores Docker, que son cruciales para el despliegue y la operación de las aplicaciones gestionadas por CRANE.

Verificación del estado del Docker Daemon en LINUX:

1. **Comprobar el estado del servicio**: Para asegurarse de que el daemon de Docker está activo, se puede ejecutar el siguiente comando en una terminal:

   ```
   $ sudo systemctl status docker
   ```

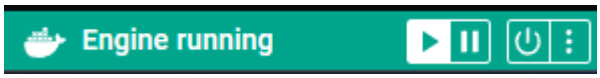
   Este comando da información sobre si el servicio está activo, inactivo o enfrentando errores.
2. **Inicio del servicio:** Si el servicio no está corriendo, se debe iniciar con el siguiente comando:

   ```
   $ sudo systemctl start docker
   ```

   Este comando activará el daemon de Docker, permitiendo que CRANE realice las operaciones necesarias sobre los contenedores.

Verificación del estado del Docker Daemon en WINDOWS:

1. Comprobar si la aplicación de escritorio Docker Desktop se encuentra abierta y se observa en la punta inferior izquierda el estado del docker daemon



Crane y su forma de evitar errores: Como el funcionamiento de Crane depende principalmente de que Docker esté funcionando en el sistema, se aplicaron ciertas validaciones durante el inicio para evitar errores posteriores.

En caso de que el daemon de Docker no esté corriendo y no pueda ser iniciado mediante los comandos habituales, se mostrará un mensaje de error. Este mensaje indicará la imposibilidad de conectar con el daemon.

La validación se realiza de la siguiente forma:

1. Utilizando la librería de Python Platform, mediante el comando platform.system() se obtiene el sistema operativo del equipo host.
2. Luego mediante otra librería de Python llamada **subprocess**, se procede a ejecutar un comando específico del sistema operativo que se obtuvo en el paso anterior y el cual permite verificar si docker daemon está corriendo.
3. En caso de que no se encuentre, lanza la siguiente alerta como muestra la Figura 23 y se detiene el inicio de la API.

```
$ uvicorn app:app --reload
INFO:     Will watch for changes in these directories: ['C:\\Users\\FFB\\crane-rest-api']
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [14500] using StatReload
ERROR:    Detectamos que Docker no está corriendo en este sistema
ERROR:    Por favor, inicie el servicio de Docker para que Crane funcione correctamente.
```

Figura 23. Mensaje de error en consola indicando que el demonio de Docker no está iniciado

En este escenario, es crucial:

- **Revisar la configuración del sistema:** Asegurarse de que no existen conflictos o problemas de configuración que impidan la ejecución del servicio.
- **Consultar los logs del sistema**: Los logs pueden ofrecer más detalles sobre cualquier error o problema que esté afectando al daemon.
- **Revisar la documentación técnica de Docker**: Problemas específicos requieren soluciones detalladas que están documentadas en las guías oficiales de Docker.

Validación de pasos previos:

Si persisten problemas con el daemon de Docker después de intentar las soluciones anteriores, es recomendable revisar los pasos descritos anteriormente para asegurarse de que no se haya omitido ninguna acción crucial para el correcto funcionamiento del sistema.

5.2.2 | Inicialización del Servicio de Reglas

Activación del Servidor de Reglas de OPA

La funcionalidad integral de CRANE comienza con la activación del servidor de reglas de Open Policy Agent (OPA). Durante el proceso de arranque de CRANE, se verifica la ejecución del cliente de OPA. Si no está activo, CRANE lo inicia utilizando las funciones CRUD integradas para manejar las políticas. Este procedimiento asegura que todas las operaciones ejecutadas dentro de CRANE sean validadas apropiadamente, desde la perspectiva de control de acceso y permisos.

Validación de Operaciones Basadas en Roles

Cuando CRANE recibe una petición, el servidor de OPA evalúa los roles asociados al usuario que realiza la solicitud. Basándose en estas evaluaciones, OPA permite o deniega la ejecución de la operación solicitada, jugando así un papel crucial en la seguridad y la gobernanza del sistema.

5.2.3 | Configuración de Políticas RBAC

Carga de Políticas en el Servidor de OPA

Uno de los pasos esenciales en la inicialización de CRANE es la carga de políticas RBAC (Role-Based Access Control) en el servidor de OPA. Dado que OPA almacena las políticas en memoria primaria, es crucial que estas se carguen cada vez que la aplicación se inicia. Este procedimiento se realiza mediante la función update_policies_file, que acepta tres parámetros:

- **OPA_RBAC_CONFIG_NAME**: El nombre del archivo de configuración de políticas.
- **OPA_RBAC_CONFIG_FILE**: La ruta al archivo que contiene las políticas.
- **FORCE**: Parámetro que indica al cliente de OPA si debe realizar alguna operación previa, como limpiar políticas antiguas antes de cargar las nuevas.

Formato y Uso del Archivo de Configuración

El archivo de configuración utiliza el formato .rego, un lenguaje de reglas declarativas específico para Open Policy Agent. Rego permite definir políticas de acceso y reglas de autorización de forma precisa y estructurada, facilitando la implementación de controles de acceso complejos en aplicaciones y servicios.

En este caso y para ilustrar el ejemplo, se usó el siguiente archivo de configuración para validar si el rol del usuario tiene permisos de acceso a un determinado recurso

```
package rbac.authz
import rego.v1

role_permissions := {
    "ADMIN": [
            {"action": "GET", "object": "APPS"},
            {"action": "POST", "object": "APPS"},
            {"action": "PATCH", "object": "APPS"},
            {"action": "DELETE", "object": "APPS"},
            {"action": "GET", "object": "ROLES"},
            {"action": "POST", "object": "ROLES"},
            {"action": "PATCH", "object": "ROLES"},
            {"action": "DELETE", "object": "ROLES"},
            {"action": "GET", "object": "OPA"},
            {"action": "POST", "object": "OPA"},
            {"action": "PATCH", "object": "OPA"},
            {"action": "DELETE", "object": "OPA"},
            {"action": "GET", "object": "MONITORING"},
            {"action": "POST", "object": "MONITORING"},
            {"action": "PATCH", "object": "MONITORING"},
            {"action": "DELETE", "object": "MONITORING"}
     ],
    "USER": [ {"action": "GET", "object": "APPS"} ],
}

default allow := false
allow if {
            roles := input.roles
            r := roles[_]
            permissions := role_permissions[r]
            p := permissions[_]
            p == {"action": input.action, "object": input.object}
}
```

