UNIVERSIDAD NACIONAL DE LA PLATA

TESINA DE LICENCIATURA

Programa de apoyo para alumnos con experiencia profesional

TÍTULO: Migración de una arquitectura basada en contenedores usando ECS a Kubernetes
AUTOR: Daniel Alejandro Cesanelli
DIRECTOR ACADÉMICO: Christian Adrián Rodriguez
DIRECTOR PROFESIONAL: Leandro Montedoro
CODIRECTOR ACADÉMICO: Miguel Luengo
CARRERA: Licenciatura en Informática

Resumen

El objetivo de esta propuesta es realizar la migración de un servicio de ejecución de contenedores a Kubernetes, por sus ventajas respecto de ofrecer un entorno más versátil, con un mayor control por parte del administrador, lo que permite maximizar la utilización de los recursos, al mejorar el control de gastos y escalar soluciones de una forma adecuada a las actualmente disponibles.
En este trabajo se analizarán los desafíos para afrontar la migración, buscando mantener el servicio disponible y utilizando infraestructura mediante código para minimizar los problemas generalmente relacionados con errores humanos.

Palabras Clave

AWS, Kubernetes, k8s, ECS, EKS, Docker, Cloud Computing, CaaS.

Conclusiones

Se demuestra que es posible realizar una migración de servicios de una plataforma de contenedores, como es el caso de ECS a kubernetes, sin necesidad de modificar las imágenes de los contenedores, maximizando la reutilización del trabajo previamente realizado en tanto a la construcción de imágenes de contenedores.

Trabajos Realizados

Se realizó la migración de una plataforma, compuesta por varios servicios y aplicaciones, corriendo en un servicio de contenedores en AWS ECS, a una plataforma Kubernetes, utilizando el servicio de AWS EKS, investigando exhaustivamente Kubernetes y herramientas para su gestión, así como el despliegue de aplicaciones.

Trabajos Futuros

Algunos trabajos futuros que se desprende de esta tesina son:

- *Kubernetes se encuentra constantemente evolucionando, por lo que es fundamental mantener actualizados los ambientes productivos.*
- *incorporación de trazas como parte de la observabilidad*
- *seguir investigando mejoras en temas de seguridad*
- *analizar alternativas serverless para maximizar ahorro de costos*

Fecha de la presentación: Febrero 2023

1. Introducción

1.1. Objetivo

El objetivo de esta propuesta es realizar la migración de un servicio de ejecución de contenedores a Kubernetes, por sus ventajas respecto de ofrecer un entorno más versátil, con un mayor control por parte del administrador, lo que permite maximizar la utilización de los recursos, al mejorar el control de gastos y escalar soluciones de una forma adecuada a las actualmente disponibles.

En este trabajo se analizarán los desafíos para afrontar la migración, buscando mantener el servicio disponible y utilizando infraestructura mediante código para minimizar los problemas generalmente relacionados con errores humanos.

1.2. Motivación

Quien elabora estos planteos trabaja para una empresa que presta servicios para Corning Incorporated. Corning Inc. es una de las principales empresas innovadoras del mundo en cuanto a las ciencias de los materiales, vidrio de especialidad, cerámica y física óptica, para desarrollar productos, principalmente para aplicaciones industriales y científicas.

En un proyecto realizado para esta empresa, se ha priorizado facilitar el armado y manejo de recursos de la infraestructura donde desplegar una serie de servicios. Para ello se utilizó un servicio de Amazon Web Services (AWS, 2022) llamado Amazon Elastic Container Service (Amazon ECS), que ejecuta servicios en forma de tareas compuestas por contenedores. Este realiza la gestión de las cargas de trabajo automáticamente, sin necesidad de un control constante de los recursos, pero a un costo mayor.

Con la evolución del proyecto han aparecido nuevos servicios y con ello la necesidad de mejorar el monitoreo, escalado automático, autohealing, facilitar

el despliegue de los mismos de forma ordenada, lo que permite incluso despliegues de tipo canary, entre otros.

Ante esta nueva realidad, se analiza la posibilidad de utilizar Kubernetes, en el mismo proveedor de cloud, con Amazon Elastic Kubernetes Service (Amazon EKS), que provee la creación de un clúster autogestionado de Kubernetes, lo que posibilita un manejo de los recursos más apropiado para la etapa actual del proyecto.

1.3. Problemas existentes

Con Amazon ECS los despliegues de una aplicación son relativamente sencillos, sobre todo si se programan desde terraform. Sin embargo, las complicaciones comenzaron a visibilizarse con el agregado de nuevos servicios. Ya la infraestructura de base era grande, y considerando además el escalamiento de algunos servicios, el aprovechamiento de recursos no se hacía evidente, lo que ocasionaba un coste mayor.

Por otro lado, al tener los servicios corriendo por separado, comenzó a ser más difícil la tarea de monitorizarlos, así como los recursos necesarios para su correcto desempeño y escalado.

En un determinado momento el proyecto posee un crecimiento grande, yendo de una aplicación para el usuario a tres aplicaciones separadas.Así, de poseer tres servicios pasamos a correr quince , todo esto aplicado en tres ambientes (de desarrollo, testeo y producción), lo que genera un crecimiento muy elevado.

1.4. Mejoras esperadas

Al finalizar este trabajo, se espera poder contar con un análisis de opciones para abordar el cambio propuesto, su diseño e implementación de la infraestructura necesaria para reemplazar el proyecto actual. Como resultado, se tendrá un cluster Kubernetes, desplegado con Infraestructura como código, que a su vez provea un procedimiento de despliegue de todos los insumos

necesarios, usando infraestructura como datos. Además, se establecerán procedimientos de despliegue continuo, combinados con integración y despliegue continuo, sin perder de vista la seguridad, la salud de los servicios, backups y escalabilidad de la solución.

1.5. Metodología del desarrollo

Esta propuesta se desarrolla en el ámbito de trabajo, ITX Corp, que brinda servicios a la compañía Corning Inc.. En el proyecto se encuentran involucradas más de 20 personas, y se trabaja con metodologías ágiles (Schwaber, 2004), un proceso iterativo e incremental de mejora continua. Se aplica Scrum (Schwaber, 2022), con Sprints de 2 semanas durante las cuales se planifica y prioriza qué actividades del backlog se implementarán. Cada entrega debe ser testeada teniendo nuevas funcionalidades usables en cada iteración.

Para este proyecto en particular se realizarán los análisis e implementación en paralelo al desarrollo de otras funcionalidades, mostrando el progreso en un ambiente de pruebas separado, para no interrumpir las entregas de dichas funcionalidades.

1.6. Organización de la tesina

Capítulo 1: Introducción. En este capítulo se plantean, el objetivo, motivación, problemas existentes, mejoras esperadas, metodologías de desarrollo y la organización de la tesina.

Capítulo 2: Análisis y marco de referencia. Presenta una introducción a la computación en la nube, cómo fueron sus orígenes y cómo fue evolucionando. Se enumeran todos los conceptos y tecnologías relacionadas que servirán como base, para los contenidos de esta tesina.

Capítulo 3: Análisis de la solución actual. En este capítulo se describe el sistema en el que se basa este trabajo, su complejidad, cómo se ha construído y sus beneficios y problemas.

Capítulo 4: Análisis de la alternativa. Se muestran las nuevas tecnologías, sus características y ventajas sobre la arquitectura implementada inicialmente.

Capítulo 5: Solución final. Se especifican los pasos realizados para llevar adelante los cambios propuestos, cómo se resolvieron los problemas que fueron surgiendo y cómo quedó implementada la solución finalmente.

Capítulo 6: Resultados obtenidos. Se encuentran los resultados obtenidos, mostrando métricas que los validan.

Capítulo 7: Conclusiones. Este capítulo presenta las conclusiones obtenidas con este trabajo y futuros cambios y mejoras posibles.

2. Análisis y marco de referencia

En este capítulo introduciremos conceptos fundamentales para el análisis y realización de esta propuesta. Comenzaremos con el concepto de Cloud Computing, que da origen a todas las tecnologías aquí utilizadas, y continuaremos con las tecnologías involucradas hasta llegar a Kubernetes.

2.1. Historia

Durante la década del ‘80 se crean las primeras computadoras personales, generando más opciones, y bajando su costo considerablemente, haciendo más popular su uso en hogares y empresas. (White, 2001)

Con la década del ‘90 surgen los SO más amigables como Windows y OS/2, y crece fuertemente el desarrollo de aplicaciones para estos sistemas. Eran sistemas monolíticos, desarrollados en lenguajes como Visual Basic, Delphi, e incluso en sistemas de gestión de base de datos como dBase y FoxPro. Estos lenguajes proveían librerías y una base sobre la cual se podía desarrollar, haciendo más simple tener un producto final. Durante el comienzo de esta década las conexiones a internet llegan a los hogares, y se hacen más populares, aunque no aún con alta disponibilidad, dado que las conexiones eran dial up, y acarreaban un alto costo telefónico.

En 1995 Netscape introduce Javascript, y el programador Reasmus Lerdorf pone a disposición el lenguaje PHP, con lo que el desarrollo de aplicaciones web comienza a ser mucho más común.

En el mismo año James Gosling, de Sun Microsystems, desarrolla el lenguaje de programación Java y lo publica como parte fundamental de la plataforma Java de Sun Microsystems. Su sintaxis deriva del C y C++ y sus aplicaciones compilan a un *bytecode*, que se puede ejecutar en cualquier máquina virtual de Java, las cuales se implementan para cada una de las arquitecturas y sistemas operativos, haciendo el código 100% portable.

Durante los siguientes años se desarrollan y toman importancia nuevas arquitecturas, como N-tier, SOA y Microservicios, que proveen distintas formas de implementar servicios, orientadas al desarrollo más distribuido.

En el año 2000 se comienza a comercializar el ADSL, y la internet por cable, que permite mejores velocidades de comunicación y una conexión constante, pagando una tarifa plana y permitiendo estar conectado 24h al día, con lo que se comienza a naturalizar el uso de sistemas basados en la web, alojados en servidores online.

2.2. Cloud Computing

A principios de la década del ‘90, para tener un servicio disponible en internet, las empresas debían comprar servidores físicos y alojarlos en datacenters propios o alquilados. Estos servidores además de ser costosos de comprar, eran complejos de mantener y a través de los años se volvían obsoletos y requerían de costosas actualizaciones. (Neto, 2014)

A fines de la década del ‘90 comienzan a aparecer servicios de nube, estos proveen de servidores dedicados o virtuales, alojados en la nube, y alquilados por un costo por hora/mes/año, que hace sea mucho más simple de manejar. Este servicio se puede cancelar en cualquier momento o incrementar en cantidad o tamaño, sin necesidad de lidiar con los mantenimientos físicos ni de software de dichos servidores.

Con el crecimiento de esta tecnología, se hace necesario virtualizar, no sólo las unidades computacionales, sino también routers, redes y subredes, almacenamiento, aplicaciones y servicios.

En 2002, Amazon lanzó al mercado AWS (Amazon Web Services) (AWS Cloud Computing, 2022), una plataforma que ofrecía almacenamiento y recursos de computación en la nube. Nacía la primera plataforma de cloud pública (Google Cloud, 2022), que hoy lidera el mercado, y más tarde surgieron Google Cloud

Platform, Microsoft Azure y la versión china de estas iniciativas, Alibaba Cloud Computing.

También marcó el desarrollo de cloud que Microsoft y Google ofrecieran sus aplicaciones de productividad bajo este modelo y con el tiempo estos servicios se fueron haciendo más accesibles al punto que en la actualidad cualquier persona puede solicitarlos a un precio relativamente bajo y en forma rápida y simple.

En 2011 el Instituto Nacional de Standards y Tecnologías de Estados Unidos (National Institute of Standards and Technology's - NIST) (NIST, 2011), define Cloud Computing de la siguiente forma:

> *"Cloud Computing es un modelo tecnológico que permite el acceso ubicuo, adaptado y bajo demanda en red a un conjunto compartido de recursos de computación configurables compartidos (por ejemplo: redes, servidores, equipos de almacenamiento, aplicaciones y servicios), que pueden ser rápidamente provisionados y liberados con un esfuerzo de gestión reducido o interacción mínima con el proveedor del servicio.".*

Este modelo está compuesto de 5 características esenciales, 3 modelos de servicios y 4 modelos de despliegue.

Características esenciales:

- **Autoservicio bajo demanda:** el consumidor puede aprovisionar de forma flexible y automática, escalando según se requiera, de todos los servicios computacionales.

- **Amplio acceso a la red**: Las capacidades están disponibles a través de la red y se accede a ellas a través de mecanismos estándares que promueven el uso de plataformas heterogéneas de clientes (por ejemplo, teléfonos móviles, tabletas, computadoras portátiles y estaciones de trabajo).

- **Puesta en común de recursos**: Los recursos informáticos del proveedor se agrupan para servir a múltiples consumidores utilizando un modelo de múltiples consumidores, con diferentes recursos físicos y virtuales asignados y reasignados dinámicamente de acuerdo con la demanda del consumidor. Los ejemplos de recursos incluyen almacenamiento, procesamiento, memoria y ancho de banda de red.

- **Agilidad en la escalabilidad**: Los recursos se pueden aprovisionar y liberar de manera elástica, en algunos casos automáticamente, para escalar rápidamente, creciendo o decreciendo, de acuerdo con la demanda. Para el consumidor, los recursos disponibles para el aprovisionamiento a menudo parecen ilimitados y pueden apropiarse en cualquier cantidad y en cualquier momento.

- **Servicio medido**: Dado que el servicio se paga por uso, los sistemas en la nube controlan y optimizan automáticamente el uso de recursos aprovechando una capacidad de medición en algún nivel de abstracción apropiado para el tipo de servicio. El uso de recursos se puede monitorear, controlar e informar, lo que brinda transparencia tanto para el proveedor como para el consumidor del servicio utilizado.

Modelos de despliegue:

- **Cloud privada**: la infraestructura se aprovisiona para uso exclusivo de una sola organización, que comprende varios consumidores (por ej. unidades de negocio). Puede ser propiedad de la organización, un tercero o una combinación de ellos, administrarlo y operarlo, y puede existir dentro o fuera de las instalaciones. Un ejemplo de este tipo de cloud es AFIP, que hace uso de openstack.

- **Cloud comunitaria**: La infraestructura de la nube está aprovisionada para uso exclusivo de una comunidad específica de consumidores de organizaciones que tienen preocupaciones compartidas (por ej., misión, requisitos de seguridad, políticas y consideraciones de cumplimiento). Puede ser propiedad de una o más

de las organizaciones de la comunidad, un tercero o una combinación de ellos, administrarlo y operarlo, y puede existir dentro o fuera de las instalaciones. Este tipo de nube es usada por ejemplo, por Google Sites.

- **Cloud pública**: La infraestructura de la nube está aprovisionada para uso abierto por parte del público en general. Puede ser propiedad de, administrada y operada por una organización comercial, académica o gubernamental, o alguna combinación de ellos. Existe en las instalaciones del proveedor de la nube. Ejemplos de este tipo de nube son Amazon Web Services, Google Cloud Platform y Azure.

- **Cloud híbrida**: La infraestructura de nube es una composición de dos o más infraestructuras de nube distintas (privada, comunitaria o pública) que siguen siendo entidades únicas, pero están unidas por tecnología patentada o estandarizada que permite la portabilidad de datos y aplicaciones.

Modelos de servicios:

- **Infrastructure as a Service (IaaS)**: En Infrastructure as a Service (Infraestructura como servicio) se provee la capacidad de aprovisionar servidores, almacenamiento, redes y otros recursos computacionales fundamentales, que permitirán al consumidor, correr software arbitrario, sin restricciones sobre los lenguajes y tecnologías usadas.

- **Platform as a Service (PaaS)**: En Platform as a Service (Plataforma como servicio) se elimina la necesidad de manejar la infraestructura de bajo nivel (hardware y sistemas operativos), y permite enfocarse en el despliegue y manejo de las aplicaciones, usando lenguajes, librerías, servicios y herramientas soportadas por el proveedor.

- **Software as a Service (SaaS)**: En Software as a Service (Software como servicio) se provee un software específico, ya instalado y configurado, listo para usar. Al igual que en Cloud Computing, el usuario no es responsable de ejecutar actualizaciones o de verificar que el hardware ni el sistema

operativo sean adecuados para utilizarlos. Dependiendo del software se puede acceder con un simple navegador o con un cliente específico en caso de bases de datos o software propietarios.

