UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
CONCEPCIÓN- CHILE

MIGRACIÓN DESDE UNA INFRAESTRUCTURA NO ESCALABLE HACIA KUBERNETES MEDIANTE PRÁCTICAS DE LA CULTURA DEVOPS

THOMAS JOAQUÍN ANDRÉS TAPIA LEÓN

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO EN INFORMÁTICA

Profesor Guía: Javier Maldonado Carmona

Agosto – 2025

CONSTANCIA DE VALIDACIÓN Y CONFIDENCIALIDAD DE MONOGRAFÍA A REPOSITORIO ACADÉMICO

1.- IDENTIFICACIÓN DEL TRABAJO ACADÉMICO

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

Título del trabajo: MIGRACIÓN DESDE UNA INFRAESTRUCTURA NO ESCALABLE HACIA KUBERNETES MEDIANTE PRÁCTICAS DE LA CULTURA DEVOPS

Nombre del candidato(a): Thomas Tapia León

Carrera / Grado: Ingeniería en informática

Campus: Concepcion ; **Departamento**: Electrónica e informática

2.- VALIDACIÓN DEL PROFESOR GUÍA/DIRECTOR DE TESIS

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

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

3.- EVALUACIÓN DE CONFIDENCIALIDAD POR PROPIEDAD INDUSTRIAL

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

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

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

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

4.- FIRMAS

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

Fecha: 13/08/2025 ; **Firma:** _____

Estudiante o Candidato(a):

Fecha: 12/08/2025 ; **Firma:** _____

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

DEDICATORIA

Para mis padres Nora y Jaime, que siempre se esforzaron para que nunca me falte nada ya que gracias a ellos soy la persona que soy ahora, para Florencia, mi pareja quién me acompañó durante todo este tiempo apoyándome y ayudándome siempre en los momentos más difíciles

AGRADECIMIENTOS

A Enerlink, por confiar en mí y permitirme liderar y llevar a cabo un proyecto tan importante para el negocio.

RESUMEN

Resumen — Uno de los mayores desafíos que enfrentan las empresas emergentes TI surge cuando el negocio comienza a escalar y el software junto a la infraestructura dejan de ser capaces de satisfacer la creciente demanda de recursos, y al mismo tiempo, los costos operativos tienden a aumentar rápidamente. Se explorará la migración de un sistema mono cliente(*single-tenant*)desde una infraestructura basada instancias virtuales alojadas en la nube que ha dejado de escalar hacia un clúster de Kubernetes.

Se abordará el proceso de planificación y ejecución para lograr una migración de infraestructura sin afectar la operación del sistema, y se analizan los beneficios obtenidos tras la transición, tales como mejoras en la escalabilidad, eficiencia operativa y reducción de costos.

Palabras Clave— *single tenancy*, infraestructura cloud, DevOps, kubernetes, contenedores, docker, internet de las cosas, microservicios.

ABSTRACT

Abstract — One of the biggest challenges faced by tech startups arises when the business begins to scale and both the software and infrastructure are no longer able to meet the growing demand for resources. At the same time, operational costs tend to rise rapidly. This thesis will explore the migration of a single-tenant system from a cloud infrastructure that can no longer scale to a Kubernetes cluster.

The planning and execution process will be addressed in order to achieve an infrastructure migration without affecting system operations, and the benefits obtained after the transition will be analyzed, such as improvements in scalability, operational efficiency, and cost reduction.

Keywords— single tenancy, cloud infrastructure, DevOps, kubernetes, containers, docker, internet of things, microservices.

GLOSARIO

APM (Application Performance Management): Conjunto de herramientas y prácticas para monitorear y gestionar el desempeño de las aplicaciones, recopilando métricas como tiempos de respuesta y uso de recursos con el fin de detectar y diagnosticar problemas de rendimiento en tiempo real.

Auto escalamiento: Mecanismo que permite a un sistema ajustar automáticamente sus recursos en respuesta a la carga de trabajo, garantizando que la infraestructura pueda crecer o decrecer según la demanda sin intervención manual.

AWS (Amazon Web Services): Plataforma de computación en la nube de Amazon que ofrece una amplia gama de servicios bajo un modelo de pago por uso, permitiendo desplegar infraestructura y aplicaciones de forma escalable y bajo demanda.

Clúster: Conjunto de nodos que operan de forma coordinada como una única plataforma para ejecutar cargas de trabajo distribuidas, proporcionando alta disponibilidad, tolerancia a fallos y capacidad de escalamiento horizontal.

Containerización: Práctica de encapsular una aplicación y todas sus dependencias en una unidad estándar llamada contenedor, de modo que pueda ejecutarse de forma aislada y consistente en cualquier entorno.

Contenedor: Unidad de software ligera que empaqueta una aplicación junto a sus dependencias permitiendo ejecutarla de manera aislada y consistente en distintos entornos.

Despliegue: Proceso de instalar o liberar una nueva versión de una aplicación o servicio en un entorno (por ejemplo, en producción), incluyendo la configuración necesaria para que esté disponible a los usuarios finales.

Despliegue continuo (CD): Práctica de DevOps en la que las nuevas versiones del software se ponen en producción de forma automática y frecuente una vez que han pasado las pruebas.

DevOps: Conjunto de prácticas, metodologías y cultura organizacional que integra los equipos de desarrollo de software y de operaciones de TI para agilizar la entrega de software de alta calidad. Se enfoca en la automatización de procesos, la colaboración constante entre equipos y la retroalimentación continua durante todo el ciclo de vida del desarrollo.

Docker: Plataforma de virtualización a nivel de sistema operativo que permite crear, distribuir y ejecutar contenedores.

Docker Compose: Herramienta de Docker que permite definir y ejecutar aplicaciones de múltiples contenedores.

Dockerfile: Archivo de texto que contiene las instrucciones paso a paso para construir una imagen de Docker.

EC2 (Amazon Elastic Compute Cloud): Servicio de AWS que ofrece capacidad de cómputo virtual en la nube mediante la creación de instancias escalables bajo demanda.

EKS (Elastic Kubernetes Service): Servicio administrado de AWS para Kubernetes. Permite crear y operar fácilmente clústeres de Kubernetes en la nube de AWS.

ElastiCache: Servicio administrado de AWS para despliegue de caché en memoria basado en Redis o Memcached.

ELB (Elastic Load Balancing): Servicio de AWS que distribuye automáticamente el tráfico de red entrante entre múltiples instancias de aplicación o contenedores.

Escalabilidad: Capacidad de un sistema para adaptarse al crecimiento o reducción de la carga de trabajo de forma eficiente.

Flux CD: Herramienta de despliegue continuo y GitOps que sincroniza automáticamente el estado de un clúster de Kubernetes con la configuración declarativa almacenada en un repositorio Git.

GitOps: Enfoque de gestión de infraestructuras y aplicaciones en el que el repositorio de código (*Git*) actúa como fuente de verdad de la configuración del sistema.

Grafana: Plataforma de visualización y monitoreo de origen abierto que permite agrupar datos de múltiples fuentes y representarlos en paneles visuales.

Infraestructura: Conjunto de recursos de hardware, software, redes y servicios base que sostienen la operación de un sistema informático, contenedores, sistemas de integración y despliegue continuo (CI/CD) y otros elementos que garantizan el funcionamiento y mantenimiento del ecosistema tecnológico.

Infraestructura como código (IaaC): Práctica que consiste en definir y gestionar la configuración de la infraestructura mediante archivos de código o *scripts*.

Integración continua (CI): Práctica de desarrollo de software en la que los desarrolladores incorporan con frecuencia sus cambios de código en una rama principal o común, desencadenando automáticamente un proceso de compilación y pruebas.

Kubernetes: Plataforma de código abierto para la orquestación de contenedores que automatiza la implementación, escalado y gestión de aplicaciones en contenedores.

Monitoreo: Proceso continuo de recolección, análisis y visualización de datos sobre el funcionamiento de un sistema, con el fin de supervisar su salud y desempeño.

Multi-tenant: Modelo de arquitectura de software en el cual una misma instancia de la aplicación y de la infraestructura asociada es utilizada de manera compartida por múltiples clientes u organizaciones (*tenants*), manteniendo aislados sus datos y configuraciones.

NAT (Network Address Translation): Mecanismo de red que traduce direcciones IP y puertos de origen o destino al pasar los paquetes a través de un dispositivo, típicamente un *router* o *firewall*.

Observabilidad: Capacidad de un sistema para exponer y generar datos internos que faciliten comprender su estado y comportamiento desde el exterior.

Pipeline: Conjunto de etapas automatizadas organizadas secuencialmente para llevar a cabo la integración y despliegue de aplicaciones.

RDS (Amazon Relational Database Service): Servicio administrado de AWS para bases de datos relacionales. RDS permite desplegar y operar motores de bases de datos populares cómo PostgreSQL, MySQL, etc.

recuperarse rápidamente de ellos, manteniendo la continuidad del servicio. Un sistema resiliente puede tolerar la caída de uno o varios componentes (por ejemplo, servidores, procesos o microservicios) sin interrumpir significativamente sus funciones, gracias a redundancias y mecanismos de recuperación automática. Asimismo, tras una interrupción parcial, el sistema resiliente vuelve a su estado normal de operación (o a uno degradado controlado) en el menor tiempo posible y con mínima intervención humana.

Single-tenant: Modelo de implementación de software en el cual cada cliente u organización (*tenant*) dispone de su propia instancia dedicada de la aplicación y de los recursos de infraestructura asociados

VPC (Virtual Private Cloud): Servicio de AWS que permite crear una red virtual privada y aislada dentro de la nube de AWS.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

INDICE DE TABLAS

INDICE DE ANEXOS

INTRODUCCIÓN

Las empresas de tecnologías emergentes (startups [1]) en los últimos años han comenzado a optar por infraestructuras cloud, ya que permiten optimizar costos (*pay-as-you-go*[1]), cumplen con altos estándares de seguridad, aseguran una alta disponibilidad, etc. Inicialmente necesariamente consideran la escalabilidad futura del negocio. La infraestructura es todo el software y hardware que da soporte a las aplicaciones, esto incluye los centros de datos, los sistemas operativos, los flujos de integración y despliegue continuo (*pipelines)*, la gestión de la configuración y cualquier sistema o software necesario para soportar el ciclo de vida de las aplicaciones [2], en el contexto de las infraestructuras *cloud*, todos estos recursos son proporcionados por proveedores cloud como Amazon Web Services, Azure, etc.

Cuando un negocio escala rápidamente se puede ver enfrentado a diversos problemas relacionado a los costos, por ejemplo, en empresas TI emergentes, es común que se experimente un aumento de los gastos de la infraestructura. También pueden surgir problemas operacionales en el software lo que genera que se afecte la relación con los clientes y dificultades en los equipos de desarrollo, que deben enfocarse en atender estos problemas por sobre el desarrollo de nuevos requerimientos funcionales (*features*) que agregan valor al producto.

En este documento analizaremos las diversas problemáticas que ha enfrentado un sistema de medición eléctrica inteligente y abordaremos el por qué la infraestructura actual y algunas malas prácticas han sido las responsables de éstas. Exploraremos los diversos componentes del software y proponiendo una migración de la infraestructura actual hacia un clúster de Kubernetes, que busca optimizar costos y recursos computacionales, permitiendo una alta escalabilidad, disponibilidad, resiliencia y observabilidad de todo el sistema, avanzando hacia la aplicación de la cultura DevOps en el ciclo de vida del software.

En el capítulo 1 se describirá el estado actual del sistema y los problemas que involucra, para luego en el capítulo 2 dar mayor contexto respecto al funcionamiento actual del sistema e introducir las nuevas tecnologías que se incorporarán. El capítulo 3 consiste en los pasos que se siguieron para llevar a cabo la migración. El capítulo 4 evidenciará los resultados de la migración, exponiendo los beneficios que trajo consigo, finalizando con la conclusión en el capítulo 5, donde se analiza el proceso con un enfoque analítico.

[1] Pay as you go es un modelo de monetización en el que se paga solo por los recursos que se consumen.

CAPITULO 1: DEFINICIÓN DEL PROBLEMA

1.1. Contexto del problema

SMS o SMI[2] es un software para la medición eléctrica inteligente, consiste en un sistema que se encarga de obtener lecturas eléctricas desde medidores inteligentes ubicados en los activos inmobiliarios de nuestros clientes, para luego procesar y enviar las notas de cobro[3] a los arrendatarios de estos, según las tarifas y la legislación vigente. El rápido escalamiento de clientes ha dejado en evidencia falencias críticas en la operación del sistema, como la constante indisponibilidad del sistema y lentitud extrema en el proceso de subida de lecturas provocando pérdidas millonarias a los clientes al no estar disponibles al momento de tarificar. Estos problemas traen consigo que los desarrolladores utilicen su tiempo en depurar los errores sin tener las herramientas adecuadas, ralentizando desarrollos importantes que generan valor al producto. Este sistema con el tiempo se ha transformado en poco confiable, generando incertidumbre en flujos críticos de la operación.

1.1.1. Smart Metering Software

SMM se compone de 4 aplicaciones: *smi-integrador*, *smi-backend*, *smi-tar* y *smi-sync*. En conjunto son el sistema encargado de obtener lecturas desde medidores inteligentes y hacen el cálculo del dinero que se le debe cobrar a los arrendatarios por concepto de consumo eléctrico.

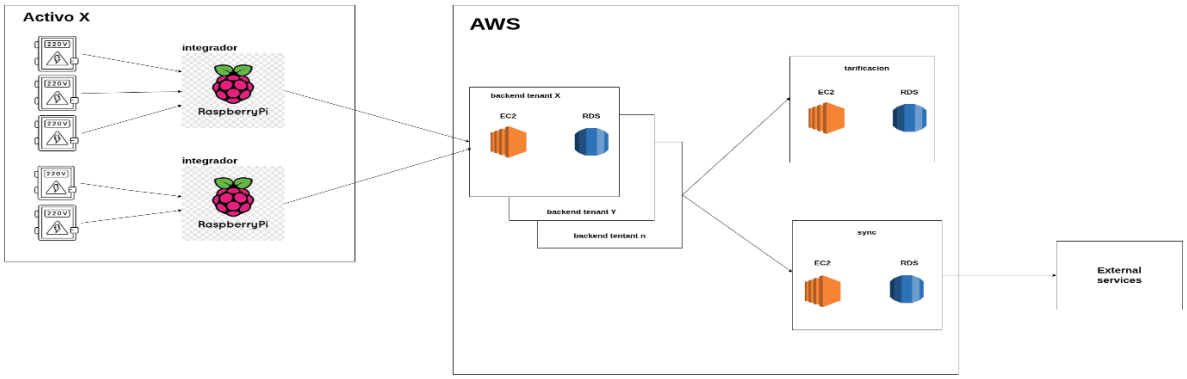

Figura 1: Diagrama básico para representar la infraestructura actual del sistema.

[2] SMI o SMS se refieren al mismo sistema, SMI corresponde a las siglas en español y SMS en inglés. El significado está en el glosario.
[3] Una nota de cobro es la boleta que se le hace llegar a los arrendatarios de los clientes.

1.1.2. Infraestructura y arquitectura actual

smi-backend es una aplicación *single-tenant*, esto quiere decir que no existen servicios transversales entre las distintas instancias y clientes, en este caso existe una instancia de EC2, RDS y ElasticCache para cada cliente. En el caso de *smi-tar* y *smi-sync* son aplicaciones *multi-tenant*, disponibles para todas las instancias de *smi-backend*.

El sistema de *smi-tar* recibe la información necesaria de todos los medidores para realizar los cálculos correspondientes a un periodo especifico de tiempo con el fin de calcular el monto a pagar por consumo mensual.

smi-sync consiste en diversas integraciones para clientes que esperar recibir los cobros en sus sistemas internos, haciéndoselos llegar en los formatos esperados.

El integrador consiste en múltiples Raspberrys PI 4 a las que llamaremos OPI, que están ubicadas físicamente en los activos (propiedades inmobiliarias de los clientes) y se conectan directamente a los medidores inteligentes. El número de OPI's instaladas depende de las limitaciones técnicas y físicas del lugar, como por ejemplo distancias entre los medidores, número de pisos, etc.

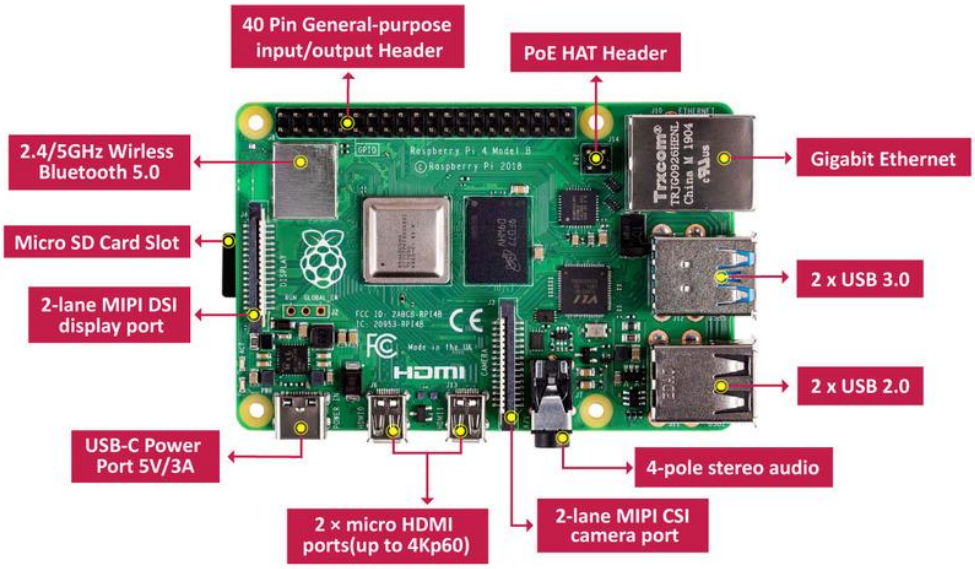

Figura 2: Diagrama de hardware de Raspberry Pi 4. Fuente: www.robot-advance.com

1.2. Dificultades actuales

Uno de los principales problemas es la nula observabilidad y monitoreo de los sistemas al no tener implementadas herramientas APM (Application Performance Management), ni tener habilitados los servicios que ofrece AWS, por lo que la única forma que existe de depuraciones (*debugging*) es acceder directo a los archivos *.log* presentes en las instancias EC2.