Figura 24. Ejemplo de archivo REGO para la definición de políticas

Proceso de Verificación de Permisos en el Archivo de Configuración .rego

- **Obtención del Rol del Usuario:**
    - El proceso inicia con la identificación del rol del usuario que realiza la petición. Este dato es fundamental para determinar los permisos aplicables.
- **Iteración sobre el Listado de Roles:**

    - Se realiza una iteración sobre el listado de roles almacenados en el sistema para localizar el rol específico asociado al usuario que hace la petición.
- **Búsqueda de Permisos Asociados al Rol:**
    - Una vez identificado el rol, se procede a buscar en la base de datos los permisos vinculados a este rol. Cada permiso define las acciones permitidas y los objetos o recursos sobre los que se pueden ejecutar dichas acciones.
- **Iteración y Comparación de Permisos:**
    - Se itera sobre cada permiso asociado al rol para encontrar aquellos que correspondan con el recurso solicitado en la petición.
    - Durante esta iteración, se compara cada permiso con la acción y el objeto especificados en la solicitud de acceso.
- **Verificación de Coincidencias:**
    - Si se encuentra una coincidencia entre los permisos del rol y los requisitos de la solicitud de acceso, se retorna true.
    - Este resultado positivo indica que el acceso al recurso solicitado está permitido bajo las políticas de seguridad configuradas.
- **Conclusión de la Verificación:**
    - En caso de no encontrar coincidencias, se retorna false, restringiendo el acceso al recurso. Este resultado asegura que solo los usuarios autorizados según sus roles y permisos puedan realizar operaciones específicas dentro de CRANE.

5.2.4 | Inicialización del Servicio de Monitoreo

El proceso de inicialización del servicio de monitoreo en CRANE es vital para asegurar la supervisión de las aplicaciones desplegadas. Similar a la gestión de políticas, se utiliza un conjunto de operaciones CRUD para administrar la configuración y ejecución del servicio de monitoreo. Este proceso implica:

1. **Configuración del Servicio:** Al momento de inicializar los servicios, se obtiene la configuración base de la carpeta de monitoreo. La misma contempla las imágenes de Prometheus y Alert Manager, sumado a las reglas y configuraciones más específicas. El usuario es libre de modificar y adaptarlas a sus necesidades.

2. **Integración dinámica con CRANE**: Al momento de crear un nuevo servicio, Crane actualiza de forma automática la configuración de Prometheus permitiendo tomar métricas desde el inicio y haciendo que la integración sea mucho más dinámica.
3. **Automatización de Tareas**: Utilizando el CRUD diseñado, se asegura que el servicio de monitoreo se reinicie o actualice según las necesidades operativas sin intervención manual directa.
4. **Monitoreo Continuo**: Una vez inicializado, Prometheus comienza a recopilar datos de rendimiento y uso, que son esenciales para el mantenimiento proactivo y la optimización de las aplicaciones gestionadas por CRANE.

5.2.5 | Configuración de Reglas y Targets en Prometheus

La configuración de Prometheus en CRANE se maneja utilizando dos archivos esenciales que definen el comportamiento del monitoreo:

Archivo prometheus.yml:

- Parámetros Configurables: Este archivo configura aspectos importantes de Prometheus, como intervalos de raspado (scrape intervals), configuraciones de endpoints para recopilar métricas, y la definición de jobs y targets específicos.
- Ejemplo de Parámetros:
    - **scrape_interval**: Define la frecuencia con la que Prometheus recopila datos de los targets.
    - **evaluation_interval**: Establece con qué frecuencia se evalúan las reglas.
    - **targets**: Lista de servicios y endpoints desde donde Prometheus debe recolectar métricas.

Archivo rules.yaml:

- **Funcionalidad**: En este archivo se definen las reglas de alerta que Prometheus utilizará para notificar a los operadores sobre condiciones específicas que requieren atención, como uso elevado de CPU o fallos en los servicios.
- Ejemplo de Configuraciones:
    - Reglas basadas en métricas que exceden ciertos umbrales durante un período determinado.

- Notificaciones automáticas configuradas para alertar al equipo de operaciones cuando se detectan anomalías.

5.2.6 | Instanciación Dinámica mediante Docker Compose

Una vez que CRANE está operativo, permite a los usuarios solicitar la creación de aplicaciones de un modo dinámico. Como se mencionó en secciones anteriores, CRANE genera un archivo docker-compose.yml temporal que se ejecuta utilizando la biblioteca Python on Whales. Este proceso facilita la creación y configuración dinámica de contenedores según las necesidades específicas de cada aplicación.

Proceso de Generación de Archivo Docker Compose

1. **Inicialización**: Se obtiene la configuración del proxy específico asociado al nombre de la aplicación proporcionada.
2. **Obtención y Verificación de Proxy:** Se obtiene y verifica si la configuración del proxy requerida es válida y está disponible. En caso de no encontrarla, la función retorna una respuesta en formato JSON indicando un mensaje de error, lo cual permite gestionar adecuadamente los casos donde la configuración necesaria no está presente o es incorrecta.
3. **Construcción del Objeto YAML mediante PyYAML:** Se construye un objeto YAML utilizando la librería PyYaml que representa el archivo docker-compose.yml. Este objeto está compuesto por:
    - La versión de Docker Compose utilizada.
    - Las redes a utilizarse, entre ellas una red externa llamada Prometheus-net y una red interna llamada Crane-net.
    - Los servicios específicos que se ejecutarán, detallando cada contenedor Docker a desplegar.
4. **Configuración del Proxy:** Dentro del objeto YAML, se añade la configuración del proxy. Esto incluye especificaciones como la imagen Docker a utilizar, el comando a ejecutar, los puertos a mapear y los volúmenes a montar.
5. **Configuración de Servicios de la Aplicación:** Cada servicio de la aplicación se procesa individualmente. Se extrae el nombre del servicio y se manipulan las etiquetas de Docker

para agregar información relevante, como el nombre de la aplicación y las reglas de enrutamiento para Traefik.