De estos 3 modelos se pueden generar otras especificaciones como CaaS (Container as a Service), FaaS (Functions as a Service), DaaS (Desktop as a Service), XaaS (Anything as a Service), aunque prácticamente todos los servicios ya tienen su equivalencia en la nube:

- Analytics as a Service
- Backup as a Service
- Business as a Service
- Communications as a Service
- Content as a Service
- Logging as a Service
- Monitoring as a Service
- Network as a Service
- Payments as a Service
- Robot as a Service
- Search as a Service
- Security as a Service
- Storage as a Service

Todas subconjuntos de uno o más de los modelos definidos por NIST.

2.3. Contenedores

La contenerización se refiere al proceso por el cual se encapsula una aplicación con todos los recursos necesarios para su ejecución. El resultado es una imagen de un contenedor, esto es un archivo que empaqueta dentro un filesystem con todos los archivos necesarios. Si esto se hace de forma correcta, tendremos un paquete reproducible, óptimo en cuanto a tamaño y velocidad de ejecución, que puede ejecutarse en cualquier ambiente con resultados predecibles.

Inicialmente se implementa Linux Containers (RedHat, 2022), haciendo uso de los namespaces y control groups para que, utilizando el propio kernel del sistema operativo que se encuentra corriendo, limitar los accesos a otros procesos, memoria y archivos, y permitir que los procesos corran en forma aislada y segura.

Docker (Docker, 2022) toma esta base y evoluciona usando diversas técnicas de virtualización para permitir correr contenedores en cualquier computadora de escritorio, en cualquier sistema operativo, como Windows o MacOS.

Estos contenedores se pueden correr en cualquier ambiente con diversas técnicas de virtualización, y son el método utilizado para distribuir el software en la nube.

Como RedHat indica en su whitepaper *Principles of container-based application design* (RedHat, 2017), en estos días es posible contenedorizar casi cualquier aplicación, pero hacerlo de forma tal que pueda ser desplegada de forma automatizada y orquestada efectivamente por una plataforma cloud-native como Kubernetes requiere un esfuerzo adicional.

Podemos enumerar una serie de principios que permitirán que los contenedores se comporten correctamente en los motores de orquestación de contenedores, lo que les permitirá programarlos, escalarlos y monitorearlos de manera automatizada

Single Concern Principle (SCP - Principio de Preocupación Única):

Este principio aconseja que una clase debe tener una sola responsabilidad. La motivación detrás de esto es que cada responsabilidad es un eje de cambio y una clase debe tener una, y solo una, razón para cambiar. La motivación principal para SCP es la reutilización y la capacidad de reemplazo de la imagen del contenedor. Si crea un contenedor que aborde una sola inquietud, y lo hace con todas las funciones, las posibilidades de que la imagen del contenedor se reutilice en diferentes contextos de aplicación son mayores. El principio SCP dicta que cada contenedor debe abordar una sola preocupación y hacerlo bien.

Si su microservicio en contenedores necesita abordar múltiples inquietudes, puede usar patrones como sidecar e init-containers para combinar varios contenedores en una sola unidad de implementación (pod), donde cada contenedor aún maneja una sola inquietud.

High Observability Principle (HOP - Principio de Alta Observabilidad):

Los contenedores proporcionan una forma unificada de empaquetar y ejecutar aplicaciones al tratarlos como una caja negra, pero debe proporcionar interfaces de programación de aplicaciones (API) para que el entorno de tiempo de ejecución observe el estado del contenedor y actúe en consecuencia. En términos prácticos, como mínimo, se debe proporcionar API para los diferentes tipos de controles de estado: actividad (liveness prove, que indica el correcto funcionamiento del servicio) y preparación (readiness probe, que indica el momento a partir del cual se puede prestar el servicio). La aplicación además debe registrar eventos importantes (logs) en la salida de error estándar (stderr) y la salida estándar (stdout) para la agregación de registros posterior con herramientas como Fluentd y Logstash (Udovicic, 2022).

Life-Cycle Conformance Principle (LCP - Principio de Conformidad del Ciclo de Vida):

La aplicación debe tener una forma de leer las señales (Kerrisk, 2022) que provienen de la plataforma, y adaptarse y reaccionar ante ellas. Hay todo tipo

de señales provenientes de la plataforma de administración que tienen como objetivo ayudar a administrar el ciclo de vida de los contenedores. Dependerá de cada aplicación o servicio decidir qué señales manejar y si reaccionar o no a ellas. Por ejemplo, cualquier aplicación que requiera un proceso de apagado limpio debe captar la señal *terminar* (SIGTERM) y apagarse lo más rápido posible. Esto es para evitar el apagado forzado a través de una señal *kill* (SIGKILL) que sigue a un SIGTERM. De no respetarlo, se pueden generar procesos zombies (Litvin, 2022).

También hay posibilidad de alterar qué sucede ante algunos eventos, como es el caso de PostStart y PreStop, que permiten colaborar en la gestión del ciclo de vida de una aplicación. Por ejemplo, algunas aplicaciones necesitan configurar alguna cuestión antes de recibir solicitudes de servicio y otras necesitan liberar recursos antes de cerrarse limpiamente. Si esto no se contempla por el manejador de señales de nuestra aplicación, es posible hacerlo con estos callbacks escribiendo algún script.

Image Immutability Principle (IIP - Principio de Inmutabilidad de la Imagen)

Las aplicaciones en contenedores están destinadas a ser inmutables y, una vez creadas, no se espera que cambien entre diferentes entornos. Esto implica el uso de un medio externo para almacenar los datos de tiempo de ejecución y depender de configuraciones externalizadas que varían según los entornos, haciendo uso de variables de ambiente, configmaps o secrets, en lugar de crear o modificar contenedores por entorno. Cualquier cambio en la aplicación en contenedores debería resultar en la creación de una nueva imagen de contenedor y su reutilización en todos los entornos.

Process Disposability Principle (PDP - Principio de Disposibilidad del Proceso)

Una de las motivaciones principales para cambiar a aplicaciones en contenedores es que los contenedores deben ser lo más efímeros posible y estar listos para ser reemplazados por otra instancia de contenedor en

cualquier momento. Hay muchas razones para reemplazar un contenedor, como fallar en una verificación de estado (liveness probe), reducir la escala de una aplicación, migrar los contenedores a un host diferente, falta de recursos de la plataforma u otro problema.

Esto significa que las aplicaciones en contenedores deben mantener su estado (por ejemplo el mantenimiento de sesiones en aplicaciones HTTP) fuera del propio contenedor, usando algún mecanismo externo. También significa que la aplicación debe iniciarse y cerrarse rápidamente, e incluso estar lista para una falla repentina y completa del hardware.

Self-Containment Principle (S-CP - Principio De Autocontención)

Este principio dicta que un contenedor debe contener todo lo que necesita en el momento de la construcción. El contenedor debe depender únicamente de la presencia del kernel de Linux y tener bibliotecas adicionales agregadas en el momento en que se construye. Las únicas excepciones son elementos como las configuraciones, que varían entre diferentes entornos y deben proporcionarse en tiempo de ejecución; por ejemplo, a través de variables de ambiente, y además en el caso de Kubernetes mediante ConfigMaps y Secrets.

Runtime Confinement Principle (RCP - Principio de Confinamiento del Tiempo de Ejecución)

Los contenedores permiten dimensionar recursos tanto en uso de CPU como el uso de memoria. Este principio RCP sugiere que cada contenedor declare sus requisitos de recursos y pase esa información a la plataforma. Además de pasar los requisitos de recursos del contenedor, también es importante que la aplicación se limite a los indicados. Si la aplicación permanece confinada, es menos probable que la plataforma la considere para su finalización y migración cuando se produzca la escasez de recursos.

La mayoría de estos principios están basados en *The 12 Factors Apps*.

2.4. The 12 Factors Apps

"The 12 Factors Apps" (Wiggins, 2017) es una metodología creada por expertos de Heroku en el año 2011, luego de ver repetidos problemas para construir aplicaciones SaaS que:

- Usan formatos declarativos para la automatización de la configuración, para minimizar el tiempo y el coste que supone que nuevos desarrolladores se unan al proyecto;
- Tienen un contrato claro con el sistema operativo sobre el que trabajan, ofreciendo la máxima portabilidad entre los diferentes entornos de ejecución;
- Son apropiadas para desplegarse en modernas plataformas en la nube, obviando la necesidad de servidores y administración de sistemas;
- Minimizan las diferencias entre los entornos de desarrollo y producción, posibilitando un despliegue continuo para conseguir la máxima agilidad;
- Y pueden escalar sin cambios significativos para las herramientas, la arquitectura o las prácticas de desarrollo.

La metodología “twelve-factor” puede ser aplicada a aplicaciones escritas en cualquier lenguaje de programación, y cualquier combinación de ‘backing services’ (bases de datos, colas, memoria caché, etc).

Los 12 principios que deben seguirse son:

I. **Código base (Codebase)**: Un código base sobre el que hacer el control de versiones y múltiples despliegues

   Las aplicaciones deben gestionarse con un control de versiones, y cada aplicación debe ser creada con un único código base.

   Si más de una aplicación comparten parte de código, esta debe separarse en una librería externa y ser usada por ambas aplicaciones.

El mismo código va a usarse para realizar los despliegues en los diferentes ambientes (desarrollo, testeo y producción).

II. **Dependencias**: Declarar y aislar explícitamente las dependencias

Las aplicaciones no pueden depender de ningún paquete instalado en el sistema. Las mismas deben declarar e instalar todas sus dependencias, usando manejadores que permitan la instalación automática y aseguren la utilización de las mismas versiones de librerías a través del tiempo en los diferentes ambientes.

III. **Configuraciones**: Guardar la configuración en el entorno

Las configuraciones de entorno son lo único que debe variar entre los distintos despliegues. Estas configuraciones incluyen strings de conexión y claves de servicios y valores de despliegue de cada entorno. Deben manejarse con variables de entorno y no como archivos para facilitar los despliegues.

IV. **Backing services**: Tratar a los “backing services” como recursos conectables

En este punto lo fundamental es tratar a todos los servicios que se usen, como externos, de la forma más agnóstica posible, que sólo dependan de sus datos de conexión, que estarán disponibles al momento del despliegue. Esto permitirá cambiar de instancias de los servicios con sólo cambiar los datos de conexión, permitiendo, por ej, cambiar una base de datos corrupta, por una instancia nueva, o un MySQL manejado por nosotros por un Amazon RDS manejado por AWS sin ningún problema.

V. **Construir, desplegar, ejecutar**: Separar completamente la etapa de construcción de la etapa de ejecución

Se separan las 3 etapas, construcción, despliegue y ejecución de forma tal que:

- **Construcción**: se genera un código ejecutable a partir del código base. Cada construcción debe incluir un identificador para poder volver a una versión anterior en caso de ser necesario.
- **Despliegue**: se distribuye el ejecutable, y se une a las configuraciones del ambiente correspondiente, para que esté lista para ejecutarse inmediatamente.
- **Ejecución**: se lanzan los procesos con los ejecutables de una distribución determinada.

Esta separación permite tener un control de versiones de las distribuciones, despliegues independientes y que los procesos se lancen rápidamente.

VI. **Procesos**: Ejecutar la aplicación como uno o más procesos sin estado

Los procesos no deben mantener ningún tipo de estado, si esto fuera necesario se debería utilizar un servicio como una base de datos. Esto permite que la aplicación escale fácilmente, y que no importe qué proceso atienda una solicitud determinada.

VII. **Asignación de puertos**: Publicar servicios mediante asignación de puertos

Las aplicaciones deben ser autocontenidas, y no necesitar de un servicio externo para su utilización. Las mismas deben publicar un puerto por el que escucha las peticiones.

VIII. **Concurrencia**: Escalar mediante el modelo de procesos

Al utilizar el modelo de procesos, y no poseer estado, el escalamiento a través de diferentes servidores se maneja de forma transparente.

IX. **Desechabilidad**: Hacer el sistema más robusto intentando conseguir inicios rápidos y finalizaciones seguras

Los procesos de las aplicaciones pueden iniciar o finalizar en el momento que sea necesario, permitiendo un escalado rápido y flexible. Los procesos deben intentar minimizar los tiempos de arranque, proporcionando mayor agilidad en el proceso de distribución y escalado. También debe asegurarse que los procesos paran de forma segura ante la señal SIGTERM del manejador de procesos.

X. **Paridad en desarrollo y producción**: Mantener desarrollo, preproducción y producción tan parecidos como sea posible

Hay que utilizar los mismos servicios de backend (servidor de base de datos, caché, colas, etc.), pequeñas incompatibilidades entre servicios similares pueden generar problemas al realizar el despliegue en diferentes ambientes.

Mantener cadencias bajas entre despliegues en producción asegura que los cambios realizados sean menores, reduciendo riesgos.

XI. **Historiales**: Tratar los historiales como una transmisión de eventos

Manejar logs de eventos de la forma más simple posible, escribiendo a la salida estándar (stdout), y permitiendo que otras herramientas capturen esas salidas y las almacenen de forma centralizada para su verificación en tiempo real o para análisis de distintos tipos.

XII. **Administración de procesos**: Ejecutar las tareas de gestión/administración como procesos que solo se ejecutan una vez

Frecuentemente es necesario ejecutar procesos de administración o mantenimiento una sola vez, por ejemplo correr migraciones o correr scripts de reparación de datos. Estos procesos deberán correr con el mismo código base y usando los mismos manejadores de dependencias. Adicionalmente se pueden habilitar accesos a la consola para ejecutar dichas tareas.

2.5. CaaS

Container as a Service (Contenedor como servicio) se trata de un servicio que permite correr contenedores sin tener que generar la infraestructura para ello. El proveedor se encarga de correr el contenedor y publicar el/los servicios dejándolos disponibles para su uso. Es un modelo de negocios basado en SaaS, que evoluciona para traer soluciones a desarrolladores y administradores de sistemas. En AWS este servicio se implementa como ECS, la tecnología que usamos inicialmente para nuestro proyecto.

2.6. DevOps

DevOps es el acrónimo de Development & Operations (Desarrollo y Operaciones) (O'Reilly, 2009), y es una metodología de desarrollo que promueve la integración de desarrolladores y administradores de sistemas para automatizar los procesos de testeo, integración y despliegue, facilitando el trabajo del desarrollador, que puede dedicar más tiempo a su tarea principal, desarrollar código, y el resto de las tareas se realizan de forma más eficiente. Esta automatización incluye también la creación de la infraestructura necesaria, por lo que esta metodología implica un cambio de filosofía a nivel global dentro de todos los involucrados en el proyecto, haciendo uso de una serie de herramientas que ayudan a agilizar cada paso del proceso, desde la planificación hasta el despliegue.

- **Herramientas de gestión de proyectos**: herramientas que permiten a los equipos crear una acumulación de historias de usuarios (requisitos) que forman proyectos de codificación, dividirlos en tareas más pequeñas y realizar un seguimiento de las tareas hasta su finalización. Muchos admiten prácticas ágiles de gestión de proyectos, como Scrum, Lean y Kanban, que los desarrolladores aportan a DevOps. Las opciones populares de código abierto incluyen GitHub Issues y Jira.

- **Repositorios de código fuente colaborativo**: entornos de codificación controlados por versión que permiten que varios desarrolladores trabajen en la misma base de código. Los repositorios de código deben integrarse con CI/CD, herramientas de prueba y seguridad, de modo que cuando el código se confirme en el repositorio, pueda pasar automáticamente al siguiente paso. Algunos ejemplos de repositorios de código fuente abierto son GitHub y GitLab.

- **CI/CD pipelines**: herramientas que automatizan la verificación, creación, prueba e implementación de código. Jenkins y gitlab CI son las herramientas más populares en esta categoría. Argo CD es otra opción popular para CI/CD nativo de Kubernetes.

- **Frameworks para automatización de tests**: incluyen herramientas de software, bibliotecas y mejores prácticas para automatizar pruebas unitarias, de contrato, funcionales, de rendimiento, de usabilidad, de penetración y de seguridad. Los frameworks populares de automatización de pruebas de código abierto incluyen Selenium, Appium, Katalon, Robot Framework y Serenity (anteriormente conocido como Thucydides).