Otro gran problema es que el proceso de despliegue *(deploy)*, donde tanto *smi-backend* como *smi-tar* dependen de 24 *bash scripts*, de los cuales el equipo asegura no entender lo que realmente ocurre cuando se realiza este flujo. Esto trae como consecuencias que el despliegue continuo (CD) siempre dependa de la supervisión humana, lo cual idealmente no debería ser así.

Gran parte de las dependencias del proyecto se encuentran deprecadas hace más de cinco años en algunos casos, causando graves problemas de ciberseguridad. Otro punto importante para considerar es que no se han dedicado esfuerzos en construir una infraestructura del todo segura, permitiendo accesos a servicios críticos sin ninguna restricción.

Finalmente, las configuraciones de estos sistemas están diseñados para no ser migrados, es decir, cualquier intento de montar esta aplicación de una manera distinta provocará múltiples errores dado la incompatibilidad del código fuente, por lo que si se quiere realizar una migración de infraestructura también deberían realizarse cambios a nivel de código

1.3. Objetivo y alcance de la memoria

1.3.1 Objetivo General

Levar cabo la migración de la infraestructura del sistema hacia un clúster de Kubernetes para permitir un mejor escalamiento, poder optimizar costos y mejorar el ciclo de vida del software.

1.3.2 Objetivos específicos

Para lograr el objetivo principal, este se dividió en objetivos específicos, los cuales son los siguientes:

- Analizar el estado actual del sistema y proponer oportunidades de mejoras para el escalamiento de este.
- Analizar y proponer cambios para que los servicios sean compatibles con K8s.

- Proponer herramientas para adoptarlas en desarrollo y preparar el camino hacia la migración.
- Diseñar un clúster de K8s acorde a los requerimientos de la organización.
- Diseñar un plan de migración sin afectar la disponibilidad del sistema.
- Analizar los resultados para comprobar los beneficios de la migración.

CAPÍTULO 2: MARCO CONCEPTUAL

En este capítulo se profundizará en el funcionamiento actual de *smi-backend y smi-tar*, explicando su funcionamiento interno y como interactúan los diversos componentes de esta aplicación dentro de las instancias EC2, también se mencionarán ciertas configuraciones de la infraestructura actual y se presentará información de importancia para el desarrollo de este proyecto de título, como los costos asociados a la infraestructura. Finalmente se hará una breve introducción a la cultura DevOps, la cual es fundamental en el proceso de ejecución de este proyecto de título y además se realizará un cercamiento a Kubernetes y a las herramientas necesarias para que la migración sea exitosa.

2.1. Profundización en las configuraciones de la infraestructura actual

Para un mejor entendimiento y facilitar la comprensión de las diferencias entre la infraestructura actual y la que se propone se profundizará en la configuración actual del sistema.

2.1.1 Ambiente de desarrollo

No existe una configuración estándar para ejecutar las aplicaciones en los equipos locales de los desarrolladores, por lo que cada uno utiliza simplemente lo que le acomoda más. Algunos ejecutan directamente el repositorio sobre el Python instalado en sus máquinas e instalan las dependencias necesarias, otros crean ambientes virtuales basados en las configuraciones del proyecto y otros han incursionado en utilizar Docker.

2.1.2 Componentes de las aplicaciones

Los sistemas *smi-backend* y *tar-backend* se componen de 3 servicios llamados: *Backend, Beat y Worker.* El servicio llamado *Backend* es una aplicación en Django que ejecuta un servidor HTTP, *Beat* es la misma aplicación, pero ejecuta Celery [3], que se encarga de programar cuando un método[4] debería ejecutarse (estos métodos son llamados *Tasks*), finalmente el *Worker (proceso que corre asíncronamente las funciones)* es el encargado de ejecutarlas. Para que tanto Beat como el Worker funcionen es necesario contar con un intermediario de mensajería (*broker)*[5], por lo que existe también una instancia de Redis que actúa como tal.

[4] Una *Task* es una tarea que debe ejecutarse de forma asíncrona, y puede programarse para un momento especifico, por ejemplo, todos los fines de mes para calcular el consumo mensual de los medidores.

[5] Sistema capaz de recibir los argumentos e información necesaria para ejecutar una *Task*, el *beat* escribe en él y el *worker* lo lee.

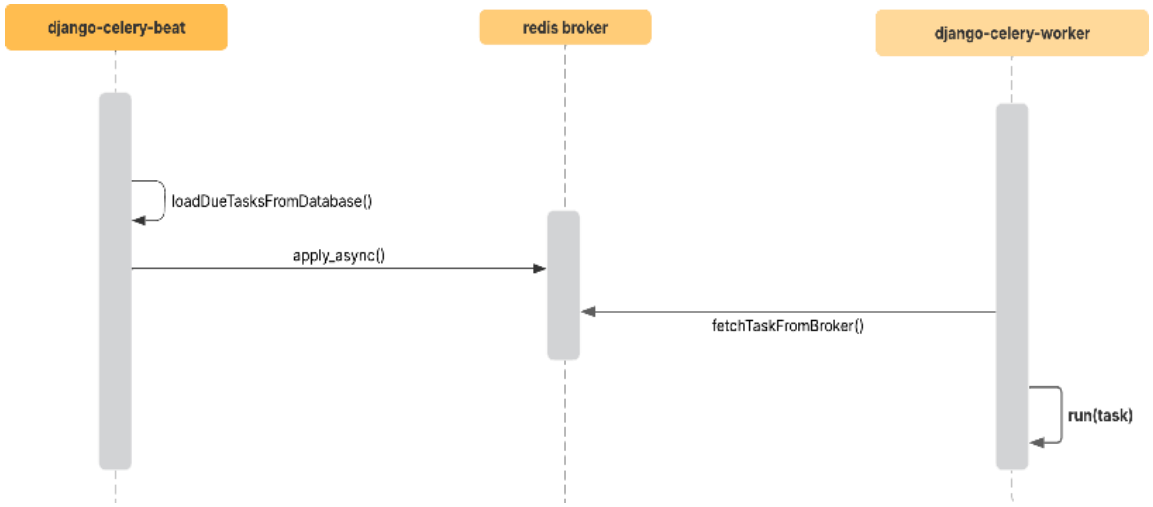

Figura 3: Representación del flujo de *Task* con Django y Celery en sistema SMI backend. Fuente: Elaboración propia.

Para ejecutar los servicios mencionados se utiliza *Supervisor*, quien se encarga de asegurar que se ejecuten correctamente los procesos en el ambiente en que se ejecutan, de escribir y persistir los registros del sistema (*logging*) y además también es capaz replicar estos procesos según se estime conveniente, permitiendo el escalamiento horizontal de los sistemas. Es relevante mencionar que la última versión estable de Supervisor es 4.2.5 lanzada en diciembre del 2022 [4].

También existe un proceso de Nginx que actúa como proxy reverso, sin configuraciones importantes y Memcached para la gestión de sesiones en la aplicación web [5].

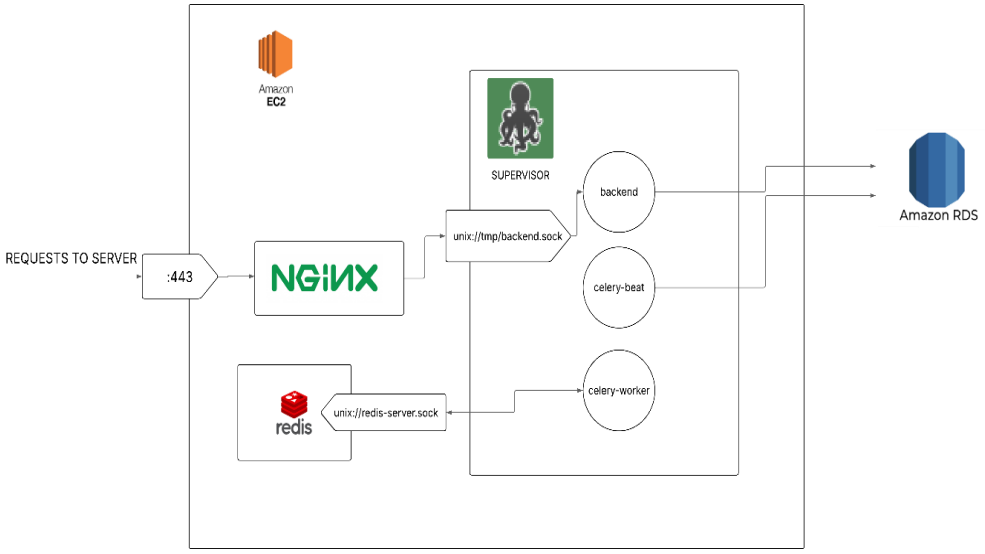

Figura 4: Arquitectura de los distintos componentes que interactúan dentro de las instancias EC2 por tenant. Fuente: elaboración propia.

2.1.2.1 Configuraciones por tenant

Para que cada *tenant* tenga sus propias configuraciones y sea único, existe un archivo no estándar llamado *copyConf.conf* en el que se escriben claves y valores mediante espacios y saltos de línea, que luego durante el proceso de despliegue se transformarán en variables de entorno de la instancia gracias a un *bash script* especialmente diseñado para esto.

```
1 DB_ENGINE postgresql
2 DB_HOST /var/run/postgresql
3 DB_PORT DB_PORT
4 DB_NAME db_$(echo ${SUB_DNS} | sed s,\-,_,g)
5 DB_USER user_$(echo ${SUB_DNS} | sed s,\-,_,g)
6 DB_PASS DB_PASS
```

Figura 5: Ejemplo de copyConf.conf. Fuente: elaboración propia.

2.1.2.2 Despliegue e integración continua

A pesar de que el despliegue de las aplicaciones está automatizado, sigue requiriendo intervención humana, debido a que el flujo se compone de pasos propensos a fallar fácilmente. Por ejemplo, el proceso de traducir el archivo *copyConf.conf* mencionado con anterioridad a variables de entorno. También se corren comandos de *git* para reconciliar las ramas dentro de las instancias de producción, si consideramos que es común realizar

cambios directos en ambientes productivos, el riesgo de que aparezcan conflictos entre la *branch local* y *origin* es altamente probable y común.

Figura 6: fragmento del script que se ejecuta para hacer deploy. Fuente: Elaboración propia.

Aunque en la imagen se observa que solo ejecutamos dos archivos de comandos en lote (*bash scripts),* ubicados en línea 3 y 13, internamente se llaman entre sí otros veintiún scripts para completar todo el flujo, los que realizan acciones tales como reinstalar dependencias, reiniciar los servicios (*backend, beat y worker*), asegurarse de que estén correctamente instalados Nginx y Redis, levantar el ambiente virtual, etc.

Si bien existe la idea de integración y despliegue continuos, no es una buena implementación debido a la constante intervención manual y desconfianza del flujo.

2.1.3 Infraestructura actual

Tanto como *smi-backend*, *smi-tar* y *smi-sync* se ejecutan en instancias de Amazon EC2 de las familias *t2*, *t3*, *t3a* y *t4g.* La selección de la familia y tamaño no siguen ninguna regla en particular, solo depende de cómo se han comportado anteriormente los *tenants* según el tamaño de los clientes[6]. Por otro lado, las bases de datos son instancias de Amazon RDS y siempre son del tipo *db.t3.micro*. En el Anexo C se puede encontrar más información respecto a las familias T2 de EC2 [6].

2.1.3.1 Costos

El costo por instancia, sin considerar las bases de datos ni otros servicios es de ~1.100 USD considerando los 42 *tenants* productivos, por temas de configuraciones de AWS solo se

[6] Cómo regla básica la cantidad de medidores instalados por cliente definirá el tamaño de las instancias, a mayor número de medidores es igual a mayor demanda de recursos computacionales.

puede consultar correctamente los costos dentro del mismo año, pero estos costos no deberían cambiar radicalmente.

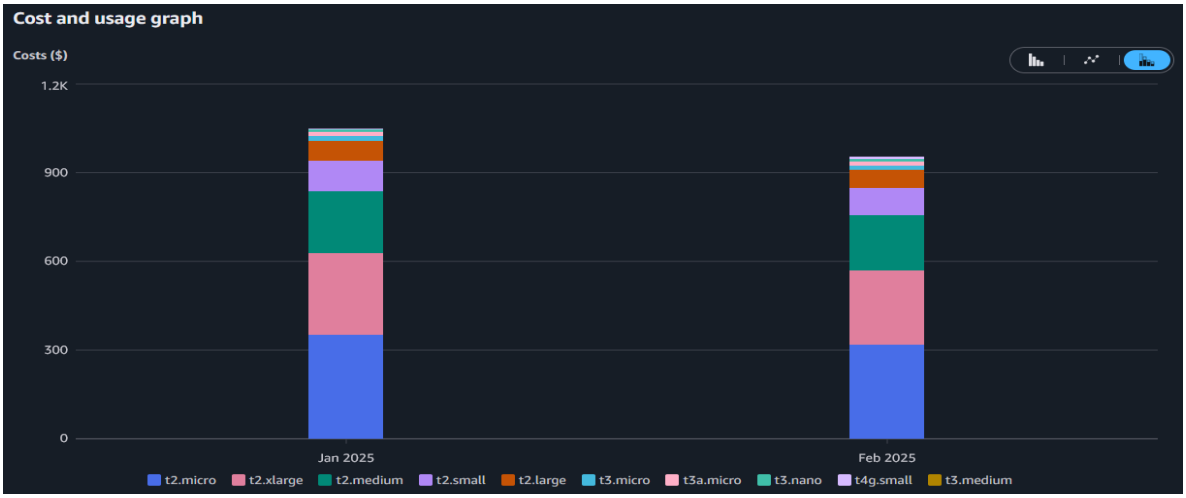

Figura 7: Costos de las instancias en producción para los *tenants* antes de migrar a Kubernetes. Fuente: Elaboración propia.

2.1.4. Monitoreo

Para el monitoreo del sistema se cuenta con dos herramientas que realizan tareas simples pero importantes: Sentry y Uptime Kuma.

2.1.4.1. Sentry

Sentry es una herramienta de pagos integrada a nuestra aplicación que permite capturar cualquier tipo de excepción o error, persistiendo la información de los eventos en caso de que se deba investigar algún incidente. También es posible generar alertas en caso de que cualquier error este ocurriendo de manera concurrente, permitiendo actuar a tiempo en caso de cualquier problema [6].

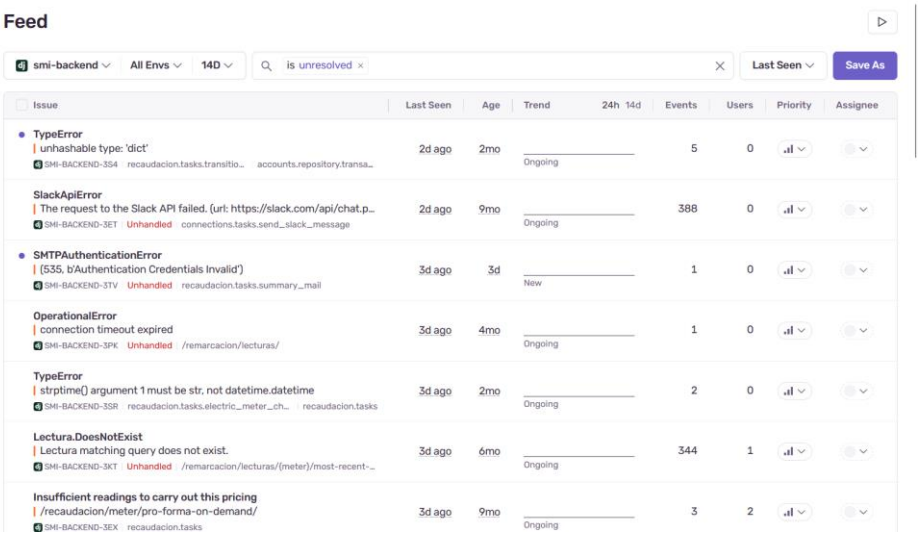

Figura 8: Excepciones y alertas capturadas por Sentry. Fuente: Elaboración propia.

2.1.4.2. Uptime Kuma

Uptime Kuma nos permite saber si nuestras aplicaciones están disponibles, ya que actúa enviando *pings* de manera continua a nuestros servicios y en caso de que la respuesta no sea la esperada, como por ejemplo errores 500 y 503. (por lo general los errores 5xx son resultado de algún error por lo que es útil notificar sus ocurrencias, pero se puede configurar alertas para cualquier respuesta HTTP) enviará una alerta notificando la instancia específica y problema correspondiente [7].