- A continuación, se incorpora la configuración de cada servicio al objeto YAML.

6. **Escritura del Archivo YAML:** Finalmente, el objeto YAML completo se escribe en un archivo docker-compose.yml en el sistema de archivos. Este archivo está listo para ser ejecutado, instanciando así los servicios requeridos de forma dinámica.

```python
def docker_compose_generator(app: App):
  ''' Generate docker-compose.yml file '''

  app = copy.deepcopy(app)
  proxy = get_config(app.name)

  if not proxy:
    return JSONResponse(status_code=400,
                        content={"message": "Wrong proxy config detected", "proxy": proxy})

  yaml_obj = [
    {
      "version": '3',
      "networks": {
        "prometheus-net": {
          "external": True
        },
        "crane-net": {}
      },
      'services': {
        proxy['name']:
        {
          'image': proxy['image'],
          'command': proxy['command'],
          'ports': proxy['ports'],
          'volumes': proxy['volumes'],
          'networks': proxy['networks']
        }
      }
    }
  ]
  app_hosts = []
  services = app.services

  for service in services:
    name = service.pop('name', None)
    labels = service.get('labels', [])
    labels.extend( [f"a.label.name={app.name}", f"traefik.http.routers.{app.name}.rule=Host(`{app.name}-{name}.docker.localhost`)"])
    app_hosts.append(f"{app.name}-{name}.docker.localhost")
    service['labels'] = labels
    yaml_obj[0]['services'][name] = service

  path = Path.cwd() / TEMP_FILES_PATH / app.name / "docker-compose.yml"
  path.parent.mkdir(parents=True, exist_ok=True)
  write_yaml(yaml_obj, path)
```

Figura 25. Servicio encargado de generar los archivos YAML.

5.2.7 | Configuración de Rutas

Para facilitar la interacción entre el usuario y Crane, se ha desarrollado una API REST utilizando FastAPI, un moderno framework de Python que permite la creación rápida de interfaces de

programación de aplicaciones altamente eficientes y escalables. Esta API es quien permite la administración y operación de las funcionalidades del sistema a los usuarios.

5.2.7.1 | Estructura y versionado de rutas

Los endpoints de la API de CRANE están diseñados para ser intuitivos y fáciles de usar, siguiendo un esquema de versionado que permite la compatibilidad hacia atrás y facilita futuras expansiones:

- **URL Base y Versionado**: Todos los endpoints comparten una raíz común, lo cual simplifica la estructura de la API y facilita su memorización y acceso. La URL base para todos los endpoints es:

  ```
  http://127.0.0.1:8000/api/v1/
  ```

  Este esquema de URL tiene un versionado (v1) y facilita la implementación de nuevas versiones (v2, v3, etc.) sin interrumpir el funcionamiento de las versiones anteriores. Su principal utilidad es la de desarrollar nuevas funcionalidades o realizar mejoras sin afectar a los usuarios que dependen del contrato actual.
- **Segmentación por Bloques Funcionales**: Cada endpoint está organizado bajo un bloque específico que agrupa funcionalidades relacionadas, mejorando la organización y el descubrimiento de la API. Ejemplo:

  `/{bloque}/`

  Dónde **{bloque}** podría ser contenedores, aplicaciones, monitoreo, etc., dependiendo del área específica de CRANE que se esté gestionando.

5.3 | Proceso de autenticación en Crane

CRANE implementa un sistema de autenticación que permite a los usuarios registrarse y acceder a la plataforma de forma segura. Este proceso sirve para controlar que solo los usuarios autorizados puedan realizar operaciones contra el sistema.

Se detalla en los siguientes puntos cada paso del proceso de autenticación

5.3.1 | Registro de Usuarios

Para comenzar a usar CRANE, los usuarios deben registrarse. El registro se realiza mediante una solicitud POST al endpoint "**/auth/register"** de la API:

```
POST http://127.0.0.1:8000/api/v1/auth/register
Content-Type: application/json
{
  "full_name": "Franco Bellino",
  "email": "franco@gmail.com",
  "password": "123456"
}
```

Figura 26. Ejemplo de solicitud POST de registro

Este endpoint recibe tres piezas de información esenciales:

- **full_name**: El nombre completo del usuario.
- **email**: Un correo electrónico, que servirá como identificador único y medio de contacto con el usuario.
- **password**: Una contraseña segura, que será encriptada para su almacenamiento seguro en la base de datos de CRANE.

5.3.2 | Proceso de Login

Después de registrarse, el usuario puede acceder a CRANE enviando sus credenciales mediante un método POST al endpoint de Login:

```
POST http://127.0.0.1:8000/api/v1/auth/login
Content-Type: application/json

{
  "email": "franco@gmail.com",
  "password": "123456"
}
```

Figura 27. Ejemplo de solicitud POST de Login

Si las credenciales son correctas, el sistema responderá con un JSON Web Token (JWT) que el usuario deberá emplear para autenticar las solicitudes posteriores a la API. La respuesta incluirá:

```
{
  "access_token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoxLCJlbWFpbCI6ImZyYW5jb0BnbWFpbC5jb20iLC
Jyb2xlcyI6WyJBRE1JTiJdfQ.QkZ8W1uxFQ9CWIgo1YFCJaOC-2-2C7J4zrPSsZhVfBM",
  "token_type": "bearer",
  "expires_in": 15
}
```

Figura 28. Ejemplo de respuesta a la solicitud de Login

5.3.3 | Análisis del Token y Verificación de Permisos

El JWT contiene información crucial sobre el usuario, como su ID, email y roles.

Al decodificar el token generado por Crane se obtiene:

```
{
  "user_id": 1,
  "email": "franco@gmail.com",
  "roles": ["ADMIN"]
}
```

Figura 29. Estructura de un JWT decodificado

Estos datos son indispensables para que el sistema de Open Policy Agent (OPA) verifique si el usuario tiene los permisos necesarios para realizar operaciones específicas dentro de CRANE. Los roles determinan el nivel de acceso y las acciones que el usuario puede ejecutar, asegurando que la plataforma mantenga un alto nivel de seguridad y control de acceso. No es posible modificar

estos tokens para alterar los datos del usuario y roles porque la decodificación usando la firma los detectará como inválidos, verificando la veracidad de los mismos.