- **Herramientas de administración de configuración**: permiten a los ingenieros de DevOps configurar y aprovisionar una infraestructura completamente documentada y con versiones completas mediante código. Se dividen en 2 tipos fundamentalmente:

    - **IaC**: Infrastructure as Code (Infraestructura como Código) permite administrar la infraestructura en la nube mediante scripts de diversos tipos. Esto permite realizar la creación, modificación o destrucción de forma automática, lo que hace el proceso repetible y evita problemas causados por errores humanos.

      Existen 2 tipos de enfoques diferentes para realizar este código:

- **Imperativo**: se define un algoritmo, que paso a paso crea los recursos necesarios y los conecta para tener la configuración deseada.

- **Declarativo**: se define el estado final deseado, y la herramienta se encargará de crear todos los recursos necesarios para llegar a dicho estado.

Se prefiere siempre usar el enfoque declarativo, para delegar el "cómo" siempre que sea posible.

Algunos ejemplos son Ansible (Red Hat), Chef, Puppet y Terraform.

- **IaD**: Infrastructure as Data (Infraestructura como Datos) apunta a no generar código propietario para definir la infraestructura, como en IaC, sino describir la misma con formatos de datos populares (como YAML o JSON). Estos formatos carecen de cualquier lógica como loops o iteraciones, y son mucho más predecibles.

  También son más claros de leer y la intención en aislar el proceso necesario para generar la infraestructura, y simplemente centrarnos en el estado final deseado.

  En el mundo real, donde Cloud Computing está evolucionando constantemente, y los proveedores deben modificar sus recursos y sus APIs para generarlos, el uso de IaC lleva consigo un mantenimiento asociado, para seguir la evolución del proveedor.

  En IaD, estos cambios son mucho menores, ya que la herramienta que usamos para la generación de la infraestructura es la responsable de actualizar sus procesos y seguir a las APIs del proveedor, y no nosotros.

- **Herramientas de monitoreo**: ayudan a los equipos de DevOps a identificar y resolver problemas del sistema; también recopilan y analizan

datos en tiempo real para revelar cómo los cambios en el código afectan el rendimiento de la aplicación. Las herramientas de monitoreo incluyen Datadog, Nagios, Prometheus y Splunk.

- **Herramientas de comentarios continuos**: herramientas que recopilan comentarios de los usuarios, ya sea a través de mapas de calor (registro de las acciones de los usuarios en la pantalla), encuestas o emisión de tickets de problemas de autoservicio.

2.7. SRE

Site Reliability Engineering (Ingeniería de Fiabilidad del Sitio) (Google, 2022) son una serie de prácticas y principios que se aplican con la finalidad de tener un producto altamente escalable y confiable. Esta disciplina está altamente relacionada con DevOps y comparten objetivos.

En una empresa, el rol de SRE, estará encargado de mantener funcionando los sistemas, a pesar de cualquier problema que pueda aparecer. Mediante métricas y alertas, se intentará automatizar tareas que permitan que el usuario final tenga acceso fiable.

El primer equipo de SRE se originó en Google en 2003 bajo la dirección de Ben Treynor Sloss. Luego el concepto se comenzó a aplicar en otras empresas, principalmente en grandes compañías de servicios web.

En 2016 Google publica "Site Reliability Engineering, How Google runs production systems" (Jones et al., 2016), donde los miembros del equipo de SRE explican cómo su compromiso con todo el ciclo de vida del software ha permitido a Google crear, implementar, monitorear y mantener algunos de los sistemas de software más grandes del mundo.

Se plantea que el modelo tradicional de administración de servicios, implica el ensamblaje de software existente, y el despliegue del mismo en un mismo ambiente para producir un servicio. Los administradores de sistemas tienen la tarea de ejecutar el servicio y responder a los eventos y actualizaciones a

medida que ocurren. A medida que el sistema crece en complejidad y volumen de tráfico, estos eventos aumentan y el equipo de administradores crece para absorber el trabajo adicional. Esto genera un costo muy alto en trabajo manual y que escala directamente proporcional a la cantidad de servicios a mantener.

El enfoque de Google fue contratar ingenieros con experiencia en software para realizar estas tareas. El resultado fue que este equipo se aburría rápidamente al realizar estas tareas manuales, y al tener las habilidades necesarias, comenzaron a automatizar las tareas.

Adicionalmente, Google limita el trabajo de los administradores, destinando un 50% de su tiempo en manejo administrativos (tickets, guardia, trabajos manuales, etc), lo que permite a los SRE realizar trabajos de automatización. Con el tiempo este porcentaje irá disminuyendo, ya que las tareas serán cada vez más automáticas, pues los servicios se ejecutan y reparan solos.

En el día a día, cada incidente que requiera una intervención manual, recibirá un análisis postmortem, la cual debe establecer qué sucedió en detalle, encontrar todas las causas fundamentales del evento y asignar acciones para corregir el problema o mejorar la forma en que se abordará la próxima vez. Google opera bajo una cultura post mortem libre de culpas, con el objetivo de exponer fallas y aplicar ingeniería para solucionarlas, en lugar de ignorarlas o minimizarlas.

> "Site Reliability Engineering representa una ruptura significativa con las mejores prácticas existentes de la industria para administrar servicios grandes y complicados. Motivado originalmente por la familiaridad - "como ingeniero de software, así es como me gustaría invertir mi tiempo para realizar un conjunto de tareas repetitivas" - se ha convertido en mucho más: un conjunto de principios, un conjunto de prácticas, un conjunto de incentivos y un campo de actividad dentro de la disciplina más amplia de la ingeniería de software"

3. Análisis de la solución actual

Durante la génesis del proyecto, comenzamos con un análisis de la situación contractual. A raíz de dicho análisis, encontramos una cantidad importante de problemas por resolver que iremos detallando en esta sección.

Por requerimiento del cliente final, el desarrollo se llevaría a cabo haciendo uso de los servicios de AWS, utilizando además, la infraestructura de redes ya creada para otro proyecto. Un punto de gran importancia, fue contemplar los ambientes por crear: desarrollo, testeo y producción. Los tres ambientes debían replicar la misma infraestructura en diferentes cuentas de AWS, para cumplir con ciertas políticas de seguridad propias de cada ambiente.

Como el acceso a las redes del cliente final (Corning) es extremadamente restringido, siendo sólo posible con dispositivos certificados y homologados por la empresa, fue necesario crear dos ambientes similares, pero con sus redes accesibles desde una misma vpn de nuestra empresa (ITX), para simplificar el acceso de los desarrolladores y el equipo de testeo externos a Corning.

En resumen, el proyecto implementa una plataforma que consta de una interfaz gráfica colaborativa en la cual los cambios deben refrescarse en tiempo real en diferentes clientes que estuvieran visualizando lo mismo, al mismo tiempo. Un requisito de la arquitectura del desarrollo fue la adopción de un modelo basado en servicios y donde el backend principal, implementa una REST API y GraphQL para manejar subscripciones a eventos. Por su parte los datos persisten en una base de datos PostgreSQL.

Desde un área de desarrollo propia del cliente final (Corning), se implementó un servicio con una REST API para acceder a datos de procesos que se almacenan en una base de datos orientada a grafos, utilizando además, otra base de datos también orientada a grafos, para mostrar información necesaria en la interfaz gráfica. Corning además implementó un servicio adicional de consulta a ambas bases de datos, también implementando otra REST API.

Estos servicios, desarrollados por Corning, también hacen uso de bases de datos ya existentes por medio de servicios de terceros.

La autenticación utiliza servicios ya existentes en Active Directory propios de Corning, validando la identidad de los usuarios para su ingreso al sistema. El mecanismo de autenticación y autorización se basa en los estándares de SAML (SAML, 2007), OIDC (OpenID, 2022) y OAuth2 (Parecki, 2022), generando tokens de identificación, acceso y refresco, utilizados para validar en los diferentes servicios de backend, que los accesos a estos servicios se realizan por personas autorizadas.

Como las consultas más frecuentemente utilizadas por la plataforma requieren procesar grandes volúmenes de datos, este es un punto crítico causante de posibles demoras. Por ello, se contempló el agregado de capas de caché en lugares estratégicos, para acelerar los tiempos de respuesta lo más posible.

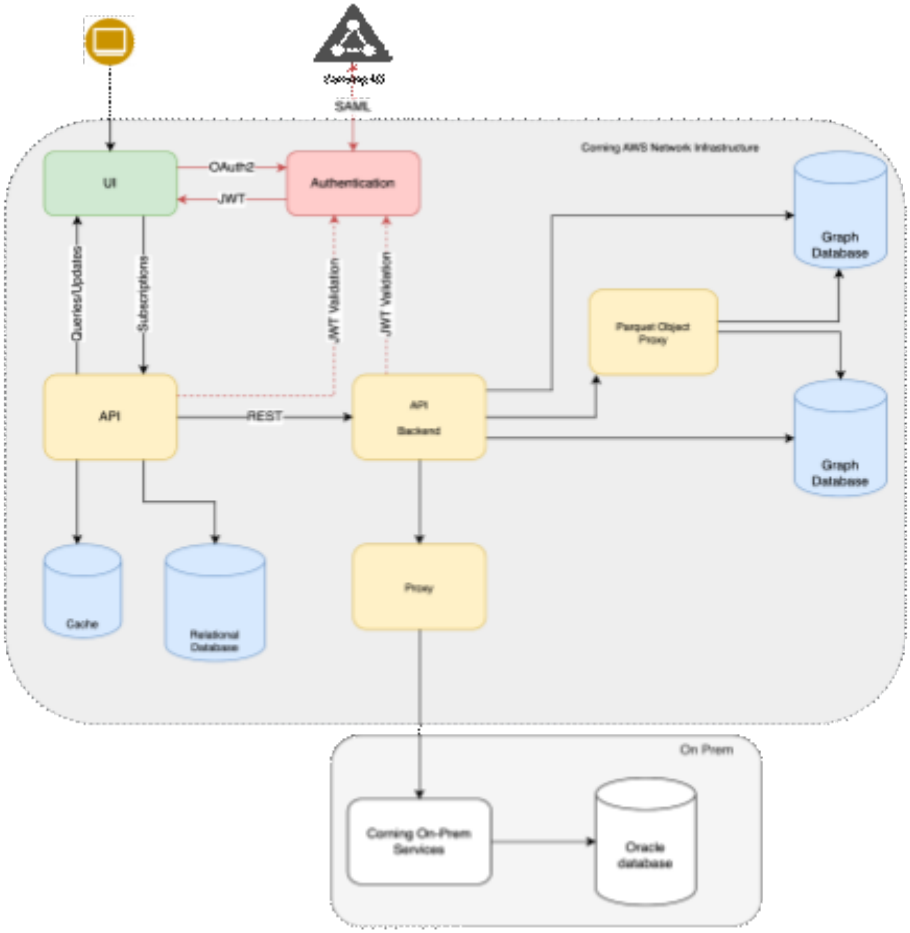


Diagrama de servicios a implementar y su integración con servicios existentes

La plataforma completa provee acceso a nivel mundial a los servicios mencionados, por lo que debe considerar la alta disponibilidad como su escalabilidad de forma simple.

Durante las etapas iniciales del proyecto, se comenzó con el desarrollo de las aplicaciones y la generación de la infraestructura para dar soporte a las mismas. Los requerimientos iniciales hacían hincapié en aplicar la filosofía DevOps, por lo que el equipo tuvo que entrenarse en estas temáticas, sobre todo para poder definir la estrategia para abordar todos los puntos mencionados en los párrafos anteriores. En línea con las buenas prácticas que pregona la filosofía DevOps, utilizar infraestructura como código fue el pilar para comenzar a trabajar. Esto significa evitar realizar tareas manuales en favor de programar cada cambio a realizar con cualquier recurso propio de la infraestructura. Los recursos en este caso, son de AWS, por lo que aquí se abrieron una vasta cantidad de posibilidades para abordar la programación de la infraestructura sobre recursos de AWS. Sin embargo, observando las tendencias mundiales, y considerando que estamos frente a una arquitectura íntegramente desarrollada para ser desplegada en un proveedor de cloud, entendimos que no podíamos dejar de analizar el concepto de cloud native (CNCF, 2018). La decisión más importante que tomamos, fue la de utilizar contenedores para manejar las cargas de trabajo, en lugar de desplegar las aplicaciones en soluciones propietarias del cloud provider. De esta forma, teníamos garantías de poder mover nuestro trabajo a un cloud provider diferente o incluso a un datacenter on premise. La clave entonces, fue la adopción de contenedores para representar nuestras aplicaciones. Lo que en esta primera instancia no teníamos claro, era con qué tecnologías correr los contenedores.

La primera aproximación que se hizo, fue levantar servidores linux, en forma manual, y dentro de ellos correr el servicio de docker con las imágenes de contenedores ya generadas. Esto permitió el inicio del desarrollo, pero no de una forma automatizada. En cada servidor se corría un servicio del tipo watcher, de código abierto, llamado *ouroboros* (Ouroboros, 2019), que revisaba

el repositorio de imágenes docker por versiones nuevas, y al encontrar una nueva, descargaba la nueva versión y la lanzaba nuevamente, por lo que los desarrolladores podían ver sus cambios en los ambientes de desarrollo / testing / producción a medida que se iban promocionando a los diferentes ambientes.

El paso siguiente fue recrear una infraestructura que soportara estos contenedores y que pudiera generarse en forma automatizada, sin pasos manuales, utilizando terraform (Terraform, 2022) que era la herramienta utilizada en la empresa para generar las redes y otros servicios con los que deberíamos integrarnos.

El número limitado de recursos humanos hizo que buscáramos una solución lo más desatendida posible en cuanto a la ejecución de las cargas de trabajo, para poder enfocarnos en las tecnologías de comunicación, almacenamiento y visualización. Por esta razón, elegimos AWS ECS como orquestador de contenedores. El servicio se combina con otros más como es el caso de EC2 o Fargate, ALB, etc. Todos fueron siendo necesarios a medida que se avanzaba con el proyecto, y es por ello que en las siguientes secciones describiremos brevemente cada uno de ellos.

3.1. Servicios AWS

3.1.1. EC2

Amazon ofrece un servicio de servidores en la nube, EC2 (Elastic Compute Cloud) (AWS EC2, 2022). El mismo permite generar un servidor de diversas características, con distintas capacidades de memoria y procesador, y agregar discos virtuales. A estos servidores virtuales, AWS los denomina instancias, y cada tipo de instancias tiene varios modelos de pago, permitiendo conseguir costos más bajos contratándolas por ejemplo durante largos períodos de tiempo, o con un sistema de subastas, o seleccionado instancias de corta vida y baja prioridad para realizar tareas tipo batch. El uso de estos servidores acarrea la responsabilidad de la configuración y administración del sistema

operativo que sea provisto, junto con el costo operativo de actualización, instalación de parches de seguridad, etc. En contraposición, y como una gran ventaja, debemos destacar su versatilidad dado que permiten realizar cualquier tipo de configuraciones, y existen una vasta cantidad de imágenes de base (templates desde los que se crean las instancias), que usan diversos sistemas operativos, arquitecturas y en algunos casos proveen licencias de uso para ciertas aplicaciones.

Este servicio fue el seleccionado en los comienzos del proyecto para correr las instancias en contenedores Docker, y que continuamos usando en la actualidad para ciertos servicios especiales.

3.1.2. Fargate

Por otro lado, en diferentes servicios que requieren cómputo, Amazon provee un servicio llamado Fargate (Amazon Fargate, 2022), un motor de cómputo sin servidor para contenedores que funciona con ECS y EKS. El mismo provee servidores compartidos, manejados por la plataforma, por lo que la administración y configuración ya se encuentra resuelta y se paga por el cómputo realizado. El escalado y el monitoreo son responsabilidad de AWS.

Fargate permite enfocarse en las aplicaciones, siendo sólo necesario encargarse de definir los requisitos de escalado, almacenamiento, redes y contenido de las aplicaciones, y se paga por el uso de cómputo otorgada en cada momento.

3.1.3. Amazon ECS

Amazon ECS (AWS ECS, 2022) es un servicio de ejecución y orquestación de contenedores de forma sencilla. El mismo provee la capacidad de escalar recursos e instancias sin necesidad de manejar la infraestructura de un clúster. ECS tiene dos modelos, que permiten diferentes niveles de control, estos son Fargate y EC2. En esta etapa inicial la elección de optimización de recursos que permite EC2 contra la delegación de responsabilidades que ofrece

Fargate, siendo el escalado uno de los puntos fundamentales, hizo que la balanza se inclinara a favor de ECS con Fargate, ya que dejaba más tiempo para la definición y desarrollo de los servicios y aplicaciones.