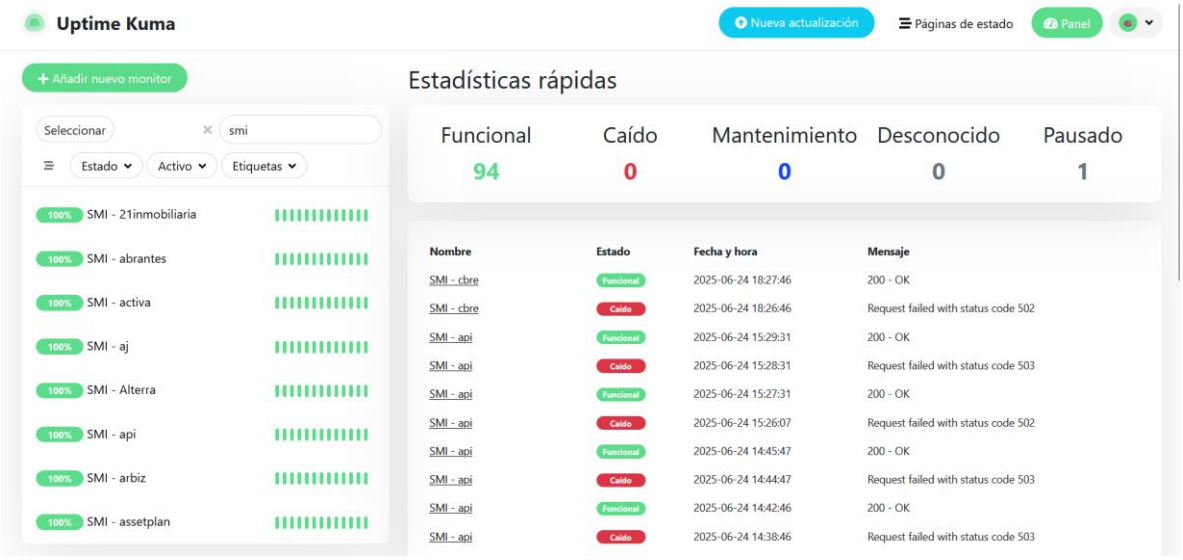

Figura 9: Interfaz de monitoreo de Uptime Kuma. Fuente: Elaboración propia.

2.2. La cultura DevOps

Lo que hoy conocemos como DevOps comenzó a cimentarse en el año 2007, por ese entonces en internet ya existía un gran número de discusiones en foros abiertos respecto a los problemas que conlleva la separación entre los equipos de desarrollo y operaciones. En ese mismo año Patrick Debois (conocido como el padre de la cultura DevOps) y otros colaboradores trabajarían en moldear una metodología capaz de resolver estas problemáticas [8].

2.2.1. ¿Cuál es el objetivo de la cultura DevOps?

El objetivo principal es agilizar y unificar el ciclo de vida completo del software, desde la planificación y priorización de nuevas funcionalidades hasta el monitoreo del software ya en producción, obteniendo *feedback* de todo el ciclo de manera constante. Para lograrlo es fundamental realizar un cambio cultural en los equipos involucrados que busca fomentar las buenas prácticas, comunicación ágil y la adopción de diversas herramientas que deben estar disponibles para todo el equipo.

2.2.2 El ciclo de vida DevOps

El ciclo de vida DevOps consiste en ocho etapas que buscan unificar las tareas de los equipos de desarrollo y operaciones. Suelen existir algunas variaciones, pero en este trabajo nos enfocaremos y utilizaremos la versión que describe Atlassian [9].

2.2.2.1. Descubre *(Discovery)*

En cada iteración en el proceso de construcción de nuestro producto como software, el equipo debe explorar opciones, organizar sus ideas, priorizar. Se debe construir de acuerdo con una estrategia que permita ir generando entregas de alto valor para el usuario. Esta etapa comienza desde cuando se planifica un nuevo *sprint* y se itera en el de forma constante durante el mismo.

2.2.2.2. Planifica *(Plan)*

Con el fin de construir software de manera rápida, eficiente y de calidad es importante que el equipo implemente algún enfoque ágil de desarrollo, el cual le permita ir construyendo de acuerdo con el valor y a la estrategia. Herramientas típicas como Jira y Trello son las que se requiere tener implementadas en esta etapa.

2.2.2.3 Construcción *(Build)*

Debido a que nuestro producto es un software, es importante contar con un sistema de control de versiones que nos permitan tener trazabilidad de los cambios, y al mismo tiempo nos brinde un enfoque de desarrollo que le permita a todos los miembros del equipo hacer modificaciones de forma simultánea. También es fundamental contar un *stack* de

herramientas para desarrollar las nuevas funcionalidades de la forma más fácil posible, acá empieza a tener peso el uso de Docker.

2.2.2.4. Prueba *(Test)*

El mantenimiento y desarrollo de *tests* para las funcionalidades del software es un pilar fundamental, es una capa que nos garantiza la confianza del software cuando se lanzan nuevas funcionalidades, estas estrategias se aplican en la integración y despliegue continuos.

2.2.2.5 Despliegue continuo *(Deploy)*

El despliegue continuo, le permite al equipo que desarrolla el producto generar entregas frecuentes y de forma automática.

2.2.2.6 Opera *(Operate)*

La operación es quien se hace cargo de todo el proceso de entrega del servicio a los clientes. Desde como diseñamos la aplicación, como se configura, como se despliega y hasta como se gestiona la infraestructura.

2.2.2.7. Observa (Observe)

Debemos ser capaces de detectar rápidamente cualquier problema en nuestra aplicación, ya sea que ya haya ocurrido y que se pueda presentar a futuro, para esto necesitamos herramientas de gestión del desempeño y otras para la gestión de incidencias o alertas.

2.2.2.8. Retroalimentación continua *(Continuous feedback)*

El equipo completo debe evaluar constantemente el impacto de sus nuevas funcionalidades de cara al cliente, rendimiento de la aplicación en términos de la experiencia de usuario, costos entorno a la infraestructura, y así sucesivamente debe ponerse continuamente en evaluación lo que se va construyendo a modo de ir mejorando.

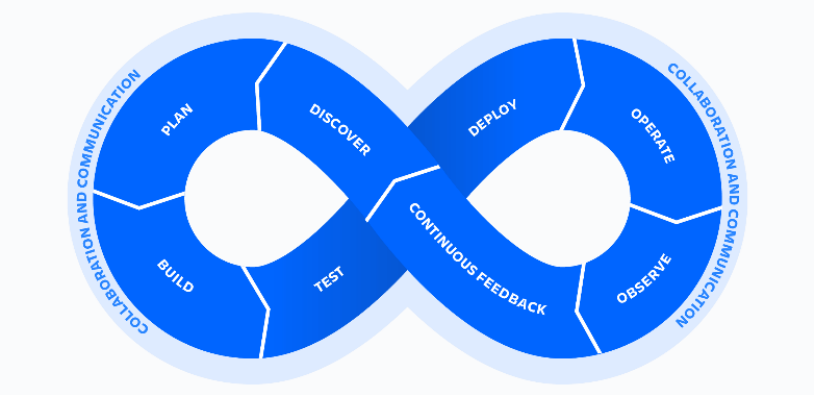

Figura 10: Ciclo DevOps. Fuente: Atlassian

2.3. Kubernetes y contenedores

En las últimas décadas, con la llegada de herramientas como Docker y Podman se ha popularizado el concepto de *containerización*, más tarde gracias al lanzamiento de Kubernetes por parte de Google como herramienta *open source* ha permitido evolucionar las soluciones a nivel de infraestructura mejorando la optimización de recursos, resiliencia y facilidad de escalar las aplicaciones sin tener que recurrir necesariamente a costos extras. Hoy en día Kubernetes y Docker son herramientas claves en organizaciones que implementan la cultura DevOps.

2.3.1. La containerización

Los contenedores es una idea que viene desde los años 70's, pero no fue hasta el 2013 cuando comenzó el *boom* de los contenedores, año en que se anuncia Docker como una herramienta revolucionaria, la que permite construir, actualizar, desplegar y gestionar contenedores de manera sencilla.

A container is a standard unit of software that packages up code and all its dependencies, so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings [10].

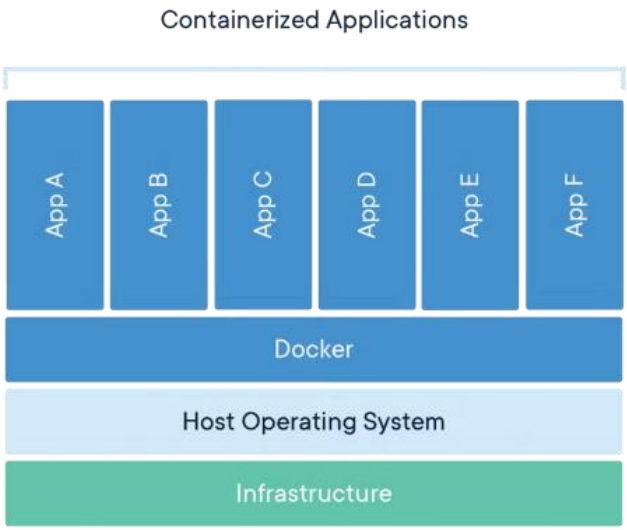

Figura 11: Diagrama de alto nivel sobre el funcionamiento de Docker. Fuente: Documentación de Docker

2.3.2. Kubernetes

Eran los años 2000 cuando Google, para mejorar el rendimiento de sus sistemas, comenzó a trabajar en el proyecto Borg. El sistema Borg de Google es un gestor de clústeres que permitió ejecutar cientos de miles de trabajos, de muchos miles de aplicaciones diferentes, en varios clústeres, cada uno con hasta decenas de miles de máquinas [11]. Una vez que Docker salió al mercado, los mismos desarrolladores de Borg vieron la oportunidad de llevar su creación más allá y con esto desarrollaron lo que sería el orquestador de contenedores llamado Kubernetes. Por diseño, Docker está pensado para correr las aplicaciones en un solo nodo. Kubernetes llegó a solucionar estas limitantes, logrando ejecutar múltiples contenedores en nodos distribuidos [12].

2.3.2.1. Que es un clúster

Un clúster en Kubernetes es el conjunto de nodos en los que se ejecutan las cargas de trabajo[7]. Existen dos tipos de nodos: un nodo maestro conocido como plano de control (*control plane)* y los nodos trabajadores, comúnmente llamados *workers.*

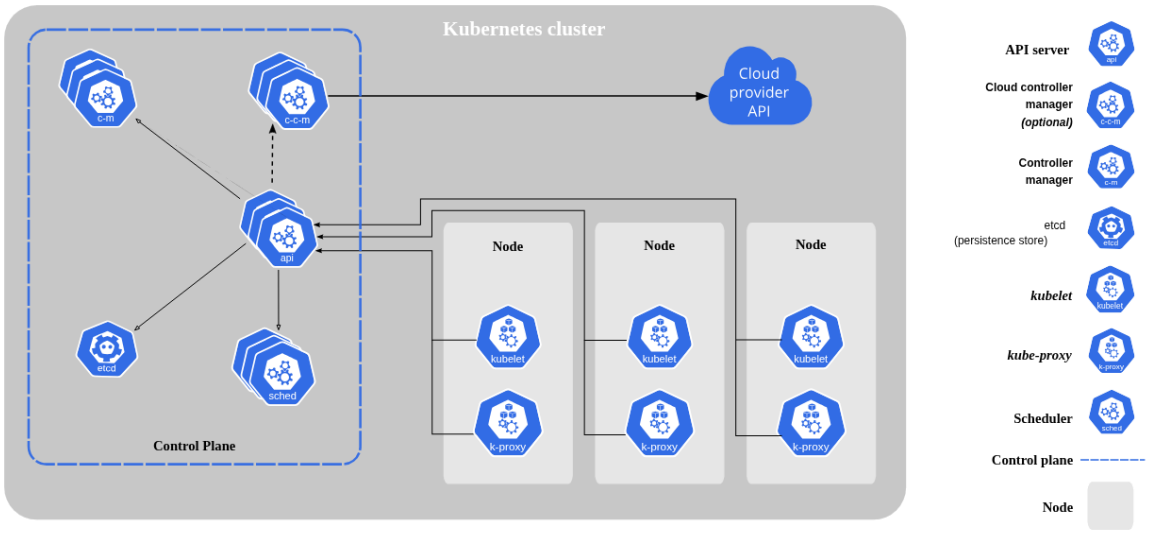

Figura 12: Arquitectura de Kubernetes. Fuente: Documentación de Kubernetes **[13]**

2.3.2.2 Componentes básicos de un clúster

A continuación, se describen los principales componentes de *Kubernetes*:

1. kube-api-server: Es la API encargada de comunicar todos los nodos, *workloads* y recursos, exponiendo los *endpoints* con los cuales podemos interactuar con el clúster.

[7] Una carga de trabajo es cualquier unidad de ejecución, es decir, las aplicaciones que se ejecutan en el clúster.

2. kube-controller-manager: se encarga de observar el estado actual de los distintos *workloads* existentes en el clúster, asegurándose siempre de que estos se encuentren en el estado deseado.
3. etcd: La base de datos en la cual se persiste la información y estado del clúster.
4. kube-scheduler: se asegura de que los *Pods* sean asignados a algún nodo, considerando requisitos como asignación de recursos, políticas de afinidad, etc.
5. Deployment: Permite declarar de manera descriptiva nuestros ReplicaSets y Pods.
6. ReplicaSet: Se encarga de mantener siempre la cantidad de replicas indicadas para un Pod.
7. Pod: La unidad más básica en Kubernetes, están compuesto de uno o más contenedores, por ende, es donde se ejecutan las aplicaciones.

Para leer sobre estos y otros componentes de manera más detallada leer la documentación oficial de kubernetes [13] .

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Figura 13: Ejemplo de un recurso del tipo *Deployment* de Nginx. Fuente: Elaboración propia.

con sólo esto logramos un ReplicaSet que se encargará de que siempre existan 3 Pods que ejecutaran un contenedor de Nginx versión 1.7.9, es decir, un Nginx escalado horizontalmente tres veces para dicha versión.

2.3.3. Probes

Los *probes* son un mecanismo que permite hacerle saber el estado de los pods al kubelet, de esta manera se asegura de que en todo momento los pods se encuentren en un estado saludable, en caso de que se detecte algún problema el kubelet toma medidas como reiniciar los Pods con problemas o no permitir que estos reciban tráfico por red. Existen tres tipos de *probes*: *readiness*, *liveness* y *startup*.

2.3.3.1. readiness probe

El *readiness probe* se encarga de determinar cuándo un pod está listo para recibir tráfico. Este está observando el estado tanto cuando un Pod se está inicializando y cuando ya está ejecutándose. En caso de detectar algún problema el pod queda fuera de la red, impidiendo que reciba tráfico.

2.3.3.2. startup probe

El *startup probe* da a conocer al kubelet cuando un pod efectivamente se inició correctamente, evitando que sean eliminado por el mismo kubelet cuando el inicio es demasiado lento.

2.3.3.3. liveness probe

El *liveness probe* se encarga de verificar el estado de los contenedores dentro de un Pod, en caso de detectar algún error o fallo propio de la ejecución del contenedor se encargará de reiniciarlo.

2.3.4. Auto escalamiento en Kubernetes

Para auto escalar pods en Kubernetes existe el *HorizontalPodAutoscaler*, el cual permite a los Pods escalar horizontalmente según una métrica definida. Es posible escalar según el uso de CPU o memoria.

2.4. Observabilidad y monitoreo

La observabilidad y el monitoreo son dos conceptos que por sí solos no tienen sentido, pero ambos en conjunto son una herramienta poderosa, no puede existir la observabilidad sin el monitoreo y viceversa. En cualquier sistema de software es deseable que existan herramientas de este tipo, facilitando el entendimiento de lo que ocurre, logrando llegar a las respuestas del por qué ocurren las cosas [14]. Actualmente en el mercado existe una amplia gama de opciones, tanto *open source* como servicios de pago *self-hosted* [8] o SaaS.

2.4.1. La importancia de la observabilidad en DevOps

La etapa llamada *observe* en el ciclo de vida DevOps está especialmente enfocado en la observabilidad de los sistemas debido a los potenciales beneficios que trae consigo. En este

[8] Deben instalarse y ejecutarse en la infraestructura propia de quien paga el servicio.

punto no basta solo con tener herramientas especializadas, sino también planear lo que se quiere observar, es decir, estudiar qué métricas queremos capturar y analizar con un fin claro y concreto de esta manera es posible encontrar las respuestas a problemas complejos y comprender de mejor forma el comportamiento de nuestros sistemas.

2.4.2. Beneficios y ventajas de la observabilidad en Kubernetes

Al inicio, cuando se decide utilizar un clúster de Kubernetes como la base de la infraestructura también puede significar complejizar el entendimiento de cómo funcionan las cosas. Es importante tener en cuenta que nos volvemos fuertemente dependientes del estado de la red, tanto en la que se comunican nuestros nodos como la red interna propia de Kubernetes. Los Pods intentan constantemente utilizar recursos computacionales, por lo que se deben limitar según las necesidades de estos y evitar que ciertos componentes dejen al resto sin recursos suficientes. ¿Cómo podemos determinar los recursos necesarios para cada componente?, ¿Es posible entender la red e intentar sacar conclusiones a partir de esta para comprender lo que está pasando? Estas son algunas de las preguntas que podemos responder gracias a la observabilidad y el monitoreo, permitiéndonos iterar de forma progresiva hacia un clúster maduro.

2.4.3. Herramientas fundamentales

Se mencionarán las herramientas fundamentales en las que existe interés de instalar en la nueva infraestructura, las cuales cuentan con versión *open source* e integraciones para clústeres de Kubernetes.

Figura 14: algunas herramientas que se usarán en el proyecto. Fuente: Elaboración propia.