5.4 | Herramientas de Desarrollo y Pruebas

Durante el desarrollo de este trabajo, se utilizó Postman, una plataforma de colaboración para el desarrollo de APIs, para probar y validar todas las funcionalidades expuestas por la API de CRANE. Postman ofrece distintas herramientas que permiten simular solicitudes, examinar respuestas y evaluar la precisión y eficacia de la API.

5.4.1 | Creación del Contenedor Whoami y Test de Carga

Uso del Contenedor Whoami en CRANE

Para evaluar la funcionalidad y rendimiento de CRANE, se utilizó el contenedor whoami, muy reconocido en el ámbito de Docker por su simplicidad y utilidad para fines educativos, pruebas y demostraciones. Este contenedor ejecuta un servidor HTTP que responde a solicitudes GET en un puerto específico dando información relevante sobre el entorno del contenedor, como:

- **Identificador Único del Contenedor**: Clave para identificar cada instancia de manera individual.
- **Dirección IP del Contenedor**: Esencial para entender cómo se asignan las direcciones dentro de CRANE.
- **Información del Sistema Operativo**: Ayuda a verificar la compatibilidad y el rendimiento del contenedor en diferentes entornos.
- **Detalles del Entorno de Ejecución**: Ofrece una visión general sobre las condiciones bajo las cuales el contenedor está operando.

Objetivo de la Instanciación del Contenedor Whoami

El principal objetivo de desplegar este contenedor es evaluar el sistema de balanceo de carga de CRANE. Al exponer el contenedor whoami a un volumen elevado de peticiones y observar su escalado, se puede demostrar cómo el balanceador de carga, en este caso Traefik, distribuye de

un modo más eficaz el tráfico entre las diferentes instancias del contenedor. Este test es vital para validar la capacidad de CRANE de manejar de forma correcta el tráfico en escenarios de alta demanda, obteniendo una demostración práctica y realista del balanceo de carga que se propone.

Creación de la Primera Aplicación Utilizando el Contenedor Whoami

Para desplegar la primera aplicación en CRANE usando el contenedor whoami, es primordial realizar una solicitud POST a la API REST de CRANE. El siguiente ejemplo detalla cómo estructurar esta solicitud y qué implica cada parte del cuerpo del mensaje.

```
POST http://127.0.0.1:8000/api/v1/apps
Content-Type: application/json

{
 "name": "my_first_app",
 "services": [
  {
   "name": "whoami",
   "image": "traefik/whoami",
   "networks": ["crane-net"]
  }
 ]
}
```

Figura 30. Ejemplo de Solicitud POST para Crear una Aplicación

Desglose del Cuerpo de la Solicitud:

- **name**: Este campo define el nombre de la aplicación. En este caso, se ha denominado "my_first_app". Este identificador es requerido para gestionar la aplicación dentro del sistema CRANE.
- **services**: Esta sección es un arreglo que lista los servicios o contenedores que formarán parte de la aplicación.
    - **name**: El nombre del servicio, que en este ejemplo es "whoami".
    - **image**: La imagen Docker a utilizar. Para este servicio, se usa traefik/whoami, que es una imagen ligera y sencilla, ideal para pruebas y demostraciones.

- **networks**: Lista de redes a las cuales se conectará el contenedor. "crane-net" es una red interna que permite la comunicación entre los servicios gestionados por CRANE.

Configuraciones Adicionales del Servicio:

En la solicitud se pueden incluir otros parámetros que permiten una configuración más detallada de cada servicio, si bien se aceptan todas las propiedades disponibles para crear un contenedor, para evitar la complejidad en la creación de aplicaciones, se decidió acotarlo a los siguientes atributos, pero el usuario es libre de extender su funcionalidad si lo desea:

- **volumes**: Define los volúmenes de almacenamiento que serán montados en el contenedor. Es requerido para persistir datos o para compartir archivos entre el host y el contenedor.
- **commands**: Específica los comandos que se ejecutarán al iniciar el contenedor, permitiendo personalizar la configuración del servicio.
- **ports**: Lista de puertos que serán expuestos por el contenedor, facilitando el acceso externo al servicio.
- **labels**: Etiquetas que se pueden utilizar para organizar y gestionar contenedores dentro de Docker, necesarias para aplicar políticas o realizar filtrados específicos.

Importancia de la Red:

La especificación de la red a la que el contenedor debe conectarse es vital porque sin esta configuración, el contenedor podría encontrarse aislado y no sería capaz de comunicarse con otros servicios o con el host. Crane le asigna a cada servicio dos redes, una para comunicarse con su propio stack, en este ejemplo **traefik <--> whoami** y por otro lado una para conectarse con el stack de monitoreo.

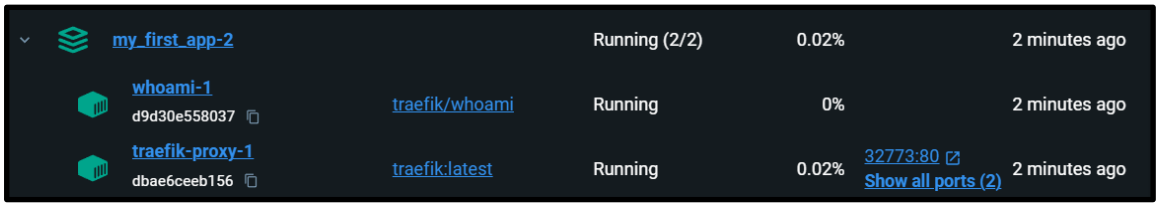


Figura 31. Pantalla de Docker Desktop con la instancia de Traefik y Whoami corriendo.