ECS es una solución propietaria de AWS. Si bien utiliza contenedores contemplados por OCI, las componentes que hacen viable la orquestación generan un alto acoplamiento con el proveedor de cloud.

3.1.4. Load Balancers (ALB)

Como sucede en la mayoría de los orquestadores de contenedores, el servicio ECS corren varios contenedores del mismo tipo con el fin de proveer redundancia ante fallos y mejores tiempos de respuesta. La escala de los contenedores es manejada de forma estática o automáticamente en base a métricas que indican qué escala se requiere para servir el tráfico. Si bien esto parece trivial durante su definición, el ingreso de tráfico adecuado hacia contenedores que mantienen una escala elástica, debe modificar constantemente las configuraciones de los recursos que ingresan tráfico. Es aquí donde el service discovery (AWS SD, 2022) surge como una necesidad para adaptar los cambios en la escala de un cluster de contenedores, y reflejarlos en recursos que balanceen el ingreso de tráfico.

No todos los servicios pueden balancearse, en general cualquier servicio HTTP que respeta los 12 factores no debería tener problemas en balancear cargas de trabajo entre varios contenedores. No es el caso de servicios TCP como bases de datos relacionales, u otros servicios que requieren que determinadas operaciones las aborde un único miembro del cluster.

En nuestro caso, los contenedores se distribuyen en dos subredes, cada una en diferentes zonas de disponibilidad, lo cual agregaba redundancia a nivel de red para que los contenedores siempre corran en ambas zonas. Al tener servicios basados en HTTP implementados siguiendo los 12 factores, hemos podido considerar el balanceo. Para ello, hemos utilizado el servicio llamado Application Load Balancer (AWS ALB, 2022) que provee no solo el balanceo

automático de carga entre las diferentes instancias de nuestras aplicaciones, sino que también considera la provisión de un certificado SSL y redirección del puerto 80 (HTTP sin seguridad de encriptación) al puerto 443 (HTTPS con encriptación SSL) para forzar el uso de encriptación para incrementar la seguridad en la comunicación.

Este servicio se encuentra totalmente integrado con ECS, por lo que el autoescalado de instancias se refleja automáticamente, incorporando o eliminando instancias al pool del ALB. Por su parte, el ALB provee un chequeo de salud de cada instancia en el pool, para comprobar que cada una esté corriendo adecuadamente, y en caso que así no sea, no enviar tráfico a instancias defectuosas.

Como beneficio adicional, ALB, integra un firewall para aplicación web llamado Web Application Firewall (AWS WAF, 2022). El mismo provee en forma inmediata protecciones contra DDoS y otros ataques comunes, que amenazan la disponibilidad. Permite también implementar protecciones a las principales amenazas detectadas por OWASP (Open Web Application Security Project) (OWASP, 2022), un proyecto abierto de seguridad que analiza y registra las principales amenazas y debilidades que poseen las aplicaciones web.

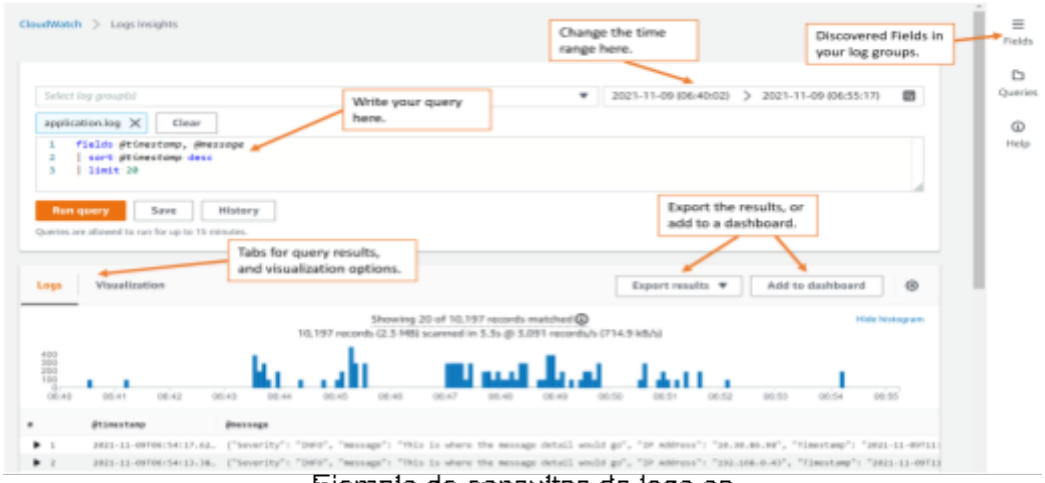
3.1.5. Métricas (CloudWatch)

Amazon CloudWatch (AWS CloudWatch, 2022) es un servicio de observabilidad (IBM Obs, 2022). El mismo recopila datos operativos sobre el consumo de recursos (métricas), como además los logs de determinados servicios. Esto permite visualizar y consultar logs, monitorizar consumos de red, memoria y procesador, como cualquier recurso medible. Como consecuencia, CloudWatch permite generar alertas y eventos para realizar acciones automatizadas, simplificando el análisis y resolución de problemas para mantener el buen funcionamiento de la infraestructura.

Este servicio está disponible en forma nativa para la mayoría de los servicios de AWS, por lo que su uso fue trivial y no necesitaba de más configuración que

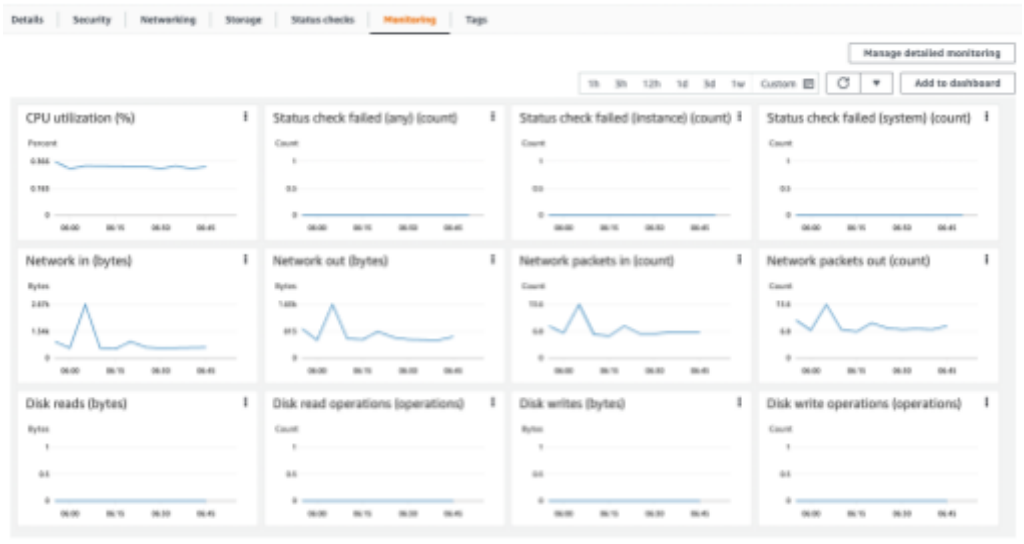
generar un grupo de logs específico para cada servicio, para tener la información separada adecuadamente.

El acceso a los logs se realiza de una forma simple a través de la consola de AWS, la cual permite consultas y filtros con un lenguaje de consultas propietario, similar a SQL, muy potente al momento de buscar en una gran cantidad de datos.



Ejemplo de consultas de logs en Cloudwatch

De la misma forma, mediante la consola de AWS, se tiene acceso a un panel de control con las métricas más importantes, sin necesidad de instalar productos o realizar configuraciones adicionales. Este panel es configurable, y pueden agregarse las métricas que sean necesarias.



Panel de control de AWS Cloudwatch

3.1.6. Permisos (IAM)

La gestión de los recursos mencionados anteriormente, requiere se definan previamente los permisos que habilitan su gestión. AWS provee Amazon Identity and Access Management (AWS IAM, 2022), que es un servicio en el cual se definen roles y políticas que permiten el acceso a recursos, y permite asignarlos a usuarios, grupos e inclusive a otros recursos.

La granularidad de permisos es extremadamente fina, siendo su gestión una tarea compleja, pero a su vez muy potente. Ofrece simplificaciones que evitan tener que redefinir permisos, en base a políticas y roles. Además, AWS provee la capacidad de asumir temporalmente roles a través de AWS STS (AWS STS, 2022).

Entender y hacer una buena gestión de permisos en cualquier cloud provider es una labor compleja pero fundamental para mantener seguros los recursos gestionados. Por ello, una buena práctica que hemos aplicado es la de ofrecer el menor privilegio necesario para operar en cada recurso (AWS IAM Prácticas, 2022).

3.1.7. Cognito

Amazon Cognito (AWS Cognito, 2022) es un proveedor de identidad completamente administrado que permite el registro, la verificación y el inicio de sesión de usuarios.

Amazon Cognito provee dos componentes diferentes:

- **Los grupos de usuarios (User Pools)** son directorios de usuarios estándar donde los usuarios inician sesión mediante usuarios definidos dentro del propio Cognito, o proveedores de identidad de terceros, como Facebook, Google, o inclusive un proveedor SAML. Luego de la autenticación exitosa, los usuarios reciben tokens de acceso, que pueden usarse para acceder a recursos.

- **Los grupos de identidad (Identity Pools)** ofrecen a los usuarios recibir credenciales temporales de AWS para ofrecer acceso directo y limitado a los recursos de AWS.

En nuestro caso se utilizó Cognito para autenticar contra el proveedor de identidad de Corning, Azure AD, que fue conectado mediante SAML, un estándar de código abierto basado en XML que permite el intercambio de información, tanto de autenticación como de autorización entre un proveedor de identidad y un proveedor de servicios, permitiendo integrar el inicio de sesión único o Single Sign-on (SSO).

La UI comienza el flujo de OIDC, que solicita el login mediante el proveedor de identidad de Azure AD, el mismo valida la información, y si las credenciales son correctas, devuelve un paquete de autorización a Cognito, el mismo genera un token JWT que es devuelto a la UI, y que la misma va a almacenar para su uso en cada solicitud que se realice a los servicios de backend. Ante cada solicitud, cada servicio valida el token recibido con Cognito, y verifica que sea válido. De esta forma todos los servicios tienen un nivel de validación de identidad.

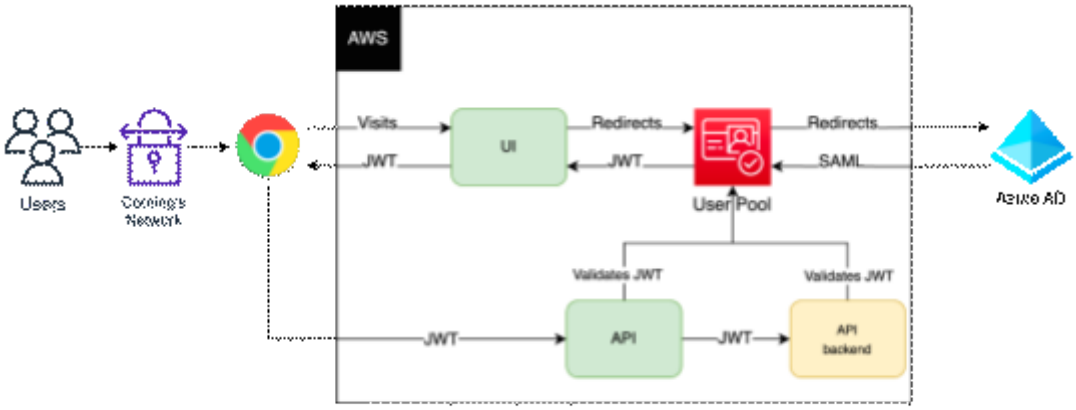


Diagrama de autenticación usando Cognito

Estos tokens expiran cada 15 minutos por lo que obtener un token válido no permite el acceso indefinido. Junto al token de autorización, se provee un token de refresco, el cual permite, mediante una llamada, el conseguir una nueva token válida, sin pasar por el flujo de OIDC nuevamente.

3.1.8. Secret Manager

Uno de los puntos importantes a resolver, era cómo almacenar las claves de acceso a los diferentes servicios. Las mismas deberían estar en un repositorio al cual sólo debían tener acceso las personas autorizadas. AWS provee un servicio para esto, llamado Secret Manager (AWS Secret Manager, 2022), donde es sencillo crear las claves, almacenarlas, rotarlas, y limitar su acceso por medio de roles IAM, asignable a usuarios o grupos.

Para evitar que las claves se encuentren almacenadas en repositorios git o similares, donde desarrolladores y otros usuarios sin nivel de acceso adecuado tengan acceso a dichas claves, se creó un módulo de terraform que las maneja, permitiendo que se generen automáticamente.

3.1.9. RDS PostgreSQL

Como se mencionó al inicio del presente capítulo, algunos sistemas requieren de una base de datos relacional. Luego de varios análisis se optó por el motor PostgreSQL (PostgreSQL, 2022). Los requisitos de escalabilidad y redundancia hacía necesario realizar la configuración de un clúster de base de datos, lo cual implica un tiempo considerable invertido en instalación, configuración y mantenimiento del mismo. AWS proporciona un servicio llamado Amazon Relational Database Service (AWS RDS, 2022), que provee un clúster autogestionado, el cual admite configurar diferentes motores de bases de datos como son MySQL, PostgreSQL, Oracle y SQL Server. El acceso desde librerías o clientes convencionales hacen que su uso sea transparente, agregando además soluciones de backups y restauración a un punto en el tiempo (AWS PIT, 2022), escalado, y actualizaciones automáticas.

3.1.10. ElastiCache

También en la introducción del presente capítulo mencionamos la aceleración de respuestas mediante el uso de una caché. El procedimiento consiste en

guardar las respuestas de consultas que son repetitivas y de poca variabilidad, en almacenamientos de alta velocidad. AWS cuenta con un servicio llamado AWS ElastiCache (AWS ElastiCache, 2022), que es compatible con Redis y Memcached, siendo el primero la opción que implementamos.

3.1.11. BD AllegroGraph y EDG

Para el almacenamiento en base de datos orientadas a grafos, se eligieron dos motores diferentes, para distintos propósitos, estos son AllegroGraph (AllegroGraph, 2022) y TopQuadrant EDG (TopQuadrant, 2022), los cuales no tienen servicios provistos por AWS.

El equipo de desarrollo de AllegroGraph, provee una imagen oficial para los EC2 de AWS, con la base de datos instalada, y una licencia de uso que se paga por hora, lo que permite una instalación y configuración muy simple y el licenciamiento ya resuelto.

Para EDG, se compraron licencias y se realizó la instalación y configuración en EC2, siendo responsabilidad del equipo el mantenimiento de la misma. Para automatizar la configuración de la misma se ha utilizado Ansible (Ansible, 2022), una herramienta de IaC, que permite mediante código automatizar este tipo de tareas.

3.2. CI/CD

La CI/CD (RedHat CI/CD, 2022) hace referencia a las prácticas combinadas de integración continua, de entrega continua y de despliegue continua. Estas prácticas consisten en la automatización de las tareas necesarias para tener disponible un producto, desde el repositorio de código fuente hasta que esté disponible en los ambientes productivos. Integración continua hace referencia a los procesos de compilación y pruebas, entrega continua al proceso de publicar en un repositorio ese artefacto, resultado de la compilación, y despliegue continuo hace referencia al despliegue de dicho artefacto en un ambiente productivo. En nuestro caso puntual, en la creación de las imágenes de las

aplicaciones y servicios se realiza entrega continua, y en el caso de infraestructura se realiza despliegue continuo.

Esta automatización permite realizar entregas con una periodicidad elevada, y ahorrar los tiempos y riesgos que implicaría realizar estos procesos de forma manual.