2.4.3.1 Grafana

Grafana es una aplicación capaz de agrupar múltiples fuentes de datos, con el fin de utilizarlos y transformarlos en información a través de distintos *dashboards*. Permite integraciones con plataformas cloud para centralizar el monitoreo total de los sistemas, además que permite configurar alertas cuando se cumplen las condiciones establecidas [15].

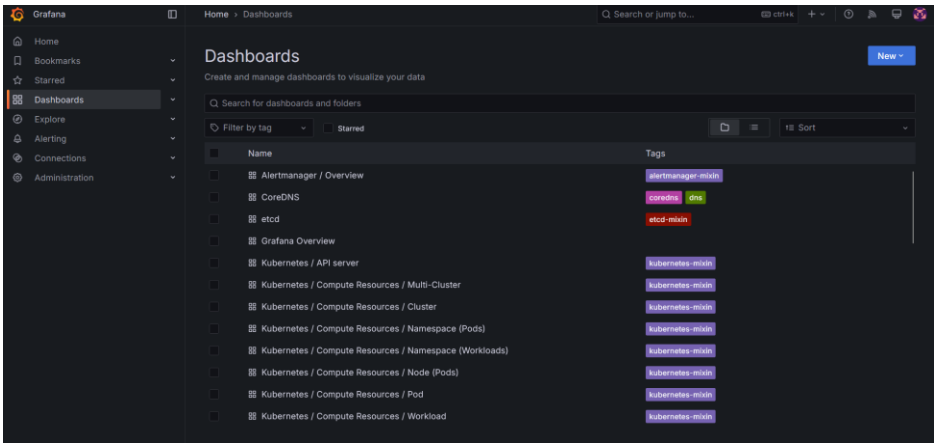

Figura 15: *dashboards* disponibles en Grafana. Fuente: Elaboración propia.

2.4.3.2 Loki

Loki permite acceder a los registros del sistema de todos los Pods del clúster, es decir, todos de todos nuestros sistemas en un solo lugar. Tiene un lenguaje de consultas simple que nos permite consultar exactamente lo que queremos. Es posible integrar Loki con servicios como Amazon S3 para mantener esta información en caso de ser necesario [16].

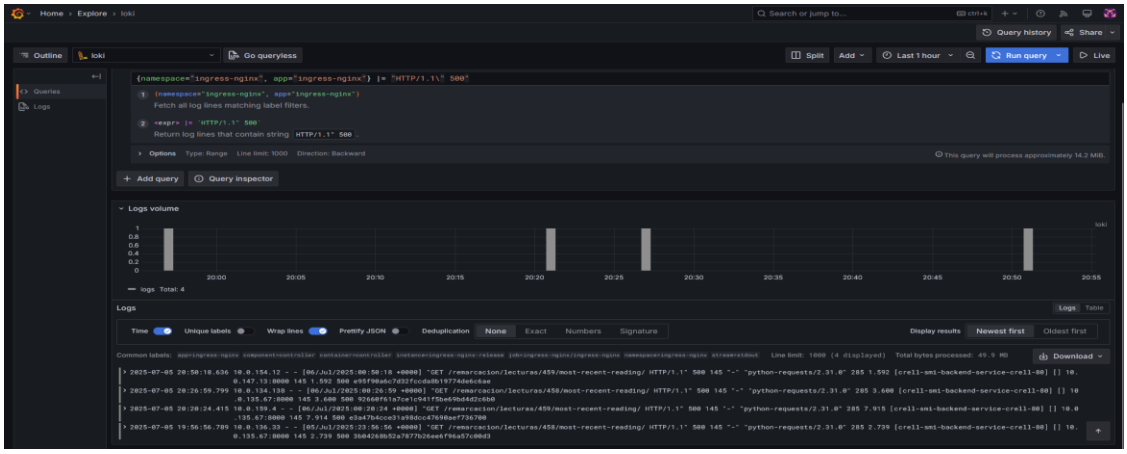

Figura 16: Logs obtenidos mediante Loki. Fuente: Elaboración propia.

2.4.3.3 Prometheus

Las métricas de nuestros sistemas son esenciales tanto para asegurar su correcto funcionamiento como para aportar información valiosa que impulse la operación y toma de

decisiones del negocio. Gracias a estas métricas es posible visualizar información, como por ejemplo para entender el estado del clúster durante un incidente, entender el comportamiento propio de cada aplicación para anticiparnos a posibles fallos, etc. Existen maneras de exponer métricas personalizadas desde el código de las aplicaciones, y de esta manera estudiar el funcionamiento del negocio, por ejemplo, contar números de requests asociados a una compra, y a partir de esto generar *dashboards* para entender cuestiones propias de los negocios [17].

Figura 17: Métricas obtenidas con Prometheus sobre el uso de recursos de un clúster de Kubernetes. Fuente: elaboración propia.

2.5 GitOps

GitOps es una estrategia fundamental en equipos que utilizan las metodologías de la cultura DevOps, nos permite automatizar el despliegue continuo de nuestras aplicaciones, convirtiendo nuestros manifiestos de Kubernetes en *IaaC* o infraestructura como código.
Herramientas como ArgoCD o Flux CD son las más populares que cuentan con versiones *open source* y graduadas por *The Linux Cloud Foundation*.
Ambas son sistemas capaces de leer el repositorio en el que versionamos nuestros componentes de Kubernetes y encontrar en el cambio de estado en los manifiestos, como por ejemplo cambios en los recursos computacionales asignados o cambio de versión en las imágenes de Docker que utilizan los Pods [18].

2.5.1 Flux CD

Flux CD es una herramienta de despliegue continuo *open source,* que nos permite declarar en manifiestos de Kubernetes, el estado de todas nuestras aplicaciones y componentes en un repositorio especifico, es decir, todo lo que queremos que se ejecute en un clúster de Kubernetes debe si o si estar declarado en el repositorio. Nos brinda una variada cantidad de nuevos componentes de Kubernetes que nos permiten configurar todo lo que se necesita [19].

2.5.1.1. Kustomization

Con *Kustomization* podemos declarar las rutas del repositorio en las que Flux CD debe leer para encontrar las configuraciones específicas para cada componente.

```yaml
apiVersion: kustomize.toolkit.fluxcd.io/v1
kind: Kustomization
metadata:
  name: backend-alterra-always
  namespace: flux-system
spec:
  interval: 10m
  timeout: 5m
  sourceRef:
    kind: GitRepository
    name: flux-system
  path: ./applications/smi/backend/alterra.enerlink.cl/always
  prune: true
```

Figura 18: Ejemplo de Kustomization. Fuente: Elaboración propia.

2.5.1.2. HelmRepository

El componente *HelmRepository* permite utilizar de manera simple Helm charts para instalar herramientas dentro del clúster de manera sencilla, dándole todo el control de estas a Flux CD. Esto provoca que Flux CD descargue la información correspondiente a una aplicación específica para que sea manejada de manera sencilla.

```yaml
apiVersion: source.toolkit.fluxcd.io/v1
kind: HelmRepository
metadata:
  name: grafana
  namespace: repositories
spec:
  interval: 24h0m
  url: https://grafana.github.io/helm-chart
```

Figura 19: Ejemplo de HelmRepository. Fuente: Elaboración propia.

También es posible automatizar *Pull Requests*[9] con cambios en el versionado de nuestras aplicaciones, notificar errores de sincronización o permite el despliegue a través de etapas dependientes entre sí cuando se requiera un orden.

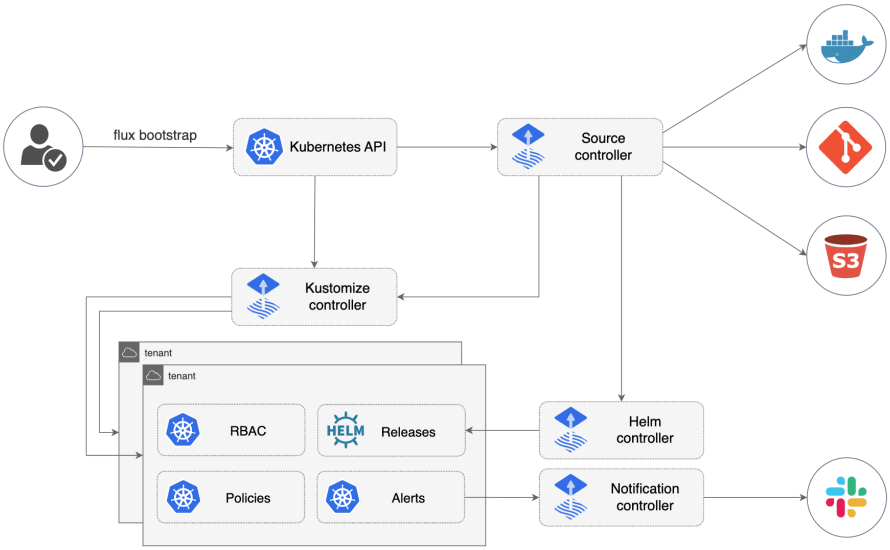

Figura 20: Diagrama de arquitectura del funcionamiento de Flux CD. Fuente: Documentación Flux CD

2.6 Amazon Web Services

Amazon Web Services (AWS) es uno de los mayores proveedores de servicios cloud del mundo ofreciendo diversos productos a precios relativamente convenientes, como se ha mencionado antes, actualmente se utiliza EC2 y RDS y en un futuro se incorporarán otros servicios que se describirán a continuación. [20]

2.6.1 Amazon EC2

En Amazon EC2 principalmente nos permite crear y administrar instancias de máquinas virtuales. Este servicio es la base de otros que requieren recursos computacionales, como por ejemplo bases de datos RDS, clústeres de Kubernetes EKS, instancias de cache como Redis, etc. [21].

2.6.2. VPC

El servicio VPC o Virtual Private Cloud permite crear y administrar redes privadas dentro de la infraestructura de AWS, sus componentes principales son las *subnets*, *security groups*, NAT [22].

[9] Pull requests o merge requests, dependiendo de que sistema se utiliza, por ejemplo: GitHub, Gitlab, Gitea, etc.

CAPÍTULO 3: PROPUESTA DE SOLUCION

Se ha diseñado un plan de migración que busca implementar una fuerte cultura DevOps, conceptos y herramientas de manera progresiva dentro del ciclo de vida del Software de forma paulatina. El plan consiste en cuatro fases enfocadas en darle solución y resolver dudas sobre cómo se solucionarán problemas existentes en el presente mediante las nuevas herramientas e infraestructura que estarán a disposición de todos. De esta manera se busca que el equipo se sumerja en esta nueva manera de hacer las cosas de forma progresiva. También se detallarán tanto las decisiones previas al levantamiento del clúster de Kubernetes como durante la misma. Finalmente, un plan de acción para identificar posibles problemáticas iniciales y futuras, aspirando a una infraestructura resiliente y confiable.

3.1 FASE 1: PREPARARSE PARA EL FUTURO

El objetivo de esta fase es introducir los contenedores en el flujo de desarrollo de todas las aplicaciones de las que se compone SMI, gracias a esto es posible dar a conocer sobre Docker a los desarrolladores que no estén familiarizados con las tecnologías de *containerización* y sus conceptos, y lo más importante se soluciona el problema de los ambientes inconsistentes entre los desarrolladores.

Durante esta fase se programaron cuatro reuniones para discutir entre el equipo los beneficios de utilizar Docker cuando se desarrolla y la importancia de compartir ambientes idénticos entre los desarrolladores. Además, se abordaron conceptos técnicos cómo las diferencias entre una imagen y un contenedor, utilidad de los volúmenes y Docker Compose.

Finalmente, se debe desarrollar un Dockerfile y docker-compose para empezar a aplicar todo lo que se aprendió durante las conversaciones.

3.1.1. Aprendizaje en conjunto

El aprendizaje en conjunto consiste en realizar reuniones de equipo durante la fase 1, donde cada reunión aborda un concepto distinto que permite entender la razón de los cambios, hacia dónde estamos apuntando y cuestiones técnicas. A continuación, se presenta una tabla con las descripciones de las reuniones que se hicieron durante esta fase.

Tabla 1: Planificación de reuniones

fecha	tema	descripción
18-11-2024	problemáticas de la infraestructura actual	Se habló sobre todos los problemas presentados en este proyecto de título y se discutió la razón por la que es necesaria la migración.
22-11-2024	Docker como base para el desarrollo	Se presentó a Docker como la herramienta de facto para desarrollar en local, además se enseñaron conceptos básicos y presentaron ejemplos.
01-12-2024	Kubernetes básico	Breve introducción sobre todo lo básico que se debe saber antes de usar Kubernetes.
16-12-2024	Kubernetes avanzado	Se realizaron pruebas con clústeres locales, haciendo despliegues de aplicaciones para entender los potenciales beneficios que trae consigo.

3.1.2. El Dockerfile

Lo primero y más importante es diseñar un Dockerfile, en este archivo se declaran los pasos que se deben seguir para construir la imagen[10] deseada empaquetando el código fuente, dependencias y librerías necesarias. En el futuro esta imagen será la que Kubernetes utilizará para ejecutar la aplicación, pero en esta fase nos servirá para incluirla en nuestro docker-compose y transformar esto en el ambiente de facto para el desarrollo.

Las imágenes de Docker idealmente deben limitarse a estar construidas en base a solo lo que de verdad necesitan según el fin para el cual existen. Es importante que el Dockerfile propuesto cumpla con esta condición, ya que nos permite hacer imágenes especializadas para una misma aplicación y así evitar tener imágenes con tamaños significativamente grandes. La versatilidad del Dockerfile nos permite en un solo archivo definir imágenes base,

[10] No confundir imagen con contenedor, una imagen es una forma estandarizada de docker en el que se empaquetan todas las librerías, binarios, archivos, etc. Esta se genera a partir del Dockerfile, un contenedor es el proceso en el que dicha imagen está en ejecución.

y partir de esta instalar ciertas dependencias según queramos utilizarla en producción, desarrollo o para ejecutar *tests* con integración continua.

```
1  FROM python:3.12.10-slim-bullseye AS python-base
2  ENV APP_HOME="/smi-backend" \
3      LANG=C.UTF-8 \
4      PYTHONFAULTHANDLER=1 \
5      PYTHONUNBUFFERED=1 \
6      VIRTUAL_ENV="/.cache/pypoetry/virtualenvs/smi-backend"
7  ENV PATH="$VIRTUAL_ENV/bin:$PATH"
8  RUN python -m venv /.cache/pypoetry/virtualenvs/smi-backend
9  RUN --mount=type=cache,target=/var/lib/apt/lists \
10     --mount=type=cache,target=/var/cache,sharing=locked \
11     apt-get update \
12     && apt-get upgrade -y -q \
13     && apt-get install -y -q --no-install-recommends \
14     # meta-packages that are essential to compile software
15     build-essential \
16     # postgress client (psycopg2) dependencies
17     libpq5 libpq-dev \
18     # translation dependency
19     gettext \
20     # html to pdf dependency
21     wkhtmltopdf \
22     # cleaning up unused files to reduce the image size
23     && rm -rf /var/lib/apt/lists/* \
24     && apt-get purge -y --auto-remove -o
   APT::AutoRemove::RecommendsImportant=false \
25     && apt-get autoremove \
26     && apt-get clean
27 WORKDIR $APP_HOME
```

Figura 21: Dockerfile que representa a la imagen base de SMI-backend. Fuente: Elaboración propia.

En este caso se define una imagen llamada *python-base* que indica el sistema operativo y la versión de *Python* a utilizar, en este caso el sistema operativo será *Debian Bullseye* y *Python* 3.12.10. Luego se instalan todas las librerías necesarias en el sistema operativo para el correcto funcionamiento de nuestra aplicación, el fin de esta imagen es ser la base para todas las versiones que necesitemos. Las demás versiones simplemente deben estar basadas en Python-base e instalar solo lo que corresponde.

```
1 FROM python-base AS development
2 ENTRYPOINT ["./docker/development-entrypoint.sh"]
3 RUN --mount=type=cache,target=$POETRY_CACHE_DIR \ poetry install --
  no-root --with test,dev
4 COPY --link . .
5 CMD ["./docker/start-development.sh"
```

Figura 22: Imagen especializada para desarrollo local, en línea 3 se indica instalar solo dependencias para test y desarrollo. Fuente: Elaboración propia.

3.1.3 Docker compose

Docker compose es una funcionalidad de Docker que permite orquestar contenedores de manera rápida y sencilla, esto significa que podemos replicar un ambiente similar a producción de una forma declarativa sin afectar a otros flujos, aunque esto suele ocuparse igualmente para orquestar aplicaciones de producción, sólo lo utilizaremos para desarrollo. La sencillez de docker compose radica en que al igual que en Kubernetes, la manera de declarar la interacción y su comportamiento de los servicios es a través de un archivo tipo YAML, el cual se especializa en la serialización de datos, pero es ampliamente usado para la configuración de infraestructura como código [23].

Para empezar, se debe descomponer el sistema para crear nuestros servicios tal cual cómo se despliegan e interactúan en producción, es decir, deberíamos crear un servicio para: *backend*, *beat*, *worker*, Redis, base de datos y memcached. No existe necesidad de que declaremos un servicio de Nginx en nuestro *docker-compose.yml*.

El resultado esperado es poder ejecutar los contenedores de *smi-backend* y *tar-backend* en una misma red interna de docker, para poder así comunicar ambos sistemas entre sí, los detalles de la implementación en el Anexo B.