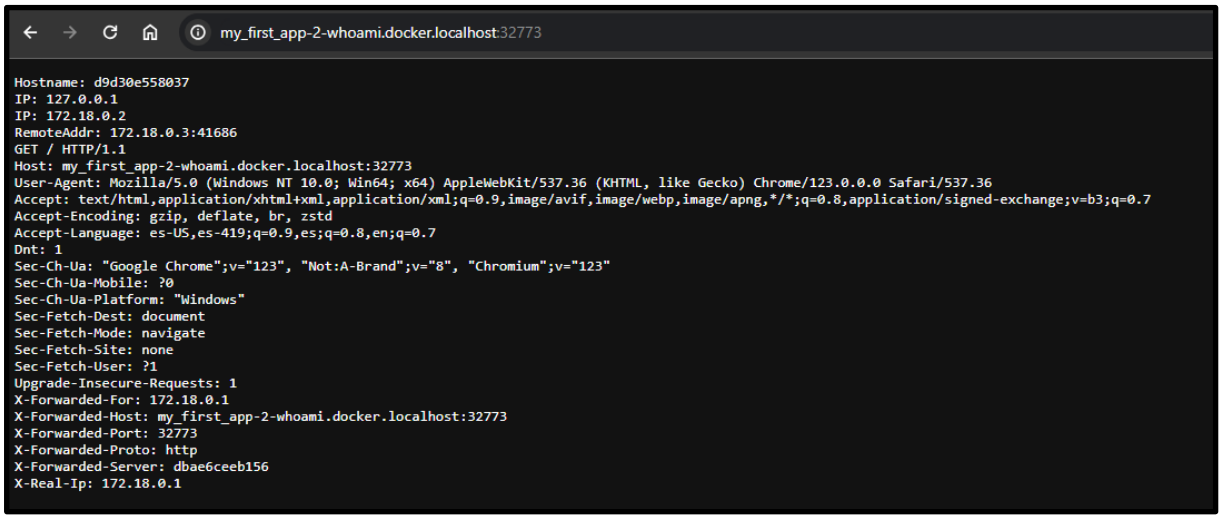


Figura 32. Respuesta de Whoami a una petición en su endpoint.

5.4.2 | Documentación y su generación automática en FastAPI

OpenAPI: Estableciendo el Estándar

OpenAPI es una especificación que define un estándar de la industria para describir APIs REST. El objetivo principal de OpenAPI es estandarizar la forma en que las interfaces de las APIs son representadas para que sean comprensibles tanto para humanos como para máquinas. La documentación OpenAPI detalla todos los endpoints, parámetros, respuestas esperadas y posibles errores en una API, acortando el tiempo necesario para entender y comenzar a usar una nueva API. Con una documentación completa, los desarrolladores tienen una guía clara sobre cómo utilizar la API de una forma más efectiva, evitando errores comunes y problemas de integración.

FastAPI: Automatización de la Documentación

FastAPI, un moderno framework web para construir APIs con Python, integra de forma nativa la especificación OpenAPI. Esto significa que genera automáticamente la documentación de la API mientras los desarrolladores escriben el código, utilizando las anotaciones de tipos de Python para definir los tipos de datos, las respuestas y las excepciones. Este fue otro de los factores por el cual se decidió usar este framework para acelerar la creación de la API de Crane.

FastAPI genera una documentación interactiva de la API (utilizando Swagger UI) que se actualiza en tiempo real conforme al desarrollo del código. Esto elimina la necesidad de mantener

manualmente la documentación y asegura que siempre esté sincronizada con la última versión de las rutas. Como se mencionó anteriormente, la documentación generada por este framework es servida usando Swagger UI, una herramienta que expone una interfaz gráfica para explorar y probar los endpoints de la API directamente desde el navegador permitiendo a los desarrolladores y a los usuarios finales comunicarse con la API sin escribir código extra.

Para acceder a la documentación en Crane o cualquier otra API generada con esta configuración se debe acceder a **http://127.0.0.1:8000/docs** y se encuentra una representación visual completa de todas las funcionalidades que ofrece la API, como sus rutas, parámetros y modelos de datos. La capacidad de ejecutar solicitudes directamente desde la interfaz de usuario permite probar y verificar la lógica de la API.