3.2.1. Versionado del código

Como explicamos anteriormente, la filosofía DevOps era un pilar fundamental en nuestro desarrollo, focalizando sobre todo en que la construcción y despliegue de las aplicaciones, así como la gestión de la infraestructura se realicen de forma automática. La forma de hacerlo posible, fue mediante infraestructura como código. Código que debe versionarse en un SCM (SCM, 2022), por ejemplo Git. Automatizar en pipelines que accionan ante eventos de Git como push, pull o merge requests (según sea la plataforma de versionado basada en git), controles de linters, testing de unidad o de integración e incluso la ejecución propia de los scripts de infraestructura como código como un paso del pipeline, convierten al proceso de modificar la infraestructura en un proceso similar al de desarrollo, solo que en vez de generar un artefacto para su despliegue, la salida del pipeline consiste en actualizar la infraestructura completa. Como plataforma de control de versiones se utilizó Gitlab, ya que el cliente ya contaba con una cuenta corporativa, y en consecuencia se eligió Gitlab CI/CD (GitLab, 2022) como herramienta de integración y despliegue continuo.

En el proceso se creó un template especial con las configuraciones comunes a todos los proyectos, la cual permitiría, al ser incluida en los scripts de CI/CD de cada proyecto, que estos sean más simples, compactos y genéricos. Este template setea variables de entorno e inicializa archivos para poder utilizar los clientes de terraform, aws, kubectl, etc. Para guardar claves de seguridad y tokens de acceso se utilizó una característica de Gitlab CI, que es proveer secretos, que se setean a nivel de grupo, y que sólo son visibles por los administradores, permitiendo el almacenamiento y uso seguro de las mismas.

También se generaron una serie imágenes con las utilidades necesarias para compilar y realizar el despliegue las diferentes aplicaciones, de esta forma armamos repositorios que constituían imágenes para construir contenedores que corrieran NodeJS, Python, entre otras. Estos repositorios tenían una CI, que construía las imágenes Docker y las guardaba en el container privado del repositorio. Los otros repositorios utilizarían estas imágenes para la construcción de sus propias imágenes.

3.2.2. Ambientes

Un factor fundamental en el proceso de desarrollo fue la inmutabilidad del código entre diferentes ambientes, era fundamental que el mismo código que se desarrollaba en desarrollo, y era probado en testing, llegara a producción sin cambios. Para esto se utilizó una estrategia de ramas en git, consistente a través de todos los repositorios, en los cuales dev, significaba desarrollo, test, testeo y prod, significaba producción. Estos nombres se usaban para nombrar los archivos de configuración que centralizaban las diferencias entre los distintos ambientes, como urls de comunicación para los servicios que se utilizaban, o ids de las cuentas donde se iban a desplegar.

Los scripts de CI tomaban valores también de variables de CI seteadas en gitlab, en forma privada a nivel cuenta, como claves y llaves de acceso para las diferentes cuentas de AWS y valores sensibles que no era deseable que estuvieran en un repositorio a la vista de todos los desarrolladores.

Con esta estructura, el agregado de nuevos ambientes implicaba sólo la creación de una nueva rama, y agregar los archivos de configuración y variables de CI, por lo que no era un trabajo complicado y permitía generar ambientes nuevos para probar cambios importantes, que pudieran afectar gravemente un ambiente.

3.2.3. Flujo de trabajo en git

Al ver que el repositorio git y sus ramas tendrían una gran responsabilidad en el despliegue de las aplicaciones e infraestructura, se decidió hacer un análisis profundo sobre el esquema de branches a utilizar, decidiendo finalmente usar la metodología de trabajo GitLab Flow (GitLab Flow, 2022).

Como el trabajo se iba a organizar con Scrum, íbamos a tener un ciclo de trabajo de 2 semanas, en el cual el día miércoles terminaba una Sprint y el día jueves comenzaba una nueva.

En cada Sprint, se definirían una serie de *Historias de Usuario* a codificar, al comienzo de cada *Historia de Usuario* se debía crear una nueva rama, a partir de la rama "dev", con el nombre "feature/dev-xxx", indicando el número de *Historia de Usuario* para identificarlo.

Cuando la *Historia de Usuario* se termina, y el desarrollador la prueba localmente, se procede a solicitar la mezcla a "dev" a través de la interfaz de Gitlab. Luego de revisarse y aprobarse el código por varias personas, se procede a la mezcla, y la CI automáticamente realiza la construcción y despliegue en la cuenta de desarrollo. Este proceso se repite durante la Sprint para cada *Historia de Usuario*, integrando los cambios solicitados y permitiendo una prueba global del producto en cada momento.

Llegado el lunes de la segunda semana de la Sprint, se procede a realizar el despliegue de la rama de desarrollo sobre el ambiente de testeo y a congelar los ambientes para permitir el testeo en forma correcta. Si bien se permite continuar desarrollando, no se permite mezclar código que no pertenezca a corrección de errores encontrados por el grupo de testeo.

La finalidad es encontrar los errores en el ambiente y solucionarlos antes del día miércoles, que es el día en que esta demostración se realiza en el ambiente de testeo, donde el equipo de testeo posee la versión más estable al momento.

Cuando el cliente aprueba el trabajo mostrado, ya se encuentra listo para ser desplegado en producción, que será realizada el día jueves de la segunda

semana, luego de lo cual se libera la rama de desarrollo para permitir incluir nuevos cambios y continuar el flujo normal.

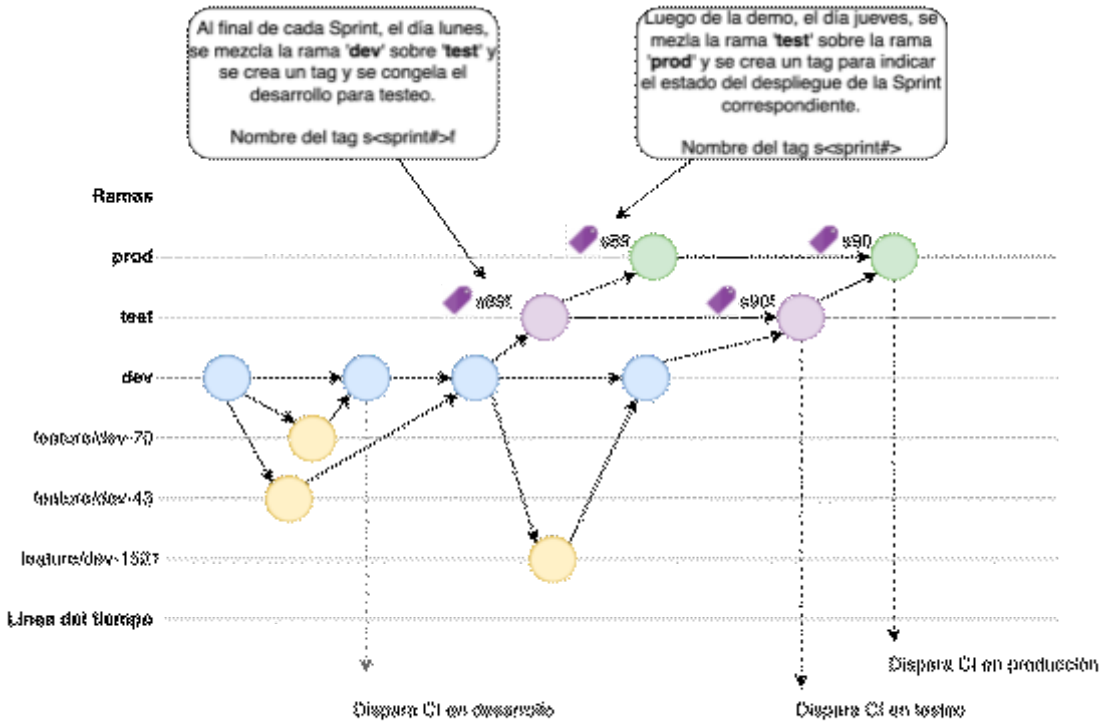


Diagrama de esquema de ramas en repositorio y sus despliegues automàticos

En casos especiales, donde se encuentran problemas en producción, en caso que el defecto sea importante, se permite agregar cambios directamente en las ramas de testeo o producción, en forma de Hotfix, ingresando el cambio mínimo necesario para solucionar el problema.

3.2.4. Aplicaciones

Como se explicó en los puntos anteriores, Gitlab CI realizaba la construcción de la imagen, y el despliegue en el ambiente de desarrollo. De la misma forma el despliegue en los otros ambientes, implicaba solo mezclar los cambios en las ramas de testeo y producción. El código siempre era el mismo, haciendo que las particularidades de cada ambiente sean variables, que fueran obtenidas desde variables de entorno o archivos de configuración, por lo que cuando llegaba a producción el código en sí se había testeado al menos 2 veces.

Los scripts de Gitlab CI para las aplicaciones y servicios, incluían una serie de tareas iniciales de validación, corriendo chequeos de calidad de código y tests automatizados, una segunda etapa de construcción, que en general usaba Docker para construir una imagen en base a un script incluido en el mismo repositorio, que dependía de la rama que se estaba corriendo, y dicha imagen era publicada en el repositorio de imágenes correspondiente. Finalmente, si las dos primeras etapas terminaban sin errores, comenzaba una tercer etapa, con una tarea que reiniciaba el servicio, para que tomara la nueva imagen recién publicada.

Desde el comienzo, se realizaron aplicaciones en diversas tecnologías, como React JS, Node JS y Python, pero en todos los casos, el proceso se unificaba en las segunda y tercer etapa, ya que para todas estas tecnologías, era posible generar un script de Docker (Dockerfile) que utilizara una imagen determinada, que permitiera compilar el código y en algunos casos utilizar una segunda imagen que permitiría implementar el servicio deseado. Una vez generada esa imagen de Docker, la publicación y el reinicio funcionan de la misma forma sin importar el contenido de la misma.

3.2.5. Infraestructura como código

Para facilitar la creación de esta infraestructura, hicimos uso de terraform (Terraform, 2022). Si bien existe una solución propia de AWS llamada Cloudformation (AWS CloudFormation, 2022), terraform abstrae justamente al desarrollador de aprender una solución propietaria y fomenta el uso de una herramienta transversal. Terraform implementa infraestructura como código (IaC) (Morris, 2016) haciendo uso de un enfoque declarativo. Es una herramienta muy popular porque posee una gran diversidad de aprovisionadores, dando soporte a diferentes plataformas de virtualización y API de cloud provider, entre las que podemos mencionar AWS, GCP, Azure, VMWare VSphere, Xen, Openstack, etc.

Terraform simplifica la creación de servidores, bases de datos, redes, y en general, cualquier otro recurso disponible en las diferentes plataformas. AWS

proporciona aprovisionadores para toda su plataforma, por lo que usamos esta herramienta para generar toda la infraestructura necesaria.

Como sucede en la programación de aplicaciones convencionales, terraform permite agrupar funcionalidades en módulos que encapsulan comportamiento reutilizable. Por esta razón, se crearon una serie de módulos de terraform que simplificaron dramáticamente los despliegues de múltiples servicios y aplicaciones a través de parametrizaciones.

Al igual que con las aplicaciones, la infraestructura hacía uso de los nombres de las ramas para definir los nombres de los recursos en cada ambiente, y de esa forma todos los scripts eran genéricos, y sólo era necesario promover el código a las ramas correspondientes, para que se despliegue la infraestructura deseada.

Se generó un repositorio de módulos de terraform, los cuales iban a ser utilizados por cada script de generación de infraestructura, inclusive en otros proyectos de la empresa.

Se contempló la posibilidad de que haya más de un ambiente por cuenta de AWS, por lo que se dividieron los scripts de creación en dos partes, una parte que generaría la infraestructura que debe compartirse entre ambientes pero debe ser única a nivel de cuenta, y la segunda parte para la infraestructura que puede generarse múltiples veces en una misma cuenta.

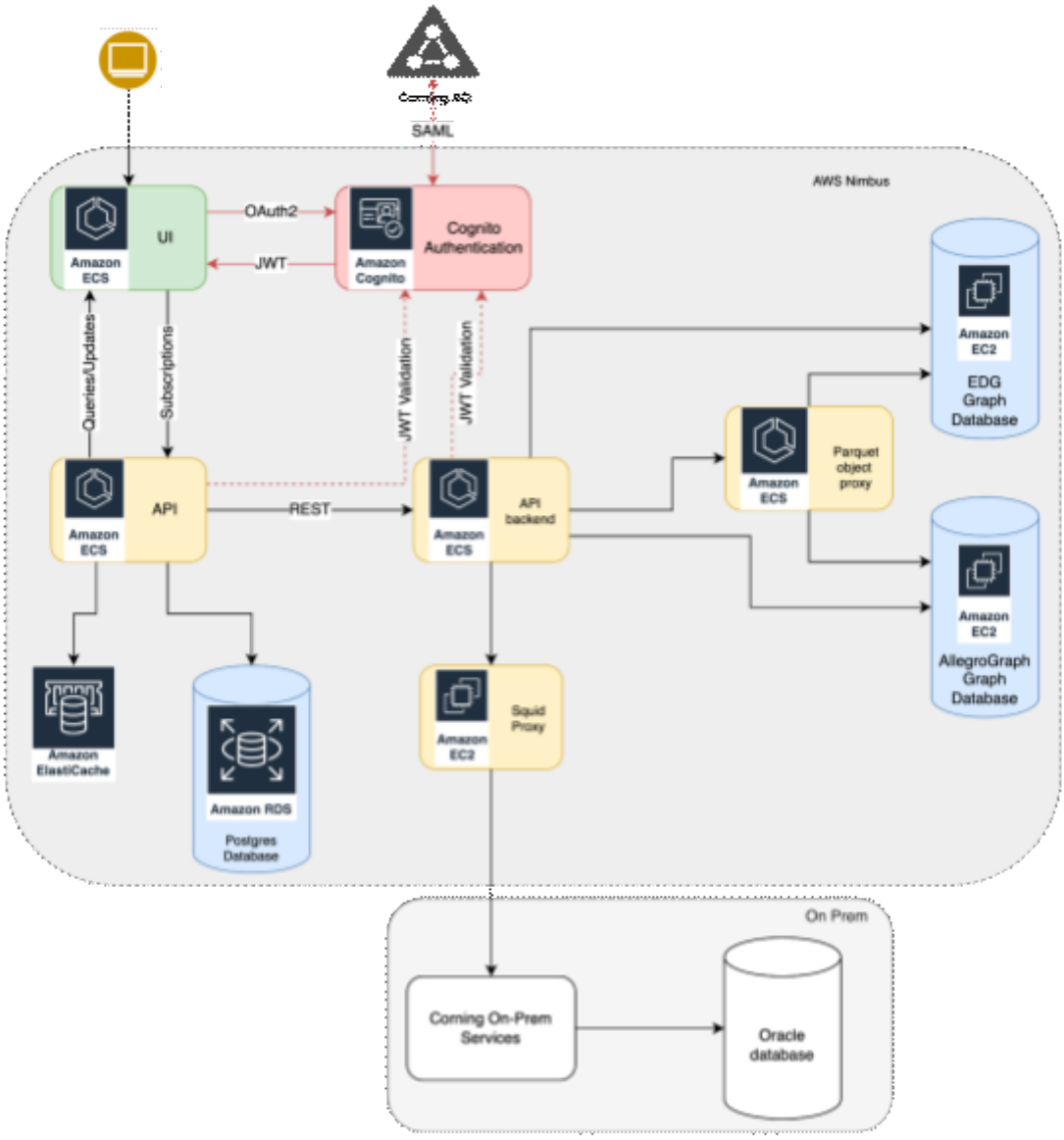
Se generaron dos repositorios para estos scripts, uno a nivel cuenta, y otro a nivel ambiente, y se adecuaron todos los módulos para que utilicen ambos valores en sus nombres, para asegurar la unicidad de los mismos. Primero era necesario correr el script de cuenta, y luego los de ambiente, que tomaban los valores necesarios de la salida de los primeros.

En cuanto a las tareas de la CI, en el repositorio de módulos sólo se implementó una tarea, de validación de código, para asegurar estándares de codificación. En los repositorios de scripts, se agregó a esta tarea de validación, tareas de creación y de destrucción para los scripts de terraform de

las mismas. Los equipos de desarrollo poseían permiso de lectura y escritura en el ambiente de desarrollo, pero sólo de lectura en los ambientes de testeo y producción, por lo que sólo la CI tendría posibilidad de modificar los ambientes superiores, para asegurar la trazabilidad de los cambios realizados.