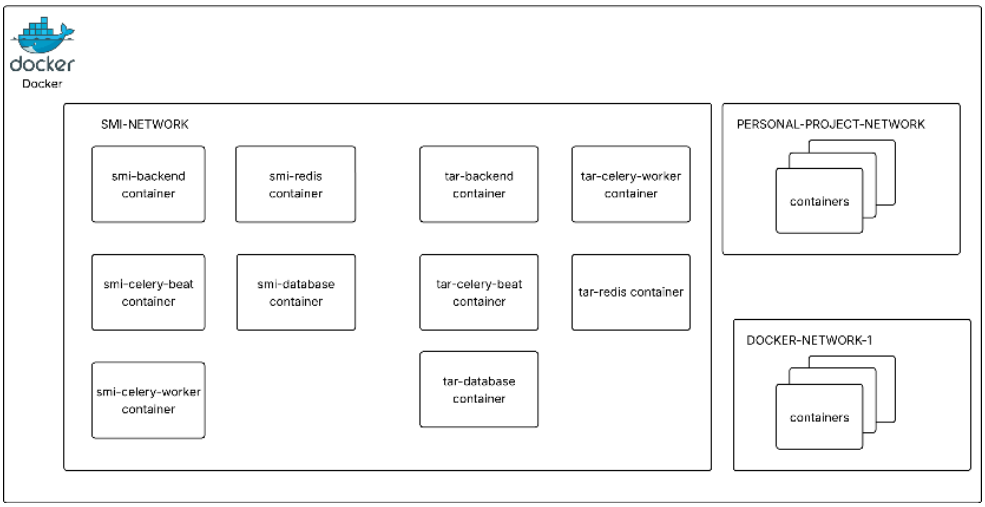

Figura 23: Estado deseado del ambiente de desarrollo. Fuente: Elaboración propia.

3.1.4. Resumen de la fase 1

La fase 1 tuvo una duración de 5 semanas, comenzando el 7 de octubre y se dio por finalizada el día 11 de noviembre, teniendo tanto el Dockerfile como el *docker-compose.yaml* en la rama llamada *master*[11] de los repositorios. Aunque fue relativamente

[11] En el repositorio donde se aloja el proyecto la rama llamada *master* es donde llegan todas las funcionalidades que pasarán a producción.

sencillo el desarrollo de lo relacionado con Docker, lo más importante es que se logró dar a entender al equipo las razones del por qué desarrollar con Docker trae beneficios por sobre lo que existía antes, además que es el primer paso para lograr una cultura DevOps sólida.

3.2 FASE 2: Entender el futuro

Esta fase consiste en estudiar las diferentes problemáticas actuales que suceden de manera recurrente y analizar cómo se deberían abordar una vez la infraestructura se encuentre migrada. El criterio para considerarlos un problema abordable en esta fase se basa en la frecuencia con la que se deben realizar normalmente durante un ciclo de desarrollo (3 semanas), criticidad para la operación o deuda técnica que deba necesariamente ser cuestionada previo a hacer la migración por limitaciones importantes.

3.2.1. Redis

Existen dos opciones para el futuro de Redis en el sistema: se mueve dentro del clúster o pagamos el servicio de caches de Amazon llamado ElastiCache. La primera opción no involucra costos adicionales directamente, pero debido a la gran cantidad de instancias que se deben crear involucraría una alta demanda de recursos y esto es igual a más nodos en el clúster, resultado en pagar por más recursos. La segunda opción ya se usa dentro de la organización en otra célula de desarrollo encargada de otro proyecto, quienes aseguran que el servicio cumple con los requerimientos esperados e involucra un gasto que se puede asumir desde un inicio. Considerando lo mencionado anteriormente y la experiencia de la otra célula de desarrollo utilizando el mismo servicio se decidió ir por la segunda opción, con instancias *cache.t3.micro* se pagarían ~12 USD mensuales por instancia. El Redis hasta ahora solo se encargaba de gestionar web sockets del sistema, pero a partir de ahora tendrá un rol más protagónico, ya que para abaratar costos también se utilizará para la gestión de caches del sistema.

3.2.2. Memcached

Redis se hará cargo de los cachés, por lo que todo lo que tenga que ver con Memcached en el sistema será deprecado, pasando a ser administrados por competo por Redis. Con esta decisión reducimos los esfuerzos del mantenimiento y operación del sistema al reducir los componentes de los cuales depende.

3.2.3. El archivo *copyConf*.conf

Se debe eliminar todo lo relacionado al archivo *copyConf.conf*, ya que en Kubernetes no tiene sentido su existencia. Uno de los componentes básicos en Kubernetes es el *Secret*, esto nos permite definir las variables de entorno de las cuales la aplicación depende para ejecutarse correctamente de manera sencilla.

3.2.4. Los deploys

Los pipelines ya no gatillaran los despliegues, simplemente deberían encargarse de construir las imágenes de Docker correspondientes a cada *release*, Flux CD se encargará de mantener el clúster en el estado correspondiente.

3.3. FASE 3: EMBARCARSE EN EL VIAJE

Terminada la fase 2, el equipo ya está más involucrado y entiende el porqué de los cambios propuestos. Ahora es posible proponer épicas para los sprints enfocadas en el desarrollo e implementación de la solución propuesta.

3.3.1. ElastiCache

Para pasar desde lo que existe en Redis hoy a los Redis de Amazon ElastiCache se diseñó un documento para realizar la operación de la forma más segura posible, evitando la perdida de *Tasks* y así reducir problemas operacionales.

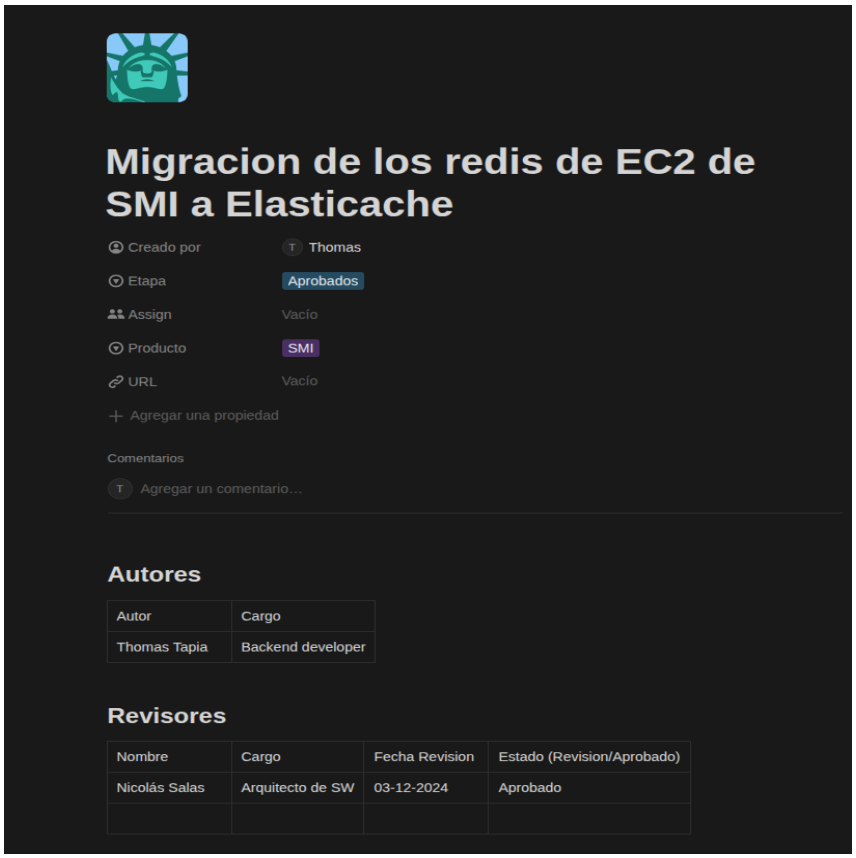

Figura 24: Portada del documento de pasos propuestos para migración de Redis a ElastiCache. Fuente: Elaboración propia.

3.3.1.1. Código fuente incompatible

El código encargado de la gestión de configuraciones provenientes de las variables de entorno está fuertemente acoplado a la infraestructura, así que solo es posible establecer conexiones mediante *Unix sockets*, por lo que es incompatible para instancias de Redis ElastiCache se admite conexiones solo mediante TCP.

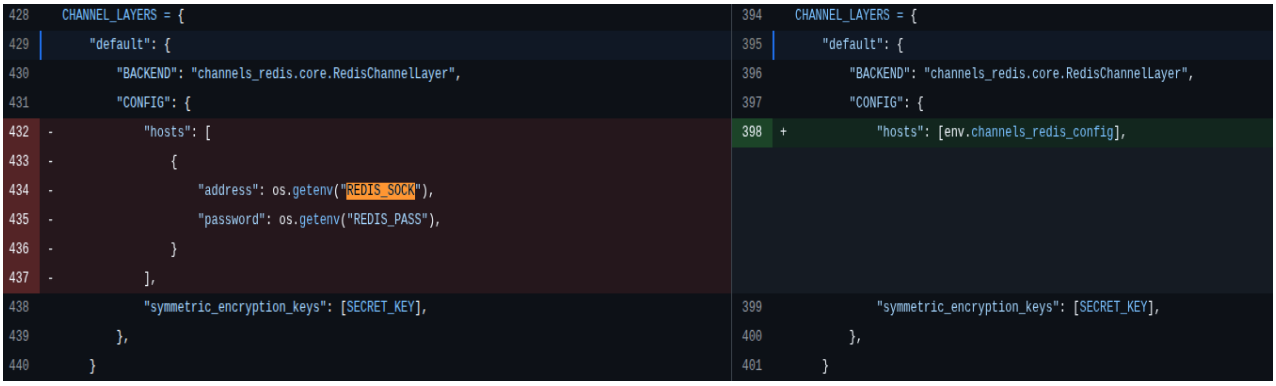

Figura 25: Cambios para permitir la conexión TCP con ElastiCache. Fuente: Elaboración propia.

El cambio es simple, deprecar las variables *REDIS_SOCK y REDIS_PASS*, dejando únicamente una variable para el *endpoint* de conexión de las instancias, este *refactor* se lanzó en la versión v3.7.0 de smi-backend.

3.3.1.2. Discrepancia en las versiones de Redis

Para evitar problemas se identificaron las versiones de Redis presentes en las distintas instancias de producción y se encontraron tres versiones distintas.

Tabla 2: Versiones de Redis en las instancias EC2

Versión Redis	Cantidad de instancias
6.4.2	15
6.0	25
7.0.1	2

La versión más alta disponible en ElastiCache en ese momento para migrar en Redis era 7.0.1, por lo que es fundamental actualizar en todas las instancias la versión de Redis a 7.0.1 previo a realizar la migración.

Se propone ejecutar un script mediante una máquina pivote que tiene acceso a todas las instancias EC2, con el fin de actualizar Redis en las instancias que corresponda, una vez finalizado este paso está todo listo para realizar la migración.

3.3.1.3. Migración a ElastiCache

Al ser las *Tasks* de *celery* criticas para la operación es importante realizar la migración evitando la pérdida de estas, por lo que se realizó un manual para lograr este proceso de la manera más limpia posible, este procedimiento se detalla en el anexo A.

3.3.2. El clúster de Kubernetes

Siguiendo la línea de utilizar Amazon como proveedor de infraestructura, se pagará por el servicio de Elastic Kubernetes Service, y se deben cumplir los siguientes requisitos no funcionales:

- Los nodos deben ser privados, no pueden tener IP pública
- Deben existir al menos dos zonas de disponibilidad
- Permitir acceso granular a los usuarios mediante IAM

Configuraciones preliminares. Se deben realizar configuraciones iniciales para configurar la red como las siguientes:

- Inicialmente se debe configurar la VPC en la que se alojará el clúster, nombrándola *smi-kubernetes-vpc* y con bloque CIDR 10.0.0.0/16 y debe tener dos *availability zones* (us-east-1a y us-east-1b).
- Configurar cuatro *subnets*, una privada y una publica por cada *availability zone*
- Configurar un NAT Gateway por *availability zone*, asociándole una IP publica, de tal forma que sea la puerta de todo el tráfico desde internet, el NAT debe estar presente en las *subnets* públicas.

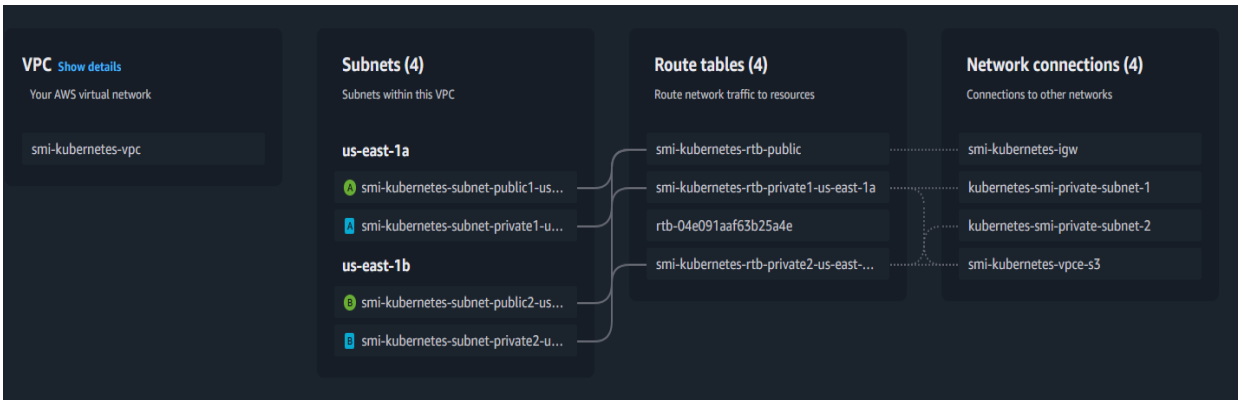

Figura 26: Esquema de red para la VPC del clúster. Fuente: Elaboración propia.

Con la red creada se puede configurar el clúster, considerando agregar los siguientes *add-ons*:

1. kube-proxy: Gestiona el tráfico de red interno del clúster, es el encargado de enrutar los paquetes a los servicios que corresponde.
2. CordeDNS: Servicio que permite la comunicación de los recursos internos del clúster a través de nombres DNS.
3. EBS CSI driver: Permite configurar volúmenes de datos para los Pods del clúster, utilizando volúmenes EBS
4. VPC CNI: Permite que los *Pods* reciban IP's en el rango de la red VPC del clúster, facilitando la comunicación con otros recursos propios de Amazon.
5. Metrics server: Se encarga de recopilar métricas sobre el uso de recursos computacionales de los Pods, permitiendo utilizar *HorizontalPodAutoscaler*.

Las características iniciales para el clúster son la siguiente:

- Numero de nodos: 2
- Tipo de máquina para los nodos: t2.medium
- CPU total: 4 vCPU
- Memoria total: 8 Gib

3.3.3. FluxCD y el GitOps

Una vez el clúster se encuentre operacional se procede con la instalación de Flux CD, para esto lo primero es definir un repositorio en el cual alojaremos toda nuestra infraestructura. El repositorio llevará de nombre *enerlink-gitops* y su estructura inicial base será la siguiente.

Figura 27: Organización del directorio para FluxCD. Fuente: Elaboración propia.

En el directorio llamado *clusters* irá toda la información correspondiente a la configuración del repositorio, en ella se encontrarán todos los manifiestos que le indican a Flux las reglas para hacer deploy, que aplicaciones debe mantener en el clúster y cuales no, además de las propias configuraciones internas de Flux. En el directorio *applications/smi* deben ir todos los manifiestos que indican las configuraciones propias de cada Deployment para los servicios que queremos desplegar en el clúster. Flux los leerá y reflejará cada cambio en este directorio en el estado de los distintos recursos de Kubernetes.

Para instalar Flux existen distintos métodos, pero el más adecuado es mediante una clave SSH propia del repositorio, así evitamos asociar a Flux con algún trabajador que podría dejar la organización en algún momento.

Figura 28: Comando para configurar FluxCD en el clúster asociando una clave privada. Fuente: Elaboración propia.

El comando anterior descarga los manifiestos necesarios para instalar los controladores de Flux dentro del clúster, los agrega al directorio *clusters/flux-system* y finalmente configura el controlador con la información proporcionada en las *flags* del comando, solo resta hacer *git push* para finalizar la instalación.

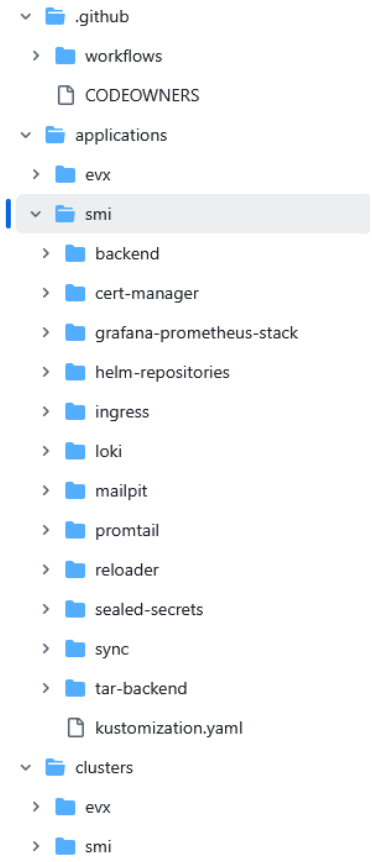

Figura 29: Estructura final de repositorio GitOps ya madurado, incluye otros proyectos que también se migraron debido a los beneficios obtenidos. Fuente: Elaboración propia.

3.3.4. Herramientas clave para Kubernetes

Con Flux CD configurado y ejecutándose correctamente en el clúster se deben instalar las herramientas que agilizan la operación propia del clúster. Estas herramientas se deben disponibilizar mediante Helm Charts para ser configuradas e instaladas, pero podrían ser confusas inicialmente y no tan fácil de entender, por lo que Flux CD administrará por completo estos recursos mediante su *feature* integrada que permite la gestión de Helm Charts.