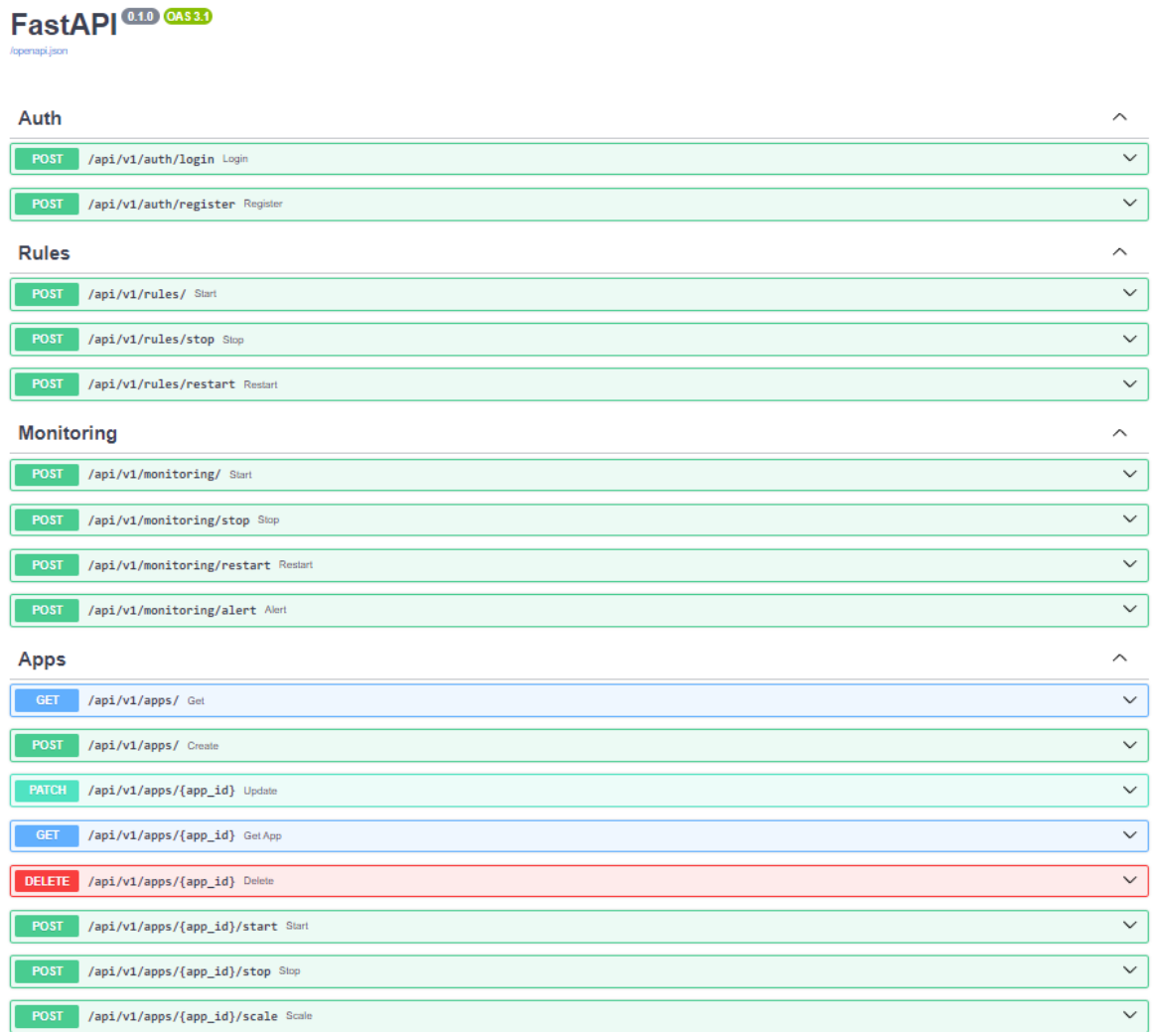


Figura 33. Ejemplo de documentación servida por Swagger.

CAPÍTULO 6 - Análisis de seguridad automatizado

6.1 | Herramientas para el análisis de seguridad

La seguridad en entornos de contenedores Docker es un punto muy importante en el mundo actual de la informática y es de interés tanto para los equipos de programación como para los de operaciones garantizar la integridad y la protección de los datos y las aplicaciones. En este contexto es que surgió la necesidad de usar herramientas que permitan identificar y mitigar las posibles vulnerabilidades en las imágenes de contenedor antes de su implementación en entornos de producción.

En este trabajo se tuvieron que analizar diferentes alternativas y se decidió por una en particular que resuelve bastante bien el objetivo de seguridad buscado. La herramienta en cuestión es Trivy, una solución de escaneo de vulnerabilidades que fue diseñada específicamente para entornos de contenedores y nos ofrece un análisis detallado de las imágenes, en busca de posibles riesgos de seguridad. Este análisis no sirve solamente para la detección de vulnerabilidades conocidas en las bibliotecas y dependencias utilizadas en la imagen, sino que también brinda información sobre las correcciones disponibles y las buenas prácticas de seguridad.

Trivy puede acoplarse en los flujos de desarrollo y despliegue, permitiendo identificar y solucionar proactivamente las vulnerabilidades antes de que se conviertan en problemas de seguridad en los entornos de producción.

Crane, al basar su núcleo en docker, necesita que las imágenes que se utilizan sean lo más seguras posibles por diferentes motivos:

- Identificación temprana de vulnerabilidades potenciales en imágenes de contenedores y así evitar riesgos de seguridad antes de desplegar en entornos de producción.
- Cumplimiento de estándares de seguridad y regulaciones.
- Concientizar a los alumnos y desarrolladores sobre buenas prácticas de seguridad.
- Protección de datos sensibles y críticos contra ataques y violaciones de seguridad.

6.2 | Instalación y uso de Trivy

Para instalar Trivy en Arch Linux se usa el siguiente comando:

```
sudo pacman -S trivy
```

Una vez finalizado, se comprueba que se instaló correctamente con el siguiente comando:

```
josemix@Arch ~ trivy -v
Version: 0.50.1
Vulnerability DB:
 Version: 2
 UpdatedAt: 2024-04-13 12:11:45.043008715 +0000 UTC
 NextUpdate: 2024-04-13 18:11:45.043008304 +0000 UTC
 DownloadedAt: 2024-04-13 13:17:08.956482077 +0000 UTC
josemix@Arch ~
```

Una vez completado, se verifica las imágenes instaladas con el siguiente comando:

```
josemix@Arch ~ docker images
REPOSITORY            TAG      IMAGE ID      CREATED        SIZE
openpolicyagent/opa   latest   4ad0aec50c90  2 weeks ago    76.5MB
prom/prometheus       latest   e350b167c4fa  2 weeks ago    262MB
traefik               latest   c1ef9171e9d1  4 weeks ago    159MB
prom/alertmanager     latest   11f11916f8cd  6 weeks ago    70.2MB
traefik/whoami        latest   9807740ea1ff  9 months ago   6.61MB
```

Para realizar el análisis de seguridad basta con ejecutar el comando **trivy image** y el nombre de la imagen que está en la columna REPOSITORY.