3.3. Diseño de infraestructura en AWS

El resultado final de la primera versión del sistema completo en AWS (Montedoro, 2021), llegamos al siguiente diagrama que representa la solución completa:



Infraestructura del Proyecto en AWS con su implementación inicial con ECS

Este modelo nos acompañó durante la primera etapa del desarrollo, siendo necesario modificar los scripts de despliegues con ajustes propios de la evolución del proyecto a lo largo de dos años de uso.

Para la generación de la infraestructura de cada nuevo servicio o aplicación, creamos un módulo que se encargaba de crear el repositorio para la imagen de Docker que se iba a correr, el servicio de ECS con su ALB correspondiente, un alias en los registros de DNS y todos las restricciones de seguridad para que sólo fueran accesibles de la forma adecuada.

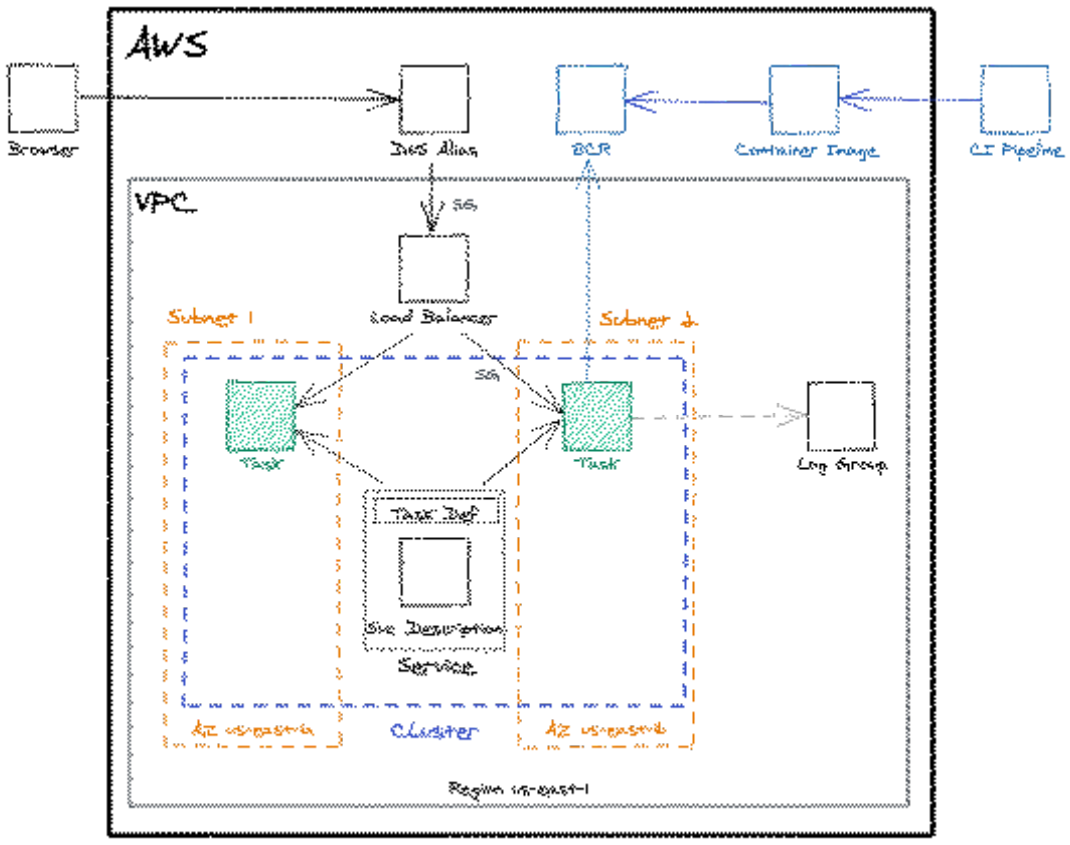


Diagrama de infraestructura AWS utilizado para cada servicio y su despliegue en su versión con ECS

Gracias a este módulo, para crear la infraestructura necesaria para correr un nuevo servicio, sólo era necesario agregar al script de terraform el llamado de este módulo, proveyendo el nombre del mismo.

Como contra fundamental, la infraestructura era absolutamente redundante y sobredimensionada para muchos de los servicios, que no requerían de gran

poder de cómputo, y en consecuencia íbamos a tener costes asociados mucho más elevados de los necesarios.

Esta automatización permitió avanzar sin contratiempos ante el agregado de cada nuevo servicio, permitiendo enfocarse en el desarrollo de los servicios y aplicaciones, sin tener que pensar en la infraestructura necesaria para que estos corran.

4. Análisis de la alternativa

Una vez que las aplicaciones y servicios maduraron lo suficiente, los cambios en la infraestructura se redujeron. Así es como dimos un paso más sobre el análisis de la infraestructura disponible buscando mejorarla en cuanto a su estabilidad, observabilidad y costos.

Casi de forma simultánea, surgieron otras dos nuevas aplicaciones que se incorporaron a la plataforma. Este hecho mostró que la madurez de la arquitectura, no estaba exenta de cambios, y puso en duda la simplicidad de orquestar nuevas componentes.

Los conocimientos adquiridos en el plano de DevOps, y habiendo aplicado las metodologías de "Twelve Factors" durante 2 años, entendimos que estábamos preparados para dar un salto a otra arquitectura, que nos permitiera tener un control más granular sobre los recursos a utilizar y poder aplicar técnicas más simples y limpias a la hora de hacer despliegues.

4.1. ¿Qué es Kubernetes?

Kubernetes (Kubernetes, 2023), deriva su nombre de la palabra griega, κυβερνήτης, que significa "timonel" o "piloto". Motiva su nombre, ser un producto orientado a la gestión de aplicaciones de contenedores. Suele abreviarse el nombre como K8s, debido al reemplazo de las ocho letras "ubernete" con el número 8.

K8s una plataforma portable y extensible de código abierto para la organización de contenedores que automatiza la creación, operación y escalado de aplicaciones de contenedores, basado en una configuración declarativa.

4.2. Historia

En su origen tenía otro nombre: Borg (Borg, 2015), y era un proyecto interno de Google, que sirvió a la empresa para correr cientos de miles de aplicaciones en contenedores por más de una década. Borg tenía una compleja arquitectura (Verma et al., 2015) que k8s tomó de base por el excelente desempeño que tuvo en Google. En 2014 Google libera una primera versión de k8s, abriendo un proyecto en la plataforma GitHub, y se presenta formalmente al mundo en una Dockercon (Dockercon, 2014). La noticia fue muy bien recibida por la comunidad de software libre, reflejándose inmediatamente en los rankings de Github siendo k8s uno de los proyectos con más estrellas y descargas. Inmediatamente se adhieren Microsoft, RedHat, IBM, Docker, AWS y otras grandes compañías al proyecto.

Al año siguiente Google se une a la Linux Foundation para formar la Cloud Native Computing Foundation (CNCF, 2022), donando infraestructura para la gestión de la CNCF (Evans, 2018). Además se presenta la versión 1.0 de K8s. Por su parte, Google Cloud lanza Google Kubernetes Engine (GKE, 2022), agregando el servicio manejado de clúster de Kubernetes a sus servicios. Este sistema simplificó aún más la creación, el funcionamiento, la protección y el mantenimiento de los clústeres de Kubernetes.

A lo largo de 3 años se realizan diferente integraciones y mejoras, y en 2017 se logra la versión 1.6, la cual es la primera de las versiones que puede considerarse estable, y comienza a adoptarse en ambientes productivos de forma masiva.

En 2018, los principales proveedores basados en la nube lanzaron varios motores, como Amazon Elastic Kubernetes Service (AWS EKS, 2022) y Azure Kubernetes Service (AKS, 2023), aumentando la oferta de clústeres manejados en la nube.

4.3. Funcionamiento de Kubernetes

Una instancia de Kubernetes en funcionamiento se denomina clúster. El cluster se divide en dos partes: el plano de control y las máquinas de cómputo o nodos. Cada nodo puede ser una máquina física o virtual y es su propio entorno el que ejecuta los pods. Un pod es la mínima unidad de computación en kubernetes, estando conformado por uno o más contenedores.

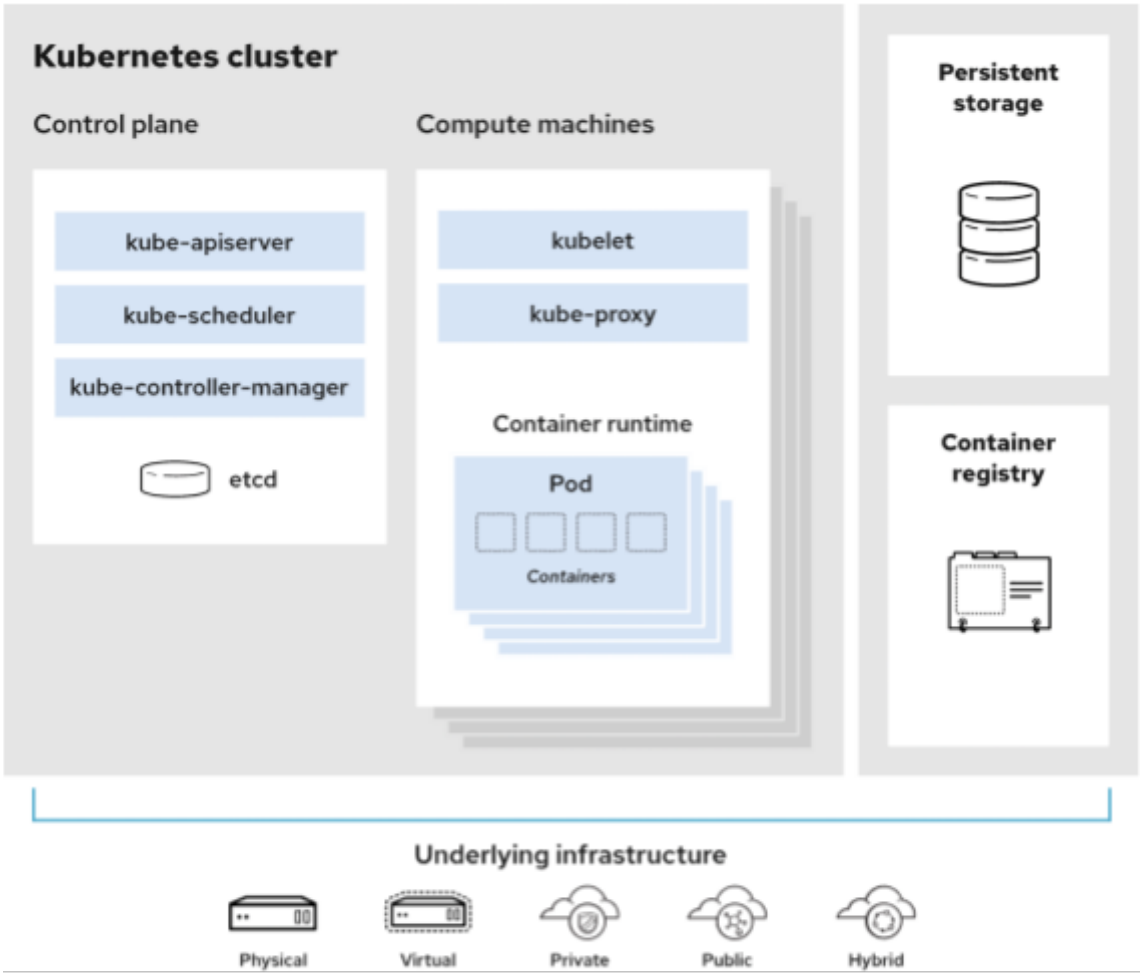


Diagrama de la estructura de Kubernetes

El administrador del clúster setea el estado deseado en forma declarativa, por medio de archivos yaml o json, llamados manifiestos, y el plano de control hará todo lo posible de lograr ese estado deseado. Para esto va a interactuar con los nodos, controlando su sistema operativo y los procesos que corren dentro del mismo.

El estado deseado, creado en base a estos objetos, define las aplicaciones o las cargas de trabajo que deben ejecutarse, junto con las imágenes de contenedores que utilizan, los recursos que deben estar disponibles y otros ajustes similares.

El plano de control chequea para cada pod existente, si la definición del mismo sigue siendo la misma, o si aún es requerido, y si no hay cambios en ese aspecto, chequea que aún responda correctamente. Si alguno de los factores no es correcto, intentará llevarlo al estado deseado, creando nuevos pods que cumplan con dicho estado, y/o destruyendo los pods existentes. El plano de control da órdenes al nodo, para que cree o destruya los procesos según sea necesario.

El proceso normal de actualización de un contenedor puede ser un proceso tedioso, implica seguir alguna estrategia, como correr un contenedor nuevo, esperar a que esté operativo y luego destruir el anterior. Esto se puede automatizar por medio de scripts, pero es algo que Kubernetes maneja en forma automática, proveyendo distintos patrones para llevarlo a cabo, disponibles con simples configuraciones.

En este proceso se ven involucrados una gran cantidad de procesos, para decidir en forma automática en qué nodo correrá cada uno, si hay recursos suficientes para hacerlo, o cómo reagrupar los mismos si hay otro estado posible que lo permita.

Como un nodo es manejado de la misma forma, sin importar su arquitectura o si es virtual o físico, otra ventaja que tiene esta plataforma es la portabilidad. El mismo set de configuraciones funcionará tanto en un clúster creado en la nube, como en servidores virtuales, en servidores bare metal (físicos y dedicados), nubes públicas o híbridas. Cualquier infraestructura tiene la misma estructura, una vez configurado el clúster, siempre existirá el plano de control y sus nodos, y Kubernetes lo maneja de la misma forma.

Esto último permite también que haya implementaciones para computadoras personales, como minikube (Minikube, 2022), o kind (Kind, 2022), que permiten

crear clusters localmente, no aptos para producción, pero muy útiles para verificar y aplicar manifiestos como en cualquier otro clúster Kubernetes.

4.4. Componentes de kubernetes

Para el correcto funcionamiento del cluster, Kubernetes posee una serie de componentes que realizan el trabajo de seteo, testeo, sincronización.

4.4.1. Componentes del plano de control

Los componentes que forman el plano de control toman decisiones globales sobre el clúster (por ejemplo, la planificación) y detectan y responden a eventos del clúster, como la creación de un nuevo pod cuando es necesario.

Estos componentes pueden ejecutarse en cualquier nodo del clúster. Sin embargo, para simplificar, los scripts de instalación típicamente se inician en el mismo nodo de forma exclusiva, sin que se ejecuten contenedores de los usuarios en esos nodos. Adicionalmente, el plano de control se suele ejecutar en varios nodos para garantizar la alta disponibilidad.

- **kube-apiserver**: El servidor de la API es el componente del plano de control de Kubernetes que expone la API de Kubernetes, recibe las peticiones y actualiza acordemente el estado en etcd. La principal implementación de un servidor de la API de Kubernetes es kube-apiserver. Es una implementación preparada para ejecutarse en alta disponibilidad y que puede escalar horizontalmente para balancear la carga entre varias instancias.
- **etcd**: Almacena toda la información del clúster en forma persistente, consistente y distribuida en formato de clave-valor.
- **kube-scheduler**: Este componente observa los nuevos pods que no tienen ningún nodo asignado y selecciona uno donde ejecutarlo. Para decidir en qué nodo se ejecutará el pod, se tienen en cuenta diversos factores: requisitos de recursos, restricciones, afinidad y anti-afinidad, localización de datos dependientes, entre otros.

- **kube-controller-manager**: Este componente se encarga de ejecutar los controladores de Kubernetes. Cada controlador es un proceso independiente, pero para reducir la complejidad, todos se compilan en un único binario y se ejecuta en un mismo proceso.
    - **Controlador de nodos**: es el responsable de detectar y responder cuándo un nodo deja de funcionar
    - **Controlador de replicación**: es el responsable de mantener el número correcto de pods para cada controlador de replicación del sistema
    - **Controlador de endpoints**: construye el objeto Endpoints, que provee una unión entre los Services y los Pods
    - **Controladores de tokens y cuentas de servicio**: crean cuentas y tokens de acceso a la API por defecto para los nuevos Namespaces.
- **cloud-controller-manager**: ejecuta controladores que interactúan con proveedores de la nube. Sólo ejecuta controladores específicos para cada proveedor de la nube. Permite que el código de Kubernetes y el del proveedor de la nube evolucionen de manera independiente. Los siguientes controladores dependen de alguna forma de un proveedor de la nube:
    - **Controlador de nodos**: es el responsable de detectar y actuar cuándo un nodo deja de responder
    - **Controlador de rutas**: para configurar rutas en la infraestructura de nube subyacente
    - **Controlador de servicios**: para crear, actualizar y eliminar balanceadores de carga en la nube
    - **Controlador de volúmenes**: para crear, conectar y montar volúmenes e interactuar con el proveedor de la nube para orquestarlos

4.4.2. Componentes de nodo

Los componentes de nodo corren en cada nodo, manteniendo a los pods en funcionamiento y proporcionando el entorno de ejecución de Kubernetes.