3.3.4.1. Observabilidad y monitoreo

Se usará el stack open source que se ha mencionado en este trabajo de título: Grafana, Prometheus, Loki y Promtail. Con estas herramientas logramos obtener un sistema robusto sin costo alguno. Estas herramientas son ampliamente conocidas por su robustez cuando se combinan, teniendo una de las comunidades de soporte más grandes del mundo.

3.3.4.2. Cert-manager

Cert-manager es un proyecto *graduated* por The Native Computing Foundation, que permite emitir y renovar certificados TLS para nuestros dominios de manera automática cuando se requiera. Es posible integrarlo a Ingress como Nginx. [24]

3.3.4.3. Ingress Nginx

Con Ingress Nginx podremos crear una vía de acceso desde internet a nuestros Pods cuando se intente acceder mediante el host, este recurso cumple la misma función que el Nginx de la infraestructura anterior, pero ahora es un controlador de Kubernetes capaz de escalarse y atender las solicitudes para todos los clientes. [25]

3.3.4.4. sealed secrets

Por default los *Secret* se persisten en el clúster en base64, un método que no es seguro. En el Secret deben ir todas nuestras configuraciones y datos sensibles, como contraseñas para bases de datos, etc. Con *sealed Secret* podemos encriptar todos los valores mediante una llave asimétrica. [26]

3.3.4.5. reloader

En kubernetes cuando se cambian valores en un secreto los cambios no se ven reflejados en las variables de entorno del Pod correspondiente hasta que ocurra algún reinicio del Pod. La función de *reloader* es identificar cambios en los Secret y reiniciar inmediatamente los Pods que los consuman. [27]

3.3.5. Migración de *smi-backend* al clúster

Smi-backend es la primera aplicación que debe pasar al clúster por su importancia y cantidad de gastos que lleva consigo su operación. Inicialmente creará un ambiente *sandbox*, configurado de igual manera como lo sería un *tenant* productivo, la idea es hacer pruebas de manera exhaustiva de flujos críticos en la operación del software y del negocio mientras se realiza monitoreo activo con Grafana, así poder determinar valores iniciales para asignar recursos a los futuros tenants de producción.

Debido a la ausencia de herramientas de monitore/observabilidad anteriormente es imposible conocer el comportamiento del software en momentos de importancia, muchas de las tareas criticas ocurren en la madrugada y nadie puede estar ahí cada día rescatando esa información.

3.3.5.1. configuración inicial

Debido a que smi-backend es un sistema *single-tenant* hay que replicar muchas veces lo mismo para cada cliente, por lo que se debe reducir la duplicación de configuraciones en su mayoría, de esta forma el repositorio es más fácil de mantener.

En *applications/smi/backend/base* se dejará el *template* base para todos los tenants, de aquí se heredará el contenido del manifiesto usando *Kustomization*.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: smi-backend
  namespace: sandbox
  labels:
    app: smi-backend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: smi-backend
  template:
    metadata:
      labels:
        app: smi-backend
    spec:
      containers:
      - name: smi-backend
        image: smi-backend
        ports:
        - containerPort: 8000
        envFrom:
        - secretRef:
            name: smi-env
        resources:
          limits:
            cpu: 200m
            memory: 1024Mi
          requests:
            cpu: 10m
            memory: 256Mi
```

Figura 30: primera versión del manifiesto de SMI-backend para realizar pruebas en *sandbox*. Fuente: Elaboración propia.

Kustomization permite heredar contenido de manifiestos de Kubernetes, con esto es posible también editar secciones para personalizar según los requerimientos no funcionales de cada *tenant.*

```yaml
resources:
  - ../../base
  - autoscaler.yaml
nameSuffix: -eurocorp
namespace: eurocorp
images:
  - name: smi-backend
    newName: 147136798105.dkr.ecr.us-east-1.amazonaws.com/smi-backend # {"$imagepolicy": "flux-system:smi-backend-prod-image-policy:name"}
    newTag: v4.12.2 # {"$imagepolicy": "flux-system:smi-backend-prod-image-policy:tag"}
patches:
  - path: backend-patch.yaml
    target:
      kind: Deployment
      name: smi-backend
```

Figura 31: *Kustomization* que hereda el manifiesto base para uno de los clientes de producción, en el campo *patches* se declara una personalización de este. Fuente: elaboración propia.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: smi-backend
spec:
  template:
    spec:
      containers:
      - name: smi-backend
        resources:
          requests:
            cpu: 600m
            memory: 512Mi
          limits:
            cpu: 600m
            memory: 1024Mi
```

Figura 32: Este manifiesto sobrescribe la asignación de recursos del manifiesto de Figura 41. Fuente: Elaboración propia.

3.3.5.2. Pruebas de estrés en sandbox

Lo más importante ahora es lograr determinar una cantidad de recursos adecuados para los distintos servicios de los *tenants* productivos, para esto se levantó un ambiente de sandbox en el clúster donde inicialmente se establecieron los siguientes recursos por servicio:

Tabla 3: Configuración inicial de recursos para SMI backend previo a las pruebas

Servicio	CPU requested	CPU limits	Memory requested	Memory limits
smi-backend	10m	200m	256Mi	1024Mi

smi-worker	10m	100m	256Mi	256Mi
smi-beat	10m	100m	256Mi	256Mi

Las pruebas consisten en utilizar el software de manera normal, ejecutar flujos críticos propensos a fallos, e interactuar con los otros sistemas. Durante las pruebas se monitoreará mediante Grafana toda la actividad para llegar una decisión final.
La prueba más importante consiste en ejecutar el flujo de *tarificación*[12], el ambiente ha sido configurado para realizar esta operación con 8000 medidores ficticios, dando un total de 768.000 lecturas, considerando 4 peticiones asíncronas por segundo. Con esto podemos establecer los límites para los *Pods* y asegurarnos de que puedan utilizar los recursos suficientes en momentos críticos en la operación del software.

3.3.5.2.1. Utilización recurrente de recursos

Esta prueba tiene por objetivo medir y establecer el uso normal de recursos para estos servicios, sin someterlos a ninguna carga. Con esto podemos definir un valor correcto en los *requests* de los *Deployments*.

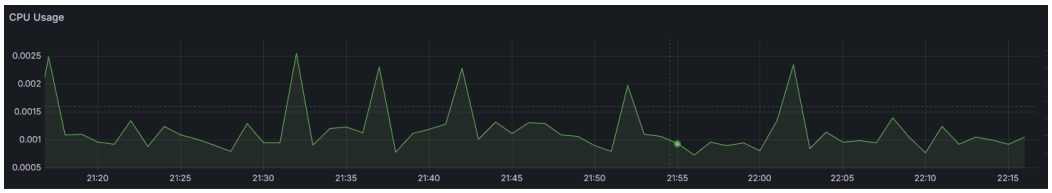

Figura 33: uso de CPU para el *beat* observado durante las pruebas. Fuente: Elaboración propia.

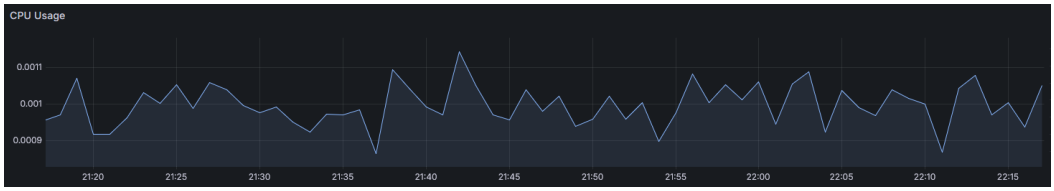

Figura 34: uso de CPU para el *worker* observado durante las pruebas. Fuente: Elaboración propia.

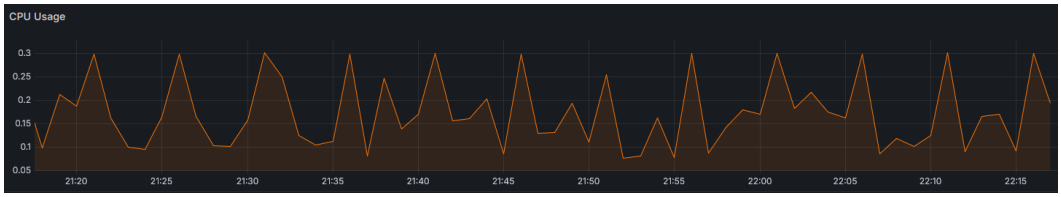

Figura 35:uso de CPU para el *backend* observado durante las pruebas. Fuente: Elaboración propia.

[12] El flujo de tarificación consiste en enviar las lecturas de un periodo especifico a la aplicación *smi-tarificacion*, con el fin de calcular el valor a pagar por cada arrendatario.

Se puede observar que el worker y beat tienen un consumo de CPU insignificante en momentos de poca carga de trabajo, mientras que el *backend* está constantemente bajo demanda.

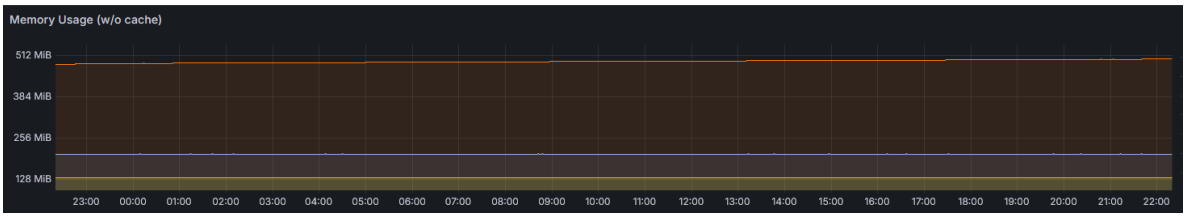

Figura 36: Uso para todos los pods de sandbox. Naranjo es *backend*, Azul es *worker* y amarillo es *beat*.
Fuente: Elaboración propia.

El consumo de memoria es bastante equilibrado, sin grandes *peaks* de uso observables en 24 horas, facilitando la asignación de *requests* al presentar un comportamiento estable.

3.3.5.2.2. Ejecución de flujos críticos e interacción con otros sistemas

El beat y backend presentan en todo momento un comportamiento estable, sin presentar notables alzas en el consumo tanto de memoria como de CPU, esto se evidencia ya que en todo momento la web estuvo disponible, con una latencia aproximada de 1.3 solicitudes por segundo. Por otro lado, el worker, al ser quien ejecuta todos los flujos de alta carga presenta altos peaks en uso de CPU, en memoria alcanzó máximos cercanos a 1.4Gib, por lo que los límites de este servicio deben presentar una holgura respecto a los requests para asegurar el correcto funcionamiento y limitar los riesgos.

Finalmente, luego de realizar todas las pruebas, los valores finales para el *template* base de los tenants quedó de la siguiente manera:

Tabla 4: Estimación de recurso para SMI backend después de las pruebas

Servicio	CPU requested	CPU limits	Memory requested	Memory limits
smi-backend	100m	300m	256Mi	1024Mi
smi-worker	1m	1000m	256Mi	2048Mi
smi-beat	2m	100m	128Mi	256Mi

3.3.6. Traspaso de tenants a producción y *deploys* por etapa

Con el manifiesto base para listo y recursos iniciales establecidos se puede proceder con la creación de los tenants dentro del repositorio *enerlink-gitop*s. En el mismo directorio en

que esta la base se deben agregar directorios con el nombre del host para identificar claramente cada tenant, dentro de ellos se crean tres directorios: *always*, *pre* y *deploy*.

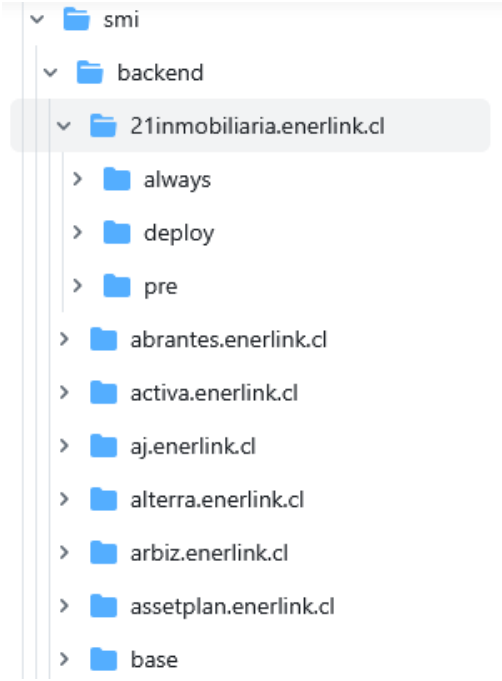

Figura 37: Estructura del repositorio, con los tenants en producción. Fuente: Elaboración propia.

En el directorio *always* se definirán los recursos de Kubernetes que deben existir en todo momento, es decir, si no están presentes no puede haber deploy del tenant. El directorio pre es el encargado de ejecutar todo comando necesario antes de levantar la instancia final de producción, como, por ejemplo, las migraciones de la base de datos. Finalmente, en el directorio deploy se definen los manifiestos para los servicios de producción, es decir el *backend*, el *worker* y el *beat*.

Para Flux, cada uno de estos directorios representará una etapa dentro del proceso de reconciliar el repositorio y los que existe en el clúster, es decir en el deploy. Para definirlo se debe agregar la aplicación dentro de del directorio *clusters*, por ejemplo, para 21inmobiliaria se define lo siguiente:

```yaml
apiVersion: kustomize.toolkit.fluxcd.io/v1
kind: Kustomization
metadata:
  name: backend-21inmobiliaria-always
  namespace: flux-system
spec:
  interval: 10m
  timeout: 5m
  sourceRef:
    kind: GitRepository
    name: flux-system
  path: ./applications/smi/backend/21inmobiliaria.enerlink.cl/always
  prune: true
---
apiVersion: kustomize.toolkit.fluxcd.io/v1
kind: Kustomization
metadata:
  name: backend-21inmobiliaria-pre
  namespace: flux-system
spec:
  interval: 10m
  dependsOn:
    - name: backend-21inmobiliaria-always
  timeout: 10m
  sourceRef:
    kind: GitRepository
    name: flux-system
  path: ./applications/smi/backend/21inmobiliaria.enerlink.cl/pre
  prune: true
  wait: true
  force: true
---
apiVersion: kustomize.toolkit.fluxcd.io/v1
kind: Kustomization
metadata:
  name: backend-21inmobiliaria-deploy
  namespace: flux-system
spec:
  interval: 10m
  dependsOn:
    - name: backend-21inmobiliaria-pre
  timeout: 10m
  sourceRef:
    kind: GitRepository
    name: flux-system
  path: ./applications/smi/backend/21inmobiliaria.enerlink.cl/deploy
  prune: true
  wait: true
```

Figura 38: Ejemplo de deploy por etapas mediante Flux CD para clientes en producción. Fuente: Elaboración propia.

En el manifiesto de la figura X, se definen tres recursos del tipo Kustomization: *backend-21inmobiliaria-always*, *backend-21inmobiliaria-pre y backend-21inmobiliaria-deploy*. Mediante el campo *dependsOn* se puede hacer la reconciliación en el orden que se estableció, así nos aseguramos de tener un deploy ordenado y robusto en el que siempre estarán presentes los recursos de Kubernetes necesarios, evitando errores inesperados.

3.3.7. La necesidad del auto escalamiento

Durante los días iniciales en que los tenants estaban ejecutándose en el clúster apareció un nuevo problema: se estaban volviendo comunes los errores 503 (servicio no disponible) en el canal de alertas que notifica Uptime Kuma, este error es indicio directo de que clúster está impidiendo el tráfico hacía la aplicación. El culpable de esto: el *readinessProbe,* quién a prepósito dejaba a la aplicación fuera de los servicios alcanzables por red.

Existían momentos específicos en que ciertos tenants recibían una demanda de recursos, causando *spikes* del uso de CPU, como consecuencia resultó en la lentitud en el procesamiento de requests, esto provocaba que el kubelet entienda que el contenedor estaba no listo para recibir tráfico provocando que quede fuera de la red, al no llegar los pings de Uptime Kuma al servidor la respuesta era siempre 503. Uno de los detalles que se observaron era que el *timeout* de los requests que se hacían al *startupProbe* era de tan solo 1 segundo, por lo que se decidió aumentar a 5. Otra medida a tomar fue el auto escalamiento de los Pods con HorizontalPodAutoscaler, para así cuando se requiera un alto uso de CPU los pods escalen horizontalmente, aumentando la disponibilidad del sistema. La métrica para saber cuándo escalar se definió a partir del uso de CPU durante un rango determinado de tiempo y la cantidad de pods a escalar dependen de la cantidad de medidores por cliente.

Tabla 5: Configuración de auto escalamiento.

Cantidad de medidores	Mínima cantidad de pods disponibles.	Máxima cantidad de pods para escalar
0 a 200	1	2
200 a 500	2	3
500 a 1000	2	4
Mas de 1000	3	5

CAPÍTULO 4: VALIDACIÓN DE LA SOLUCIÓN

El clúster ha entrado en funcionamiento en finales de febrero, y hoy en día ya han pasado aproximadamente cuatro meses de funcionamiento, mientras que los cambios culturales y uso de nuevas herramientas es un proceso cercano a los 8 meses. Durante este tiempo han pasado cinco *sprints* de desarrollo, en los que se han notado cambios notables en el desempeño del equipo, optimización de costos, reducción de horas hombre en resolver errores, etc. En este capítulo se detallarán cada uno de los beneficios que ha traído consigo todo este cambio cultural y de infraestructura en la empresa, como de igual manera ciertos puntos que se deben mejorar.