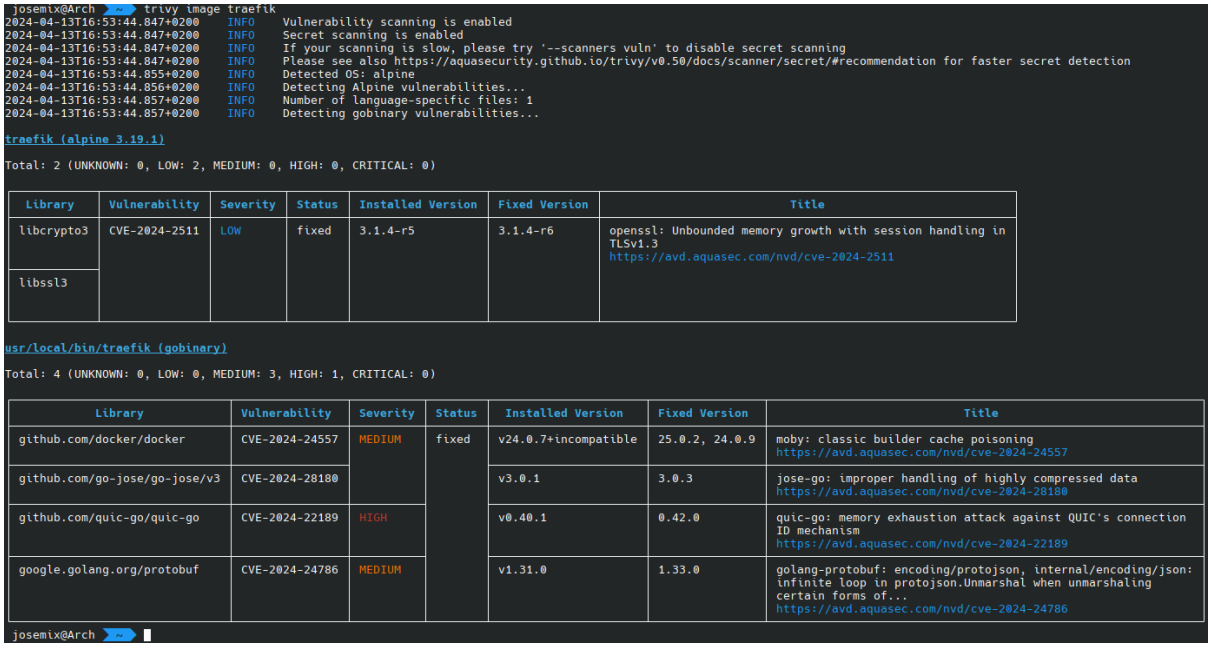


Figura 34. Salida en consola de los fallos de seguridad en los contenedores

La tabla con el resultado del análisis se compone de las siguientes columnas:

- **Library**: Nombre de la biblioteca o dependencia afectada por la vulnerabilidad.
- **Vulnerability**: ID de la vulnerabilidad (CVE) asociada con la biblioteca afectada.
- **Severity**: Gravedad de la vulnerabilidad, indicada por categorías como "LOW", "MEDIUM", "HIGH" o "CRITICAL".
- **Status**: Estado actual de la vulnerabilidad, que puede ser "fixed" (corregida) o "unfixed" (no corregida).
- **Installed Version**: Versión de la biblioteca o dependencia instalada en la imagen de contenedor.
- **Fixed Version**: Versión corregida de la biblioteca o dependencia afectada por la vulnerabilidad.
- **Title**: Título descriptivo de la vulnerabilidad, que da una breve explicación del problema de seguridad.

En el caso de ejemplo sobre la imagen de Traefik se informan 4 vulnerabilidades y que versión corrige las mismas:

- LOW en openssl
- MEDIUM en moby, protobuf, jose-go
- HIGH en quic-go

Se puede acceder al detalle de cada una de las vulnerabilidades usando el buscador: https://cve.mitre.org/cve/search_cve_list.html

CAPÍTULO 7 - Contribuciones

Durante el desarrollo de este trabajo se propuso presentar a Crane en los distintos congresos integrados por la Facultad de Informática de la Universidad Nacional de La Plata y toda la Red UNCI. Estos no solo fueron fundamentales para dar a conocer y revalidar esta idea sino también permitió enriquecer el conocimiento y compartir ideas con un montón de estudiantes, docentes y profesionales del ámbito con mucha experiencia.

Ambos artículos fueron aceptados para su exposición en los siguientes congresos:

- **WICC (13 y 14 abril 2023 - Junín - Bs. As.)**: Despliegue de aplicaciones contenerizadas: un caso de implementación basado en Crane. Link
- **CACIC (10 octubre 2023 - Luján - Bs. As.):** CRANE: Simplificando el Despliegue de Aplicaciones Contenerizadas en Entornos Locales. Link

CAPÍTULO 8 - Conclusiones y trabajos futuros

Crane comenzó como una idea teórica, limitada a pequeñas pruebas con bash scripts y presentada en el documento “Crane: A Local Deployment Tool” [5]. El objetivo principal de este trabajo fue llevar esa idea a la práctica. Para esto, uno de los primeros desafíos fue analizar la viabilidad técnica de la teoría de Crane, esto implicó desglosar el paper, extraer sus componentes de arquitectura y entender cómo deberían interactuar entre sí para alcanzar el objetivo final.

Realizando un estudio individual de cada pieza para verificar si existían alternativas más adecuadas o eficientes a las planteadas inicialmente, se desarrolló el primer prototipo en Python siguiendo la estructura de Crane. Esto permitió comprobar y validar la comunicación entre sus diferentes partes. En este punto del proceso, fue indispensable aprender y extender el conocimiento sobre Python, sus complementos como FastAPI, y su interacción con las librerías para comunicarnos con Docker.

Superada esta etapa inicial y con una idea más formada y validada, se automatizó el despliegue utilizando Docker Compose y la librería Python on Whales, facilitando la instanciación y escalabilidad de los servicios. Con estas funciones listas, mediante solicitudes HTTP se pudo realizar todo el CRUD de servicios desde una interfaz más intuitiva y dio más sentido a llevar Crane a la práctica.

La siguiente etapa consistió en desplegar y probar el primer servicio, utilizando una imagen "whoami", e integrando progresivamente los componentes necesarios: Traefik para la gestión del tráfico, Prometheus para la monitorización de métricas, Alert Manager para la gestión de alertas, y Open Policy Agent para la aplicación de políticas de escalado y seguridad.

Se completaron pruebas de carga en los contenedores instanciados, confirmando que el escalado, desescalado y reinicio funcionaban correctamente. Esto permitió alcanzar el objetivo principal de brindar un servicio REST para crear y desplegar servicios Docker con sus respectivas herramientas de medición, políticas de escalado y gestión de alertas.

Con el objetivo principal cumplido, solamente faltaba integrar los modelos de datos correctamente y aplicar la capa de autenticación y autorización sobre la API diseñada. En este punto ya se tiene una API REST con sus funciones principales completas y con los mecanismos de

seguridad adecuados para que el usuario pueda hacer uso de los mismos y comenzar el proceso de aprendizaje.

Más allá de su utilidad técnica, esta solución pretende tener un valor significativo en el ámbito educativo. Alumnos y docentes podrían utilizarla para aprender sobre prácticas de desarrollo y despliegue de servicios, pero también para la simulación de prácticas de DevOps en entornos locales, un proceso que, como se explica a lo largo de este documento, es muy complejo y costoso.

Para los trabajos futuros, se plantea la creación de un Frontend utilizando React. Este Frontend consumirá todos los endpoints REST creados, desde la creación y despliegue de servicios Docker hasta la medición, definición de políticas de escalado y gestión de alertas. Esto proporcionará a los usuarios una interfaz completa para comunicarse con todas las funciones principales de Crane.