- **kubelet**: Este agente se ejecuta en cada nodo del clúster. Se asegura de que los contenedores estén corriendo en cada pod, toma un conjunto de especificaciones de Pod, llamados PodSpecs, y garantiza que los contenedores descritos en ellos estén funcionando y en buen estado.
- **kube-proxy**: permite abstraer un servicio en Kubernetes manteniendo las reglas de red y ruteando las comunicaciones entre los pods y las redes externas al cluster.
- **Runtime de contenedores**: es el software responsable de ejecutar los contenedores. Kubernetes soporta varios de ellos, como containerd (containerd, 2022), CRI-O (CRI-O, 2023) y cualquier implementación de la interfaz de runtime de contenedores de Kubernetes.

4.5. Objetos de Kubernetes

Para definir el estado deseado, Kubernetes provee una serie de objetos, que son entidades persistentes en el sistema, usadas para representar el estado del clúster. Estos objetos pueden definir qué aplicaciones corren en cada nodo, los recursos que se le asignan, políticas de reinicio, actualización y tolerancia a fallos, entre varios más. A continuación describiremos aquellos objetos básicos de Kubernetes, incluyendo ejemplos simples de manifiestos, que permiten definirlos.

4.5.1. Pod

Es la unidad mínima de computación que se puede crear y gestionar en Kubernetes, es un grupo de uno o más contenedores, con recursos de almacenamiento y red compartidos y la especificación de cómo correr cada contenedor. Los contenedores dentro de un mismo pod corren de la misma

forma que lo harían en una misma máquina física o virtual, comparten IP y puertos, pudiendo comunicarse entre ellos a través de localhost, o usando comunicación estándar entre procesos, como semáforos de SystemV o memoria compartida de POSIX. Estos tienen también acceso a volúmenes compartidos, que se definen a nivel de pod y pueden ser montados en el sistema de archivos de cada aplicación.

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

4.5.2. Service

Es el objeto que describe un servicio del sistema, define una dirección IP, un DNS y una serie de puertos, que permite acceder a uno o más pods, funcionando como balanceador de carga entre ellos, de forma nativa. Los pods son recursos no permanentes, creados y destruídos en base a las necesidades del sistema o ante posibles fallos, por lo que el objeto service ofrece un acceso estable a los pods a lo largo del tiempo. El servicio rutea a los pods con name igual al indicado en el selector

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app.kubernetes.io/name: nginx-demo
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

4.5.3. Namespace

Los espacios de nombres proveen una mecanismo para organizar el clúster en separaciones lógicas que permiten ordenar grandes grupos de objetos (a objetos que admiten estar bajo un namespace) y además limitar los permisos de los usuarios.

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: my-namespace
```

4.5.4. ConfigMap

Un ConfigMap es un diccionario de ajustes de configuración. Este diccionario consta de pares de cadenas clave-valor. Kubernetes proporciona estos valores a sus contenedores. Los Pods pueden utilizar los ConfigMaps como variables de entorno, argumentos de la línea de comandos o como ficheros de configuración en un volumen. Este objeto permite desacoplar la configuración de un entorno específico de una imagen de contenedor, así las aplicaciones son fácilmente portables

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: game-demo
data:
  # property-like keys; each key maps to a simple value
  player_initial_lives: "3"
  ui_properties_file_name: "user-interface.properties"
  #
  # file-like keys
  game.properties: |
    enemy.types=aliens,monsters
    player.maximum-lives=5
  user-interface.properties: |
    color.good=purple
    color.bad=yellow
    allow.textmode=true
```

4.5.5. Secret

Los objetos de tipo Secret en Kubernetes permiten almacenar y administrar información confidencial, como contraseñas, tokens y llaves ssh. Poner esta información en un Secret es más seguro y más flexible que ponerlo en la definición de un Pod o en un container image.

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4=
  password: MWYyZDFlMmU2N2Rm
```

4.5.6. Volume

Los archivos dentro de un contenedor son efímeros, y desaparecen en cuanto un pod termina. Cuando se tiene la necesidad de compartir archivos entre distintos contenedores o incluso pods, o se necesita que los archivos persistan a través del tiempo (duren más allá del tiempo de vida del pod), es necesario definir este tipo de objetos. Existen distintos tipos de volúmenes, que definen la forma de conectarse a distintos tipos de almacenamiento, como discos en la nube, mapeos a archivos o directorios locales en el nodo, discos de red, etc.

Los más representativos para nuestro proyecto son:

- **emptyDir**: Un volumen emptyDir es creado cuando se asigna un Pod a un nodo, y existe mientras el Pod está corriendo en el nodo. Como su nombre lo indica un volumen emptyDir está inicialmente vacío. Todos los contenedores en el Pod pueden leer y escribir los archivos en el volumen, aunque se puede montar en diferente ruta en cada contenedor. Cuando un Pod es removido del nodo por alguna razón, los datos en emptyDir se borran permanentemente

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: registry.k8s.io/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
  - name: cache-volume
    emptyDir:
      sizeLimit: 500Mi
```

- **configMap**: Un ConfigMap provee una manera de inyectar datos de configuración a los pods. Estos datos se pueden referenciar en un volumen de tipo configMap y luego ser consumidos por aplicaciones contenerizadas.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: configmap-pod
spec:
  containers:
    - name: test
      image: busybox:1.28
      volumeMounts:
        - name: config-vol
          mountPath: /etc/config
  volumes:
    - name: config-vol
      configMap:
        name: log-config
        items:
          - key: log_level
            path: log_level
```

- **secret**: Un volumen secret se utiliza para pasar información sensible, como contraseñas, a los Pods. Estos secrets pueden ser guardados en la API de Kubernetes y montarlos como ficheros para usarlos con los pods sin acoplarlos con Kubernetes directamente. Los volúmenes secret son respaldados por tmpfs (un sistema de ficheros respaldado por la RAM) así que nunca se escriben en un almacenamiento no volátil.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
      readOnly: true
  volumes:
  - name: foo
    secret:
      secretName: mysecret
      optional: true
```

- **nfs**: Un volumen nfs permite montar un NFS (Sistema de Ficheros de Red) (NFS, n.d.) compartido en tu Pod. A diferencia de emptyDir que se borra cuando el Pod es removido, el contenido de un volumen nfs solamente se desmonta. NFS puede ser montado por múltiples escritores simultáneamente.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: registry.k8s.io/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /my-nfs-data
```

```
      name: test-volume
  volumes:
  - name: test-volume
    nfs:
      server: my-nfs-server.example.com
      path: /my-nfs-volume
      readOnly: true
```

- Adicionalmente existen drivers que permiten conectar los diferentes filesystems que proveen los diferentes proveedores de nube, en nuestro caso usaremos Elastic Block Store (AWS EBS, 2022), que provee discos que se conectan a un servidor en la nube, y Elastic File System (AWS EFS, 2022), que provee discos que pueden ser montados en varios servidores al mismo tiempo, con un protocolo similar a NFS. Los controladores de Container Storage Interface (CSI, 2022) de Amazon EBS (CSI EBS, 2023) y de Amazon EFS (CSI EFS, 2023) proporcionan una interfaz CSI que permite a los clústeres de Kubernetes que se ejecutan en AWS administrar el ciclo de vida de los sistemas de archivos provistos por Amazon Web Services.

4.5.7. PersistentVolume

Los PersistentVolume (PV) nos ofrecen una API tanto para usuarios como para administradores, lo que nos permite abstraernos de los detalles de cómo se consume y proporciona lo que se almacena. Es un recurso de Kubernetes que representa los volúmenes que tenemos en el cluster, donde se van a definir detalles del backend, como políticas, modos de acceso, tamaños, políticas de reciclaje, etc.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mongodb-pv
  namespace: mimodels
spec:
```

```
  storageClassName: "gp2"
  accessModes:
    - ReadWriteOnce
  csi:
    driver: ebs.csi.aws.com
    volumeHandle: {{.Values.mongodbVolumeId}}
  capacity:
    storage: 100Gi
  nodeAffinity:
    required:
      nodeSelectorTerms:
        - matchExpressions:
            - key: topology.ebs.csi.aws.com/zone
              operator: In
              values:
                - us-east-1a
```

4.5.8. PersistentVolumeClaim

Un PersistentVolumeClaim (PVC) es una solicitud de almacenamiento por parte de un usuario. Los PVC consumen recursos de PV, pueden solicitar modos de acceso y tamaños específicos. Es común que los usuarios necesiten PV con diferentes propiedades, como el rendimiento, para diferentes problemas, para lo cual los administradores de clústeres pueden ofrecer una variedad de PVs, sin exponer a los usuarios a los detalles de cómo se implementan esos volúmenes

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mongodb-pvc
spec:
  storageClassName: "gp2"
  volumeName: mongodb-pv
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 100Gi
```

4.5.9. Controladores y Operadores

Un controlador en Kubernetes se encarga de realizar el seguimiento de uno o más tipos de recursos y se encarga de que se encuentren en el estado deseado. Estos controladores monitorean y miden el estado de un recurso en el clúster para ajustar los parámetros que divergen del estado deseado.

Hay una serie de controladores incluidos dentro del Kubernetes controller manager, y son los encargados de manejar los objetos básicos de la plataforma.

Los controladores pueden ser extendidos para que implementen cualquier funcionalidad.

Un operador es una forma especializada de controlador. Los operadores implementan el patrón de controlador, lo que significa que mueven el clúster a un estado definido, pero están personalizados para aplicaciones específicas. Agregan extensiones de API de Kubernetes a través de definiciones de recursos personalizadas, creando nuevos tipos de objetos que son utilizados por la aplicación que administran. Hay operadores que realizan todo tipo de tareas, entre ellas mapear recursos de nube en objetos de Kubernetes, e incluso crear infraestructura en la nube.

Los proveedores de nube proveen operadores para crear casi cualquier infraestructura lo que permite definir infraestructura como datos, ya que una configuración en un archivo yaml, leída por un operador, permite la creación de bases de datos, servidores y hasta redes.

4.5.9.1. ReplicaSet

Este controlador se encarga de mantener un conjunto estable de pods ejecutándose en todo momento, se usa para garantizar un número específico de pods idénticos.

Si bien es posible manejar ReplicaSets, se recomienda hacer uso de los controladores de más alto nivel que se describen a continuación.

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  # modify replicas according to your case
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
```

4.5.9.2. Deployment

La función de un deployment, es proporcionar actualizaciones declarativas para pods y ReplicaSets. Un deployment declara el estado deseado de los pods, y la cantidad de instancias a mantener. Kubernetes permite diferentes estrategias de despliegue ante los cambios de un deployment, y mantiene una historia de dichas actualizaciones, lo que permite revertir los cambios si así se requiriera.

```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:1.14.2
        ports:
        - containerPort: 80
```

4.5.9.3. StatefulSet

Al igual que un Deployment, un StatefulSet gestiona pods y ReplicaSets, pero este último mantiene una identidad asociada a sus pods, otorgando a cada pod un identificador persistente que mantiene a lo largo del tiempo. Esto es útil para aquellas aplicaciones que necesitan identificadores de red estables, almacenamiento persistente exclusivo, despliegue y escalado ordenado, entre otras.

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  selector:
```

```
    matchLabels:
  app: nginx # has to match .spec.template.metadata.labels
  serviceName: "nginx"
  replicas: 3 # by default is 1
  minReadySeconds: 10 # by default is 0
  template:
    metadata:
      labels:
      app: nginx # has to match .spec.selector.matchLabels
    spec:
      terminationGracePeriodSeconds: 10
      containers:
      - name: nginx
        image: registry.k8s.io/nginx-slim:0.8
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: "my-storage-class"
      resources:
        requests:
          storage: 1Gi
```

4.5.9.4. DaemonSet

Un Daemonset garantiza que cada nodo ejecute una copia de un pod, si se añaden más nodos al clúster, se añadirán nuevos pods a los mismos en forma automática. Es posible usar selectores para correr pods en ciertos nodos seleccionados.

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
  labels:
```

```
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      tolerations:
      - key: node-role.kubernetes.io/control-plane
        operator: Exists
        effect: NoSchedule
      - key: node-role.kubernetes.io/master
        operator: Exists
        effect: NoSchedule
      containers:
      - name: fluentd-elasticsearch
        image: quay.io/fluentd_elasticsearch/fluentd:v2.5.2
        resources:
          limits:
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
        volumeMounts:
        - name: varlog
          mountPath: /var/log
      terminationGracePeriodSeconds: 30
      volumes:
      - name: varlog
        hostPath:
          path: /var/log
```

4.5.9.5. Job

Este controlador ejecuta uno o más pods, y se asegura de que un número específico de ellos termina en forma satisfactoria. Si un pod falla, lo lanza nuevamente y a medida que los pods terminan correctamente, se incrementa un contador, que al llegar a la cantidad solicitada, da fin al Job y los pods que se encuentren corriendo, si los hubiera.

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    spec:
      containers:
      - name: pi
        image: perl:5.34.0
        command: ["perl","-Mbignum=bpi","-wle","print bpi(2000)"]
      restartPolicy: Never
  backoffLimit: 4
```

4.5.9.6. CronJob

Este controlador crea Jobs con un cronograma establecido, a intervalos regulares. Un objeto CronJob es como una línea de un archivo crontab (Crontab, 2018) de linux. Ejecuta un trabajo de forma periódica según un horario programado escrito en formato Cron.

```yaml
apiVersion: batch/v1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "* * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: hello
            image: busybox:1.28
            imagePullPolicy: IfNotPresent
            command:
            - /bin/sh
            - -c
            - date; echo Hello from the Kubernetes cluster
          restartPolicy: OnFailure
```

4.6. Despliegue de aplicaciones

Para realizar el despliegue de cualquiera de los objetos mencionados, Kubernetes provee un comando llamado kubectl que permite realizar acciones de diversas formas.

La forma más directa, es realizar un comando que cree el objeto deseado, especificando todas sus propiedades como parámetros en la línea de comandos.