4.1 Costos

La reducción de costos es uno de los cambios positivos, se logró pasar de más de 40 instancias EC2 a sólo 8, presentando una reducción aproximada de 1400 USD, la que comenzó a ser notoria en abril, ya que durante el periodo de marzo aún se convivía con las maquinas antiguas por si ocurría algún incidente.

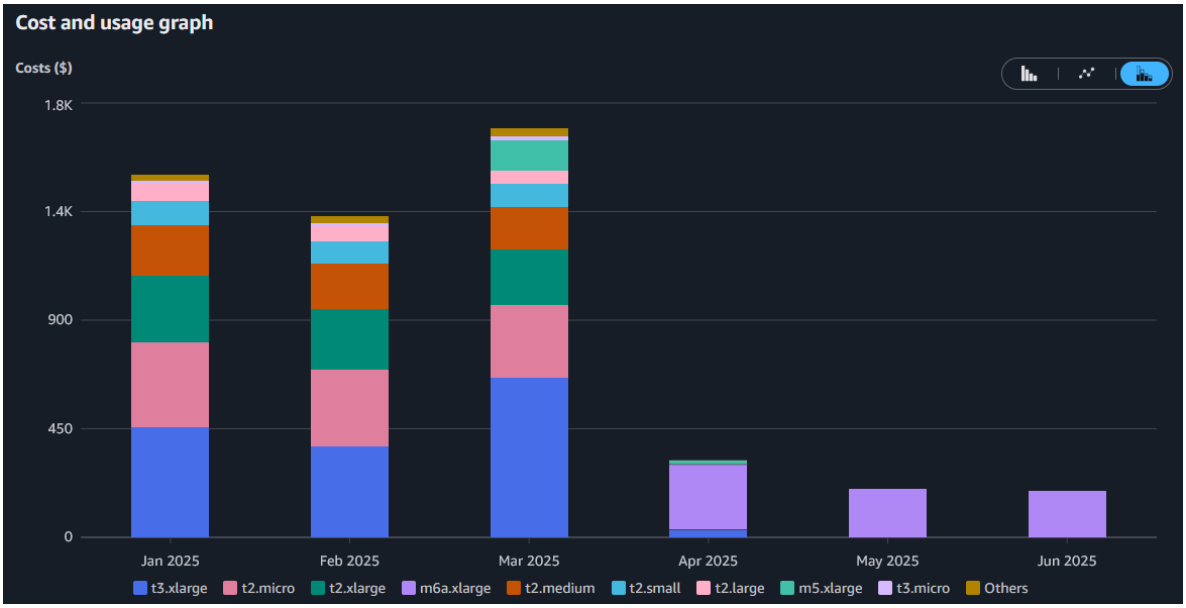

Figura 39: Gráfico que evidencia la reducción de costos, luego de la migración a Kubernetes. Fuente: Elaboración propia.

4.2 Eficiencia del equipo

Se ha notado una evidente mayor eficiencia por parte del equipo de desarrollo, las métricas de los *sprints* han presentado mejoras en comparación a periodos anteriores, en conversaciones con el equipo esto se puede deber a que ya no se requieren tantas horas

hombre en solucionar problemas relacionados a *deploys* fallidos, los errores son mucho más fáciles de depurar y las herramientas de desarrollo son más eficientes y menos propensas a fallar, agilizando la velocidad de los desarrolladores.

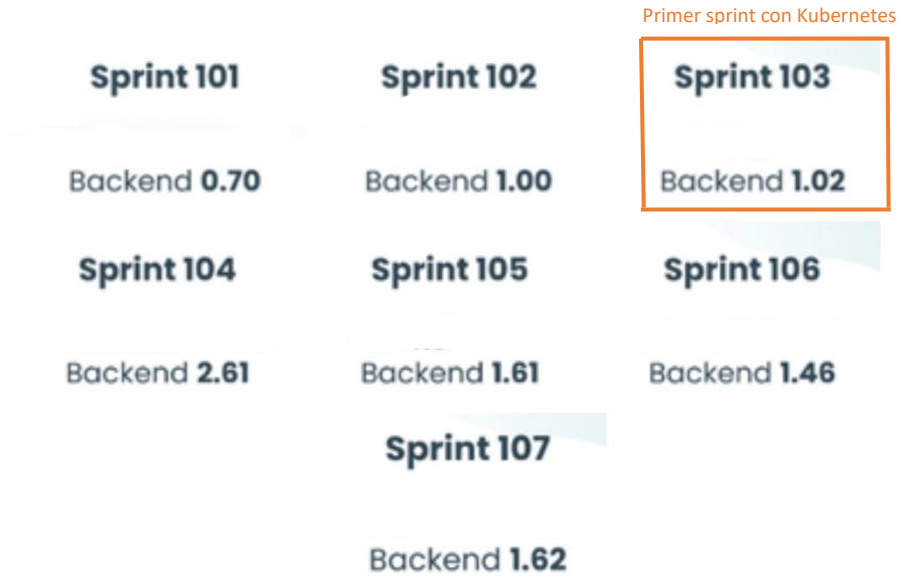

Figura 40: Evolución de métrica que calcula los puntos de *sprint* logrados por día por desarrollador, Fuente: Elaboración propia.

En las figuras se puede apreciar el incremento que se ha ido experimentando en los puntos de historia (*story points*)[13] conseguidos por día durante los *sprints* por cada desarrollador a partir del *sprint* 103. Esta métrica es importante para el equipo, ya que refleja el desempeño de los desarrolladores.

4.3. Migraciones posteriores

La migración de la infraestructura a Kubernetes, siendo smi-backend la primera migración en migrarse agradó y gustó a la célula de desarrollo por completo, la implementación de flujos de la cultura DevOps trajo notables beneficios, lo que produjo el querer migrar todo lo que existía en su momento en la infraestructura ya deprecada. Hoy en día todas las aplicaciones que estaban en EC2 ahora pertenecen al clúster de Kubernetes, lo que significa que tanto *smi-backend*, *smi-tar* y *smi-sync* son parte de la nueva infraestructura.

[13] Los puntos de historias son la unidad que mide el esfuerzo por tareas a desarrollar en un *sprint*.

CAPITULO 5: CONCLUSIÓN

El desarrollo de este proyecto de título permitió concretar la migración de un sistema *single-tenant* con infraestructura poco escalable hacia un entorno moderno orquestado con Kubernetes, logrando implementar además una sólida cultura DevOps en el equipo.

Con los resultados, se puede afirmar que tanto el objetivo general como los específicos planteados al inicio fueron alcanzados satisfactoriamente. El objetivo general consistía en llevar la infraestructura de los sistemas de *Smart Metering Software* (SMS) a un clúster de Kubernetes para mejorar el escalamiento y optimizar los costos operacionales, elevando la confiabilidad del producto mediante alta disponibilidad y resiliencia, todo esto impulsando a su vez la adopción de prácticas DevOps en la organización. En retrospectiva, esta meta se cumplió plenamente: hoy el sistema opera sobre Kubernetes con mayor capacidad de crecimiento, costos más eficientes y niveles superiores de disponibilidad, habiéndose incorporado en el proceso una serie de herramientas y prácticas que consolidan la cultura DevOps dentro del ciclo de vida del software.

En primer lugar, se realizó un diagnóstico exhaustivo del estado inicial de la plataforma, identificando las limitaciones que impedían su escalabilidad y eficiencia. Este análisis crítico permitió evidenciar diversos problemas en la infraestructura original, por ejemplo, la existencia de entornos de ejecución inconsistentes, un proceso de despliegue con alta intervención manual y propenso a errores, dependencia de componentes que dificultaban el escalado y un monitoreo limitado de la operación. A partir de este diagnóstico se propusieron oportunidades de mejora concretas orientadas a preparar el terreno para la migración. De igual manera se examinó la compatibilidad de los servicios actuales con un entorno Kubernetes, detectando qué cambios serían necesarios en el código y la configuración para asegurar que todas las aplicaciones pudieran ejecutarse dentro de contenedores en el nuevo clúster. Esta etapa inicial fue crucial: brindó una comprensión clara de las brechas entre la situación de partida y el objetivo deseado, y delineó los requerimientos técnicos que guiaron las siguientes fases del proyecto.

El siguiente paso fue introducir nuevas herramientas y prácticas en el flujo de desarrollo para facilitar la transición tecnológica. Aquí se destaca la incorporación de la containerización mediante Docker. Se incluyeron los archivos Dockerfile y Docker Compose para cada servicio, estandarizando la forma de construir y ejecutar las aplicaciones. Esto permitió eliminar las discrepancias entre los ambientes de cada desarrollador y alinearlos con el futuro ambiente de producción en Kubernetes.

Luego se procedió a diseñar la nueva infraestructura basada en Kubernetes, acorde a los requerimientos de la empresa. Se tomaron decisiones técnicas importantes durante esta etapa para garantizar que el clúster de Kubernetes cumpliera con altos estándares de seguridad, rendimiento y disponibilidad. En línea con las políticas de la empresa, se optó por desplegar el clúster en AWS, usando el EKS, aprovechando así un plano de control

administrado que reduce la complejidad operativa. Se configuró el clúster con múltiples zonas de disponibilidad distribuyendo los nodos entre al menos dos *Availability Zones*, para asegurar tolerancia a fallos regionales, y se estableció que todos los nodos de cómputo fueran privados, sin IP pública, reforzando la seguridad de la solución. Asimismo, se dimensionó inicialmente el clúster con dos nodos e integrando *addons*, para replicar funcionalidades básicas de la infraestructura previa. Esta configuración inicial sentó las bases de una plataforma escalable y de alto desempeño, capaz de adaptarse al crecimiento de la empresa, adicionalmente se decidió externalizar ciertos servicios de apoyo para optimizar recursos como por ejemplo, en vez de desplegar Redis dentro de Kubernetes, lo que consumiría una cantidad considerable de recursos dado el número de instancias necesarias, se migró este componente a *Amazon ElastiCache* y con esta decisión técnica se buscó reducir la sobrecarga del clúster además de mejorar la confiabilidad de el caché, a cambio de un costo mensual razonable. Por otro lado, también se eliminó el antiguo uso de Memcached, centralizando todas las funciones de caché en Redis para simplificar la arquitectura y el mantenimiento. Cada una de estas decisiones contribuyó a que la infraestructura propuesta cumpliera con el propósito de ser más eficiente, escalable y fácil de operar.

Paralelamente al diseño del clúster, se elaboró un plan de migración detallado para mover gradualmente los sistemas al nuevo entorno sin afectar la continuidad operacional, por lo que se planificó una coexistencia temporal de la infraestructura antigua y la nueva. Durante el primer mes posterior al despliegue de Kubernetes, las antiguas instancias en EC2 se mantuvieron activas en paralelo al clúster, listas para asumir el tráfico en caso de cualquier incidencia inesperada en el nuevo sistema. Este enfoque conservador brindó seguridad que resultó fundamental para ganar confianza en la estabilidad de la nueva plataforma antes de desactivar completamente la anterior. El proceso de migración se ejecutó tenant por tenant, trasladando uno a uno los 42 clientes al clúster. Antes de migrar cada grupo de servicios, se llevaban a cabo pruebas exhaustivas en entornos de *staging containerizados*, validando que cada componente (backend, *workers* de tareas, etc.) funcionara correctamente bajo Kubernetes. Un caso ilustrativo de la minuciosidad de esta fase fue la migración de Redis a ElastiCache: para evitar la pérdida de tareas de Celery durante el cambio de un servicio de cache a otro, se diseñó un procedimiento paso a paso que incluía pausar temporalmente la generación y ejecución de *Tasks*, extraer un volcado de datos (*dump*) desde la instancia antigua de Redis y restaurarlo en la nueva instancia gestionada. Solo tras verificar la integridad de los datos y la conectividad de los servicios, se procedía a apuntar los aplicativos al nuevo Redis administrado. Gracias a tácticas como esta, se logró transferir componentes críticos sin incurrir en pérdidas de información ni tiempos muertos significativos. En suma, la migración fue ejecutada de forma transparente para los usuarios finales, cumpliendo a cabalidad con el objetivo específico de no afectar la disponibilidad del sistema en ningún momento.

Culminada la transición, se llevó a cabo una evaluación integral de los resultados para comprobar en qué medida la solución implementada cumplía con las expectativas y

objetivos planteados. Los beneficios concretos que experimentó la organización tras la migración evidencian el éxito del proyecto. En términos de costos operativos, la diferencia fue sustancial: se logró reducir drásticamente la infraestructura subyacente, pasando de más de 40 instancias EC2 independientes a solamente 8 nodos de Kubernetes, con la consiguiente caída en el gasto mensual de la plataforma. Este ajuste representó un ahorro aproximado de USD $1400 al mes en costos de cómputo, una cifra altamente significativa para la escala del sistema. El impacto de esta optimización empezó a notarse al segundo mes de operación del clúster, una vez que las máquinas legadas se pudieron apagar definitivamente, consolidando el nuevo esquema de gastos reducidos. Desde una perspectiva de escalabilidad, la infraestructura Kubernetes demostró ofrecer una flexibilidad muy superior. Ahora es posible escalar horizontalmente los servicios de forma cuando aumenta la carga de trabajo, algo que antes no era posible.

Otro logro notable fue el incremento de la resiliencia y confiabilidad de la plataforma. La nueva arquitectura eliminó varios puntos únicos de falla y redujo la dependencia en servidores específicos. Gracias al despliegue en múltiples zonas de disponibilidad y al uso de contenedores replicados, el sistema adquirió una alta tolerancia a fallos: ante la eventual caída de un nodo o pod, Kubernetes reubica automáticamente las cargas en otras partes del clúster, asegurando que el servicio siga disponible. Los clientes ahora disponen de un producto con alta disponibilidad, capaz de seguir operando incluso si ocurren incidentes en la infraestructura subyacente. Esto mejora la experiencia de los usuarios finales y refuerza la confianza en el servicio ofrecido, ya que se minimizan las interrupciones no planificadas. Adicionalmente, la incorporación de herramientas de observabilidad y monitoreo robustas (como Prometheus y Grafana para métricas, Loki para log centralizado, entre otras) brinda a los equipos de TI una visión en tiempo real del estado del sistema y sus componentes. Con estas tecnologías, es posible detectar y diagnosticar problemas de manera proactiva, muchas veces antes de que escalen a incidencias mayores, lo que se traduce en tiempos de respuesta más rápidos y en última instancia en una operación más estable.
Asimismo, las herramientas introducidas mejoraron la calidad de vida del desarrollador: gracias a Docker y a la uniformidad entre ambientes, desaparecieron muchos problemas de la categoría "funciona en mi máquina"; con un monitoreo integrado, los errores en producción son más fáciles de depurar; y con la infraestructura gestionada como código, la colaboración entre desarrolladores e ingenieros de operación es más fluida, pues ambos comparten visibilidad sobre la configuración del sistema. Todos estos factores han contribuido a agilizar el ritmo de entrega de valor al cliente. Como indicaron las métricas comparativas de los sprints, la velocidad de desarrollo aumentó y la dedicación a incidentes de infraestructura disminuyó significativamente. En resumen, la transformación digital emprendida no solo mejoró la tecnología, sino también la forma de trabajar del equipo, volviéndolo más productivo y permitiendo un enfoque mayor en la innovación y menos en el mantenimiento reactivo.

Al cierre de este proyecto, más allá de las mejoras técnicas y beneficios, es pertinente reflexionar sobre el aprendizaje adquirido y su impacto en el ámbito profesional. La

realización de esta iniciativa significó una experiencia enriquecedora en términos de infraestructura moderna y DevOps, la migración desde una plataforma tradicional hacia Kubernetes brindó la oportunidad de trabajar de primera mano con conceptos avanzados de arquitectura de software que trascienden la teoría, llevando a la práctica principios como la infraestructura como código, el escalamiento automático, la observabilidad integral y la automatización de despliegues. Diseñar y levantar un clúster productivo en AWS, integrando múltiples servicios de nube y herramientas *open-source*, permitió consolidar habilidades técnicas de alto valor. Por otro lado, se logró fortalecer las competencias en administración de contenedores, configuración de entornos *cloud*, orquestación con Kubernetes y monitoreo de sistemas distribuidos, entre otras. Se comprobó empíricamente cómo cada decisión de arquitectura puede influir en la robustez y eficiencia global de la plataforma, lo cual mejoró la capacidad de análisis crítico y atención al detalle del equipo técnico.

En el ámbito de la cultura DevOps se Implementaron herramientas como Flux CD bajo el enfoque GitOps para simplificar y automatizar los despliegues, pipelines CI/CD eficientes que reemplazaron los scripts poco confiables, un *stack* de monitorio y observabilidad compuesto por Grafana, Prometheus y Loki brindando al equipo información valiosa en tiempo real sobre el rendimiento del sistema. Esta visibilidad mejorada permitió al equipo identificar y resolver problemas de manera proactiva antes de que impactaran al usuario final. Estas implementaciones técnicas transformaron la forma de trabajo del equipo al habilitar despliegues más confiables y ágiles.

Por último, cabe resaltar las lecciones en gestión del cambio y liderazgo técnico que dejó el proyecto, dando como resultado un equipo que confía en los procesos y flujos del software que mantienen. De hecho, luego de la primera migración de smi-backend, el equipo decidió replicar el enfoque en el resto de las aplicaciones de la empresa, migrando también *tar-backend* y *sync-backend* al nuevo clúster. Esto refleja cómo el proyecto sentó un precedente positivo y un modelo a seguir para futuras iniciativas dentro de la organización.

En conclusión, la migración desde una infraestructura no escalable hacia Kubernetes, acompañada de la adopción de la cultura DevOps, cumplió con los objetivos propuestos y entregó resultados palpables en términos de costos, rendimiento y confiabilidad.