El propósito de este Frontend es hacer que Crane sea accesible y eficiente para un amplio abanico de usuarios, desde principiantes hasta expertos. No solo busca hacer más simple el proceso técnico, sino también fomentar el entendimiento y aprendizaje sobre la gestión de servicios en contenedores.

El sistema de alertas también puede mejorar utilizando métricas del contenedor, como el uso de memoria y CPU, para sus políticas de escalado y otros parámetros. Actualmente, las métricas se toman directamente del tráfico del enrutador.

Todo este camino recorrido hizo posible, no solo aprender sobre muchos conceptos de desarrollo y DevOps, sino también ampliar el stack técnico con herramientas que antes eran desconocidas. Tener aplicaciones que sigan buenas prácticas y cuenten con la resiliencia necesaria para solventar cualquier problema es esencial para cualquier persona que se desempeñe en el mundo del software.

En el futuro, se continuará explorando y perfeccionando esta herramienta, con la meta de posicionar a Crane como una solución flexible y robusta. Que permita a los estudiantes desenvolverse en el mundo del desarrollo y la gestión de servicios en contenedores, y garantizara que Crane siga creciendo para adaptarse a nuevas necesidades y tendencias que surjan en el campo del software.

Referencias

[1] Rani, D., & Ranjan, R. K. (2014). A comparative study of SaaS, PaaS and IaaS in cloud computing. International Journal of Advanced Research in Computer Science and Software Engineering, 4(6).

[2] Bullington-McGuire, R. and Dennis, A.K. and Schwartz, M. (2020). Docker for Developers: Develop and run your application with Docker containers using DevOps tools for continuous delivery. Packt Publishing.

[3] Kubernetes - https://kubernetes.io/es/docs/concepts/overview/what-is-kubernetes/

[4] Ebert, C., Gallardo, G., Hernantes, J., & Serrano, N. (2016). DevOps. *Ieee Software*, *33*(3), 94-100.

[5] Arcidiacono, J., Bazán, P., del Río, N., & Lliteras, A. B. (2022). Crane: A Local Deployment Tool for Containerized Applications. In Conference on Cloud Computing, Big Data & Emerging Topics (pp. 58-71). Springer, Cham.

[6] Python - https://www.python.org/doc/

[7]Adekotujo, A., Odumabo, A., Adedokun, A., & Aiyeniko, O. (2020). A comparative study of operating systems: Case of windows, unix, linux, mac, android and ios. *International Journal of Computer Applications*, *176*(39), 16-23.

[8]Costo de servicio Amazon EC2 - https://aws.amazon.com/ec2/pricing/on-demand

[9]Costo de servicio Azure virtual Machine https://azure.microsoft.com/es-es/pricing/calculator

[10] Costo de servicio Google Compute Engine https://cloud.google.com/compute/all-pricing

[11] Burns, B., Beda, J., Hightower, K., & Evenson, L. (2022). Kubernetes: up and running. " O'Reilly Media, Inc.".

[12] Docker - https://docs.docker.com/

[13] Docker Swarm - https://docs.docker.com/engine/swarm/

[14] Docker Classic Swarm - https://github.com/docker-archive/classicswarm

[15] Ibrahim, M. H., Sayagh, M., & Hassan, A. E. (2021). A study of how Docker Compose is used to compose multi-component systems. *Empirical Software Engineering*, *26*, 1-27.

[16] Ben-Kiki, O., Evans, C., & Ingerson, B. (2009). Yaml ain't markup language (yaml™) version 1.1. *Working Draft 2008*, *5*(11).

[17] Sommerlad, P. (2003, June). Reverse Proxy Patterns. In *EuroPLoP* (pp. 431-458).

[18] Traefik - https://doc.traefik.io/traefik/

[19] Johnson, R. E. (1997). Frameworks=(components+ patterns). *Communications of the ACM*, *40*(10), 39-42.

[20] FastAPI - https://fastapi.tiangolo.com/

[21] OpenAPI - https://github.com/OAI/OpenAPI-Specification

[22] JSON Schema - https://json-schema.org/

[23] Lathkar, M. (2023). *High-Performance Web Apps with FastAPI: The Asynchronous Web Framework Based on Modern Python*. Apress.

[24] https://github.com/gabrieldemarmiesse/python-on-whales

[25] YAML - https://yaml.org/

[26]PyYAML Documentation - https://pyyaml.org/wiki/PyYAMLDocumentation

[27] Jones, M., Bradley, J., & Sakimura, N. (2015). *Json web token (jwt)* (No. rfc7519).

[28] JWT Documentation - https://jwt.io/

[29] IANA JWT - https://www.iana.org/assignments/jwt/jwt.xhtml

[30] Prometheus Documentation - https://prometheus.io/docs/introduction/overview/

[31] Alert Manager Documentation - https://prometheus.io/docs/alerting/latest/alertmanager/

[32] Knight, J. C. (2002, May). Safety critical systems: challenges and directions. In *Proceedings of the 24th international conference on software engineering* (pp. 547-550).

[33] Ferraiolo, D., Cugini, J., & Kuhn, D. R. (1995, December). Role-based access control (RBAC): Features and motivations. In *Proceedings of 11th annual computer security application conference* (pp. 241-48).

[34] SQLite Documentation - https://www.sqlite.org/docs.html

[35] Paredaens, J., De Bra, P., Gyssens, M., & Van Gucht, D. (2012). *The structure of the relational database model* (Vol. 17). Springer Science & Business Media.

[36] Härder, T. (2005). DBMS Architecture–the Layer Model and its Evolution. *Datenbank-Spektrum*, *13*, 45-57.

[37] SQLAlchemy Documentation - https://docs.sqlalchemy.org/en/20/

[38] Ireland, C., Bowers, D., Newton, M., & Waugh, K. (2009). Understanding object-relational mapping: A framework based approach. *International Journal On Advances in Software*, *2*(2).

[40] Bhardwaj, S., Jain, L., & Jain, S. (2010). Cloud computing: A study of infrastructure as a service (IAAS). *International Journal of engineering and information Technology*, *2*(1), 60-63.