REFERENCIAS BIBLIOGRÁFICAS

[1] S. Blank, «Steve Blank,» 25 Enero 2010. [En línea]. Available: https://steveblank.com/2010/01/25/whats-a-startup-first-principles/.

[2] J. Garrison y K. Nova, Cloud Native Infrastructure Patterns for Scalable Infrastructure and Applications in a Dynamic Environment, Sebastopol: O'Reilly Media, Inc.,, 2018.

[3] Celery, «Celery - Distributed Task Queue,» 2023. [En línea]. Available: https://docs.celeryq.dev/en/stable/.

[4] supervisord, «Changelog — Supervisor 4.2.5 documentation,» 2022. [En línea]. Available: https://supervisord.org/changes.html.

[5] Amazon, «Memcached,» 2024. [En línea]. Available: https://aws.amazon.com/es/memcached/.

[6] W. Siddiqui, «Introduction to Sentry: An overview of the platform, its features, and how it can be used to monitor and debug applications.,» 26 4 2023. [En línea]. Available: https://medium.com/@waqas.siddiqui619/introduction-to-sentry-io-5456dfa1b3a4.

[7] S. Ulili, «Better Stack,» A Complete Guide to Monitoring With Uptime Kuma , 25 9 2024. [En línea]. Available: https://betterstack.com/community/guides/monitoring/uptime-kuma-guide/.

[8] Tom Hall, Atlassian, «What is DevOps culture?,» 2025. [En línea]. Available: https://www.atlassian.com/devops/what-is-devops/devops-culture.

[9] Atlassian, «Devops,» 2025. [En línea]. Available: https://www.atlassian.com/devops.

[10] Docker Inc., «What is a Container?,» Docker, 2025. [En línea]. Available: https://www.docker.com/resources/what-container/.

[11] A. Verma, L. Pedrosa, K. Madhukar, D. Oppenheimer, E. Tune y J. Wilkes, «Large-scale cluster management at Google with Borg,» p. 18, 2015.

[12] Kubernetes, «¿Qué es Kubernetes?,» 2025. [En línea]. Available: https://kubernetes.io/es/docs/concepts/overview/what-is-kubernetes/.

[13] Kubernetes, «Documentacion oficial,» 2025. [En línea]. Available: https://kubernetes.io/docs/home/.

[14] Amazon, «¿Cuál es la diferencia entre la observabilidad y el monitoreo?,» 2025. [En línea]. Available: https://aws.amazon.com/es/compare/the-difference-between-monitoring-and-observability/.

[15] Grafana, «Grafana documentation,» 2025. [En línea]. Available: https://grafana.com/docs/grafana/latest/?pg=oss-graf&plcmt=hero-btn-2#learn.

[16] Grafana, «Loki overview,» 2025. [En línea]. Available: https://grafana.com/docs/loki/latest/get-started/overview/.

[17] Prometheus, «Overview,» 2025. [En línea]. Available: https://prometheus.io/docs/introduction/overview/.

[18] Flux, «Core concepts,» 2025. [En línea]. Available: https://fluxcd.io/flux/concepts/#gitops.

[19] Flux, «Flux Documentation,» 2025. [En línea]. Available: https://fluxcd.io/flux/.

[20] Amazon Web Services, «What is AWS,» [En línea]. Available: https://aws.amazon.com/es/what-is-aws/?nc1=f_cc.

[21] Amazon, «What is Amazon EC2?,» 2025. [En línea]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html.

[22] Amazon, «What is Amazon VPC?,» [En línea]. Available: https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html.

[23] yaml, «YAML Ain’t Markup Language (YAML™) version 1.2,» 01 10 2021. [En línea]. Available: https://yaml.org/spec/1.2.2/.

[24] The cert-manager Authors, «cert-manager,» 2025. [En línea]. Available: https://cert-manager.io/docs/.

[25] Kubernetes, «How it works,» 2025. [En línea]. Available: https://kubernetes.github.io/ingress-nginx/how-it-works/.

[26] Bitnami-labs, «Sealed Secrets for Kubernetes,» [En línea]. Available: https://github.com/bitnami-labs/sealed-secrets/blob/main/README.md.

[27] Stakater, «Introduction,» 2023. [En línea]. Available: https://docs.stakater.com/reloader/index.html.

[28] Docker Inc., «Docker Compose,» 2025, [En línea]. Available: https://docs.docker.com/compose/.

[29] Redis, «Open Source,» 2025. [En línea]. Available: https://redis.io/docs/latest/get-started/.

[30] Kubernetes, «Workloads,» 2023. [En línea]. Available: https://kubernetes.io/docs/concepts/workloads/.

ANEXOS

Anexo A

1. Crear un security group, nombrándolo ec2-redis, permitiendo el acceso solo desde los rangos de IP de las instancias EC2.
2. Configurar un *subnet group*, para asociarlo a todas las *availability zones* que se requieren.
3. Detener el *beat* y *worker*, para evitar encolar y ejecutar las Tasks presentes en el momento de la migración.

```
1 sudo supervisorctl stop <client>.enerlink.cl_worker <client>.enerlink.cl_beat
```

Figura 41: Comando necesario para detener los servicios durante la migración. Fuente: Elaboración propia.

4. Conectarse al servicio de Redis y crear un *dump* con los datos presentes.

```
redis /var/run/redis/redis-server.sock[1]> save
OK
```

Figura 42: Comando para obtener un Dump de las tareas encoladas en Redis. Fuente: Elaboración propia.

5. Conectarse con SFTP desde la maquina pivote a la maquina en la que se realizara la migración, para así poder obtener el dump (No es posible desde la misma máquina, ya que están configuradas para evitar obtener cualquier recurso desde ellas).
6. Configurar un *bucket* de S3 para que pueda ser leído por ElastiCache.

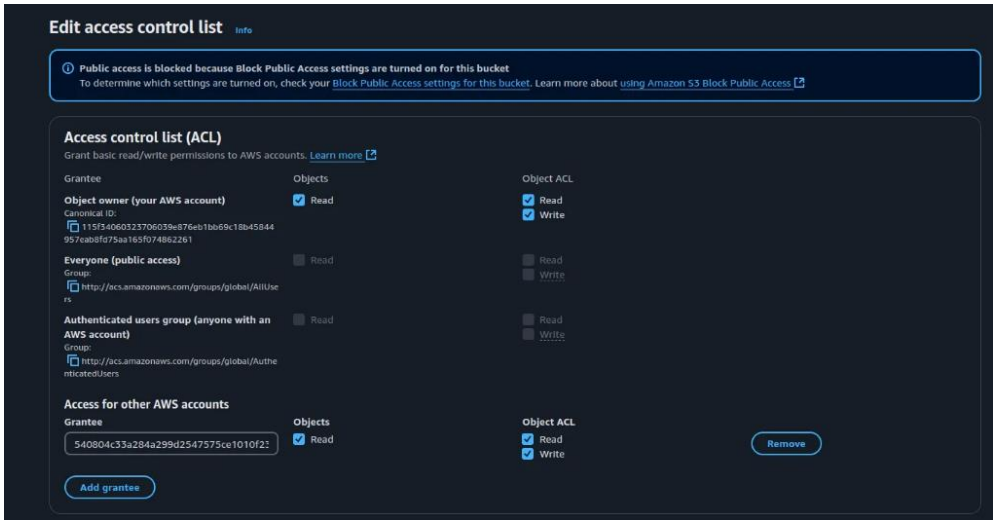

Figura 43: Configuraciones necesarias para preparar el *bucket* S3 para recibir los *dumps*.

7. Subir el *dump* al bucket configurado.
8. Crear la instancia de Redis desde ElastiCache, el *engine version*[14] debe ser 7.1 (compatible con Redis OSS 7.0.1) y el tipo de la maquina *cache.t3.micro* sin excepciones. El nombre de las instancias debe seguir el formato smi-<nombre-cliente>

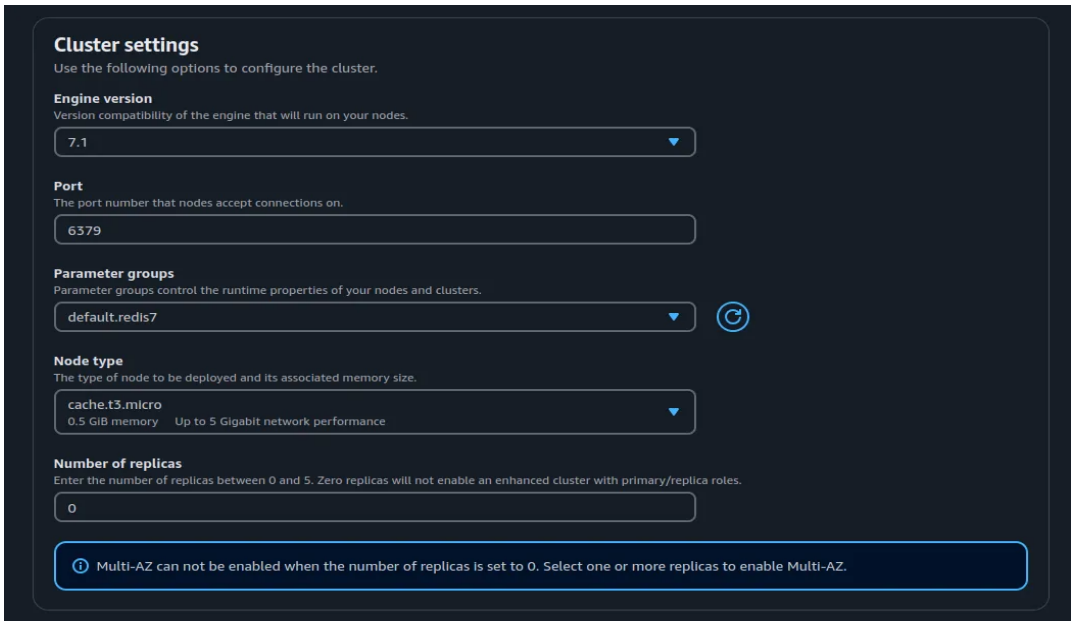

Figura 44: Algunas configuraciones para las instancias de ElastiCache.

[14] El *engine version* es la base del sistema ElastiCache, no la versión de Redis que soporta.

9. Conectarse a la instancia EC2 que se está migrando y modificar el copyConf.conf para agregar las nuevas configuraciones y reiniciar.

Con Redis en ElastiCache se abre el camino para poder comenzar con las configuraciones del clúster.

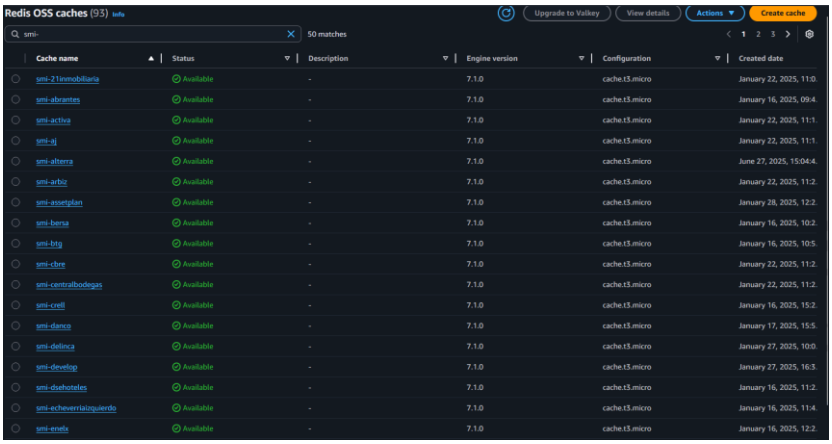

Figura 45: Instancias Redis ya en producción para todos los tenants en ElastiCache. Fuente: Elaboración propia.

Anexo B

Con un Dockerfile ya creado se puede desarrollar el docker-compose.yml. A continuación, se presenta la configuración para el *backend* y luego se procederá con las explicaciones.

```yaml
services:
  smi-django: &django
    build:
      context: .
      dockerfile: ./docker/Dockerfile
      target: development
    container_name: smi-django
    volumes:
      - .:/smi-backend
    env_file:
      - .env
```

Figura 46: Servicio SMI-backend en docker-compose.yaml. Fuente: Elaboración propia.

Con esto Docker compose leerá el Dockerfile desarrollado y construirá una imagen para el ambiente de desarrollo, una vez construida la imagen automáticamente se ejecutará el contenedor. Para ambientes de desarrollo el campo, *volumes* es muy útil ya que permite compartir un directorio de la maquina local con un directorio propio del contenedor, de esta forma cada cambio que hagamos también se verá reflejado en el contenedor.

Con esto ya tenemos nuestro backend listo para ejecutar, pero hay que agregar el resto de los servicios. Gracias a que es un archivo tipo YAML, podemos heredar la declaración del *backend* y solo sobrescribir el comando a ejecutar para tener el *worker* y el *beat*.

```
1   smi-worker-celery:
2     <<: *django
3     container_name: smi-celery
4     command: 'celery -A root.celeryapp worker -l INFO -c 2 -n worker_celery'
5   smi-beat-celery:
6     <<: *django
7     container_name: smi-beat
8     command: 'celery -A root.celeryapp beat -l INFO'
```

Figura 47: Declaración del *worker* y *beat* en el archivo docker-compose.yaml. Fuente: Elaboración propia.

Para los servicios como la base de datos y el Redis, se deben buscar las imágenes oficiales en Docker Hub[15] y se agregan al docker-compose.yml, por ejemplo, para *PostgreSQL* se hace lo siguiente:

```
1   ...
2   smi-postgres:
3     image: postgres:15.7-alpine
4     container_name: smi-postgres
5     env_file:
6       - .env
7     volumes:
8       - smi-backend_local_postgres_data:/var/lib/postgresql/data
9     environment:
10      - POSTGRES_USER=${DB_USER}
11      - POSTGRES_PASSWORD=${DB_PASS}
12      - POSTGRES_DB=${DB_NAME}
13  ...
```

Figura 48: Configuración de base de datos PostgreSQL de desarrollo. Fuente: Elaboración propia.

[15] Docker Hub es el lugar dónde regularmente se suben las imágenes de todos los sistemas open source.

Finalmente, es importante poder mantener nuestros contenedores en una misma red interna de docker para que estos puedan comunicarse de manera correcta, la simplicidad de docker compose nos permite resolver este problema de red en 3 líneas que se agregan en el docker-compose.yml.

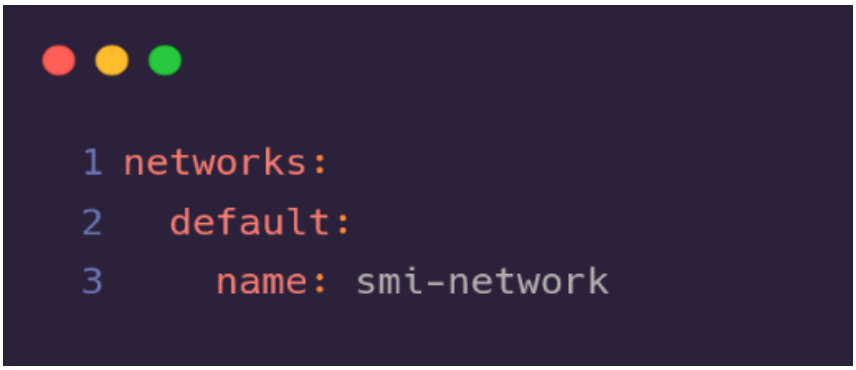

Figura 49: Declaración de la red en el *docker-compose.yaml* de *smi-backend*. Fuente: Elaboración propia.

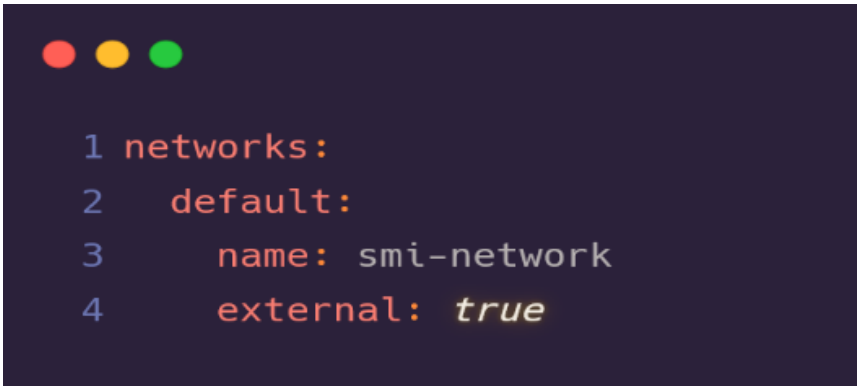

Figura 50: Declaración de la red en *docker-compose.yaml* de *tar-backend* como externa. Fuente: Elaboración propia.

Cada *Docker-compose.yaml* crea de forma automática su propia red interna, por eso es importante gestionarla y debe ser configurada también en tar-backend para que Docker entienda que también puede acceder a esos servicios a través de la red.

Anexo C

Instancia	CPU virtual*	Créditos por hora de CPU	Memoria (GiB)	Almacenamiento	Rendimiento de red
t2.nano	1	3	0,5	Solo EBS	Bajo
t2.micro	1	6	1	Solo EBS	De bajo a moderado
t2.small	1	12	2	Solo EBS	De bajo a moderado
t2.medium	2	24	4	Solo EBS	De bajo a moderado
t2.large	2	36	8	Solo EBS	De bajo a moderado
t2.xlarge	4	54	16	Solo EBS	Moderado
t2.2xlarge	8	81	32	Solo EBS	Moderado

Figura 51: Especificaciones de la familia T2