```
# create a CronJob that prints "Hello World" every minute
kubectl create cronjob hello --image=busybox:1.28
  --schedule="*/1 * * * *" -- echo "Hello World"
```

Esto se torna tedioso y más difícil de repetir cuando se desean especificar parámetros más complejos, por lo que se proporciona una forma de escribir el manifiesto en formato yaml, y aplicarlo directamente con el comando:

```
# create resource(s)
kubectl apply -f my-manifest.yml
```

Si bien es posible instanciar un pod en forma directa, con un manifiesto simple, esto no es lo ideal en un ambiente productivo, ya que los pods son en su esencia efímeros, Kubernetes se toma la libertad de destruir un pod y relanzarlo en los casos que sea necesario, por necesidad de espacio o procesamiento. En un ambiente productivo se debe emplear controladores como Deployment, StatefulSet, etc. Estos aseguran que los pods correrán con la cantidad de instancias deseadas, y permiten especificar métodos de escalado.

Helm (Helm, 2022), es un proyecto de código abierto adoptado por la CNCF y mantenido por la comunidad, que permite manejar paquetes Kubernetes. Estos paquetes son denominados *charts* y son colecciones de archivos que describen

un conjunto de recursos de Kubernetes. Luego de empaquetar los recursos relativos a una aplicación en un *chart*, se puede instalar o desinstalar con un simple comando.

Estos *charts* son ampliamente usados, y existen repositorios públicos donde los miembros de la comunidad CNCF publican sus implementaciones en esta forma, lo que permite tener instalaciones estándares, simples y consistentes con sólo ejecutar un comando.

Al utilizar git como repositorio de control de versiones del código, y seguir las metodologías de DevOps, fue natural la implementación de GitOps (Schlesinger, 2022), un conjunto de prácticas que permiten a los desarrolladores realizar más tareas relacionadas infraestructura basada en código y procedimientos operativos que utilizan Git como la única fuente de información y el mecanismo de control para crear, actualizar y eliminar arquitectura del sistema. Se utilizan solicitudes de mezcla de cambios de Git para verificar e implementar automáticamente modificaciones en la infraestructura del sistema.

Como alternativas para la automatización de las tareas de despliegue siguiendo la filosofía GitOps, existen varias herramientas, dentro de las cuales destacan:

- Argo CD (Argo CD, 2022) es una herramienta de despliegue continuo que funciona en forma declarativa. Se instala en un clúster, y observa uno o varios repositorios de git, en busca de cambios, ante un cambio en la rama observada, se dispara el cambio en forma automática. Esta herramienta provee una interfaz gráfica muy amigable, que permite ver el estado de sincronización de los charts y tomar medidas como forzar sincronización, eliminar algún componente, etc.
- Helmfile (Helmfile, 2022) es una herramienta para aplicar una serie de *charts* de Helm en un orden determinado. En Helmfile se define una serie de archivos o *helmfiles*, que hacen referencia a otros *helmfiles* o *charts* de Helm, por lo que se reusaron todos los *charts* creados

anteriormente, y el cambio de herramienta necesitó de muy poco esfuerzo.

4.7. Orquestación

El tipo de organización que realiza Kubernetes es denominada "orquestación", al igual que una orquesta se basa en el "director" u "orquestador". Como k8s agrupa las cargas de computación en pods escalables, alcanzables a través de servicios, es normal considerar a k8s como un orquestador de microservicios. Las interacciones de microservicios requieren la orquestación de varios aspectos, para permitir una adecuada respuesta al cliente final, como por ejemplo utilizar balanceadores que se configuren al cambiar la escala de pods, auto sanado en caso de la falla de contenedores en un pod, limitar el flujo de tráfico hacia un contenedor cuando este no esté disponible para hacerlo, entre varias más.

Kubernetes provee entonces mecanismos para que los contenedores dentro de un pod puedan constatar el estado de salud, su disponibilidad e incluso determinan de qué forma se espera que reaccionen ante ciertas señales.

Al lanzar un contenedor, k8s comenzará a realizar pruebas periódicas para verificar que el mismo esté listo para recibir pedidos. Estas pruebas son denominadas **readiness probe**, e indican que el contenedor esté "listo" para recibir tráfico. Si no responde adecuadamente en un tiempo determinado, intentará eliminarlo y volver a crearlo.

Una vez que un contenedor dentro de un pod se encuentre "listo", el k8s comenzará a constatar que el servicio continúe saludable, y en caso contrario intentará realizar el mismo proceso de recrear el contenedor. Estas pruebas son denominadas **liveness probe**.

Tanto las pruebas readiness y liveness proveen métodos alternativos para su implementación, como responder a una solicitud HTTP, conectarse a un puerto TCP o responder de forma exitosa a la ejecución de un comando de shell. Esto

se configura en cada contenedor del pod, siendo independientes los métodos para readiness y liveness.

Son varios los escenarios en los cuales una carga de trabajo debe ser eliminada, para lo cual k8s utilizará la señal SIGTERM (Kerrisk, 2022) para notificar al contenedor, y esperar que este reaccione con un apagado ordenado, esto es guardando datos o cerrando conexiones si es necesario, notificando a sus procesos hijos y evitando procesos huérfanos. Si esto no ocurre durante un lapso de tiempo, el orquestador puede eliminar el proceso de una forma inmediata utilizando la señal SIGKILL (Kerrisk, 2022).

Esta comunicación permite que la manipulación de procesos se haga en forma ordenada, y asegura que el estado deseado del cluster sea correcto.

Los objetos service de k8s, ofrecen una simplificación al balanceo entre la comunicación entre las cargas de trabajo. Esto se debe a que un service en k8s expone un DNS e IP que son independientes de los pods que se encuentren detrás de él. Los objetos service crear otros objetos de k8s llamados endpoints que son creados para cada pod que pertenezca al mismo namespace donde se crea el service, que matcheen los labels mencionados en el service. De esta forma, al escalar un deployment o stateful set, se crearán tantos pods como sea necesario y de esta forma el servicio balanceará inmediatamente el tráfico a cada contenedor del pod que esté en estado listo.

En ciertas situaciones, un deployment deberá cambiar su escala en base a métricas, para lo que Kubernetes provee al objeto Horizontal Pod Autoscaler, gracias al cual una carga de trabajo podrá crecer elásticamente de forma autónoma en respuesta a determinada métrica. El diseño de los servicios mencionado en el párrafo anterior, hace que todo se mantenga funcionando de igual forma, solamente con la salvedad que la escala será automática.

4.8. Soluciones en la nube

La configuración de un clúster depende de servidores que funcionen como plano de control y como nodos, los cuales si bien pueden ser computadoras físicas, instaladas en un centro de cómputos propio, acarrea las complejidades estudiadas en el capítulo 2, cuando describimos Cloud Computing. La mayor adopción de Kubernetes en el mundo, llevó a los proveedores de nube a crear sus propios servicios, donde proveen la infraestructura necesaria y la configuración y mantenimiento del plano de control, permitiendo al usuario final centrarse en el estado deseado.

A continuación se describen los principales proveedores con sus características principales.

4.8.1. Google Kubernetes Engine (GKE)

Google fue el creador de Kubernetes, y como tal, también el primero en proveer un servicio manejado en la nube, siendo lanzado al mercado en 2015.

Es el que posee más características y capacidades de automatización, como por ejemplo la actualización automática del plano de control o la reparación automática de nodos.

4.8.2. Azure Kubernetes Service (AKS)

Azure es el servicio de nube de Microsoft, en sus orígenes tenía un servicio de manejo de contenedores llamado Azure Container Service (ACS), el cual no solo proveía Kubernetes, sino también otros manejadores de contenedores llamados Apache Mesos y Docker Swarm. En 2018, ante el incremento de popularidad de Kubernetes, reemplaza este servicio con AKS (AKS, 2022).

En su modelo de negocios, no se paga por el plano de control, sólo por los nodos que esté manejando.

Es el que mejor integra con los productos Microsoft, por lo que si se trabaja con herramientas de Microsoft como VS Code, Active Directory, Helm, entre otros, la integración será más simple.

4.8.3. Amazon Elastic Kubernetes Service (EKS)

Amazon, en forma similar a Microsoft, proveía inicialmente el servicio ECS, comentado en el capítulo 3, como manejador de contenedores, y en 2018 lanza el servicio EKS, sin discontinuar ECS, ya que los servicios tienen distintas características, con diferentes consumidores finales.

Como en el caso de Amazon ECS, Amazon EKS ofrece opciones de manejo para los nodos de trabajo, con EC2 o Fargate, siendo el primero más complejo de manejar, pero permitiendo una mejor optimización de recursos, y el segundo más simple pero a un mayor costo.

4.9. Conclusión

Luego de analizar esta alternativa, y ver la versatilidad que ofrece, no nos dejó dudas que era el camino a seguir. Si bien había un desafío importante en la configuración y migración de los servicios y aplicaciones, el gran crecimiento de la plataforma y de la necesidad de escalabilidad, hizo que la decisión sea clara.

La inversión de tiempo y esfuerzo en migrar a la nueva infraestructura, iba a decantar en ahorros de dinero y tiempos de mantenimiento de una arquitectura más compleja de administrar.

5. Solución final

Como se explicó en el capítulo anterior, se decidió migrar la infraestructura a Kubernetes, y dada la experiencia del equipo sobre AWS, la decisión fue la de adoptar el servicio EKS. En este capítulo describiremos cómo se realizó este proceso y las decisiones que se tomaron para llevarlo a cabo.

5.1. Esquema de infraestructura propuesto

El cambio fundamental de infraestructura radica en su gestión. Anteriormente cada aplicación y servicio se gestionaban de forma independiente en varios servicios ECS, con la nueva propuesta la gestión se centraliza en un solo clúster de Kubernetes manejado por EKS, simplificando los despliegues heterogéneos.

Los backing services, tanto los gestionados por AWS como es el caso de las bases de datos, como servicios desplegados en instancias EC2, serían mantenidos como hasta ahora, al menos en una primera etapa ya que lo fundamental era poder concentrar el esfuerzo en adaptar las aplicaciones ya contenerizadas, y de esta forma realizar un mejor aprovechamiento de los nodos, y en consecuencia abaratar costos.

Se generaron espacios de nombres para los procesos globales a toda la plataforma, y para cada aplicación en particular, y los procesos se distribuyeron a través de dichos espacios de nombres, para tener un mejor control.

En el siguiente diagrama se muestra la nueva infraestructura definida para la aplicación descrita en el capítulo 3, sólo que en este gráfico marcamos con un recuadro naranja las componentes que se despliegan en kubernetes. De esta forma es posible evidenciar la equivalencia entre ambas, haciendo hincapié en el proceso de migración y no en el agregado de nuevas aplicaciones o servicios.

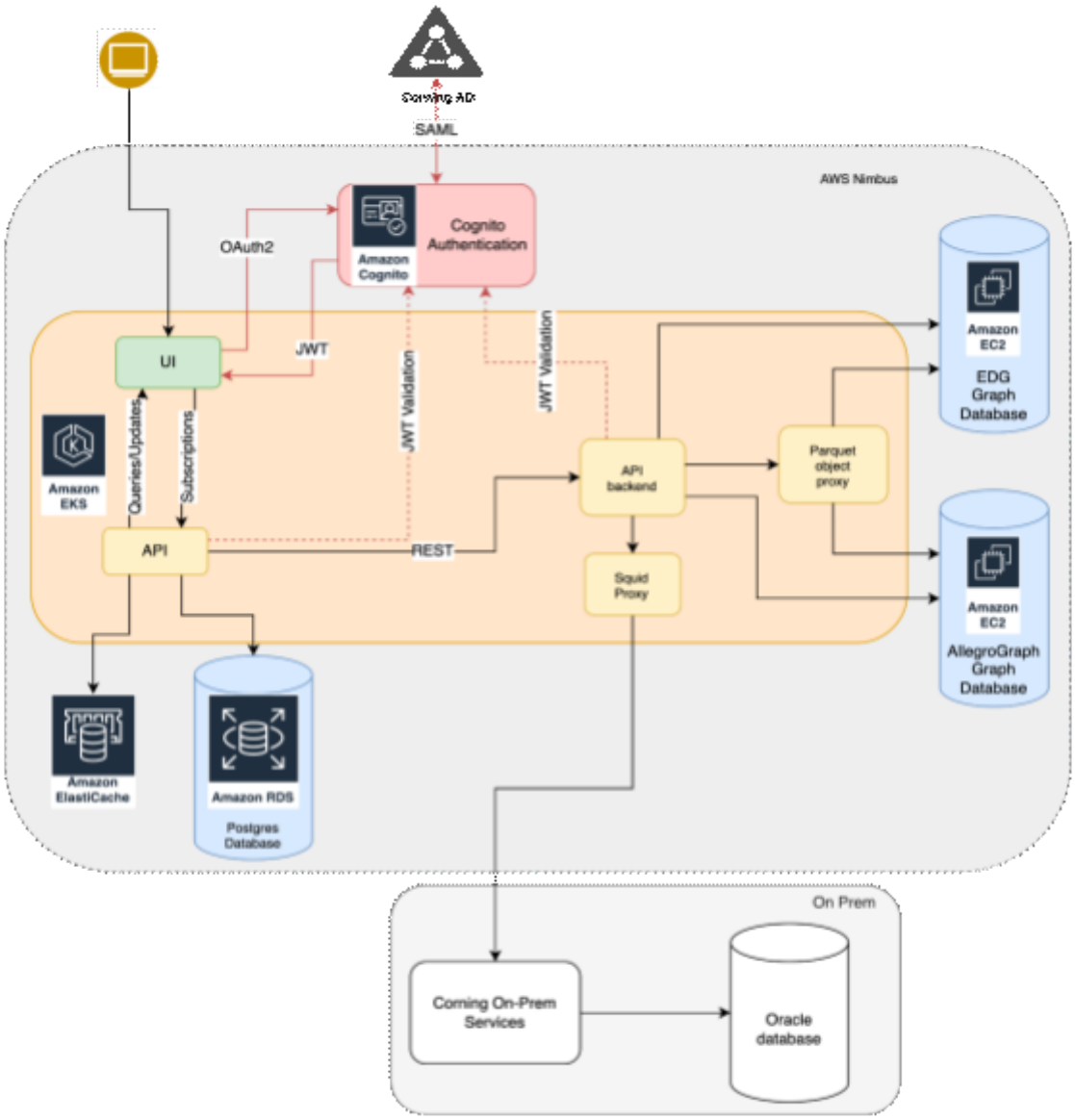


Diagrama propuesto para plataforma con EKS

El gráfico mantiene los backing services intactos, permitiendo que el cambio de infraestructura sea lo más transparente posible para el funcionamiento de las aplicaciones y servicios.

5.2. Integración y despliegue continuos

Cuando se comenzó con el análisis del cambio de infraestructura, siempre estuvo claro para el equipo mantener terraform como herramienta de IaC, siendo necesario el desarrollo de un nuevo módulo, que creara el clúster EKS con las características propias de nuestro proyecto.

Debido a que nuestro proyecto original creaba y configuraba los backing services además del cluster ECS, la nueva infraestructura no considera su creación, porque deben ser reutilizados en nuestras pruebas con kubernetes desde EKS. La transición sería adaptar el código para que en vez de crear un cluster ECS se cree el cluster EKS. Desde una perspectiva simplificada, durante las pruebas tenemos la infraestructura previa y un módulo que creará además un cluster EKS.

Más en detalle sobre la organización de la infraestructura como código, se mantiene un repositorio en Git con el código para la creación de la infraestructura de base. Luego se utilizan otros dos repositorios que dependen de la infraestructura de base para el despliegue de las aplicaciones originales, y el otro para las nuevas aplicaciones que se han desarrollado durante la elaboración del presente documento.

La mayor diferencia entre los despliegues en kubernetes respecto de ECS, es la cantidad de herramientas disponibles. Se analizaron y probaron diferentes alternativas, de las cuales hemos decidido utilizar Helm (Helm, 2022) y Helmfile (Helmfile, 2022), ambas muy parecidas. Comenzamos por empaquetar todas las aplicaciones y recursos necesarios en *charts* de Helm. La principal ventaja que ofrece Helm es la parametrización de los despliegues a través de archivos separados que permiten inyectar valores que son usados para interpolar en los recursos que serán utilizados. Esto simplifica sustancialmente los despliegues en diferentes ambientes. Por su parte Helmfile, permite agrupar el despliegue de varios helm charts en un único archivo de configuración, evitando el uso de charts basados en dependencias (Helm Dependency, 2022).

Al tener los Helm charts de cada componente, sumado a un helmfile que agrupa cada chart necesario en un despliegue completo para un ambiente, obtuvimos un nuevo esquema de despliegue donde únicamente se necesita:

- Un helmfile que describe cada chart a ser considerado
- Un conjunto de valores necesarios para hidratar los charts del helmfile

Este nuevo paradigma de despliegue ofrecía un ciclo de vida donde es posible:

- Crear nuevos despliegues
- Actualizarlos
- Eliminarlos

El problema es que cada acción antes mencionada debe realizarse por alguien ejecutando el comando helmfile.

De esta forma, cuando tuvimos una versión funcional de todas las aplicaciones corriendo en kubernetes desplegadas con helmfile manualmente, decidimos automatizar el proceso de despliegue con Argo CD (Argo CD, 2022). Este producto resultó la articulación buscada porque nos permitió implementar GitOps (Schlesinger, 2022) casi de forma directa. Sólo fue necesario versionar los helmfile y valores en un repositorio que identifica un despliegue en un ambiente. Luego, Argo CD observa los repositorios cada cierto tiempo (o mediante eventos lanzados por medio de webhooks (Argo CD Webhook, 2022)). Manteniendo el espíritu declarativo propio de kubernetes, Argo CD introduce sus propios Custom Resource Definition (CRD K8S, 2022) (Argo CD CDR, 2022) Applications, que describen cada despliegue y apuntan a un repositorio en Git.

Ya nuestros repositorios de IaC mantenían scripts de CI/CD para actualizar la infraestructura.

Ahora, estos pasos se complementan con Argo CD según muestra el siguiente gráfico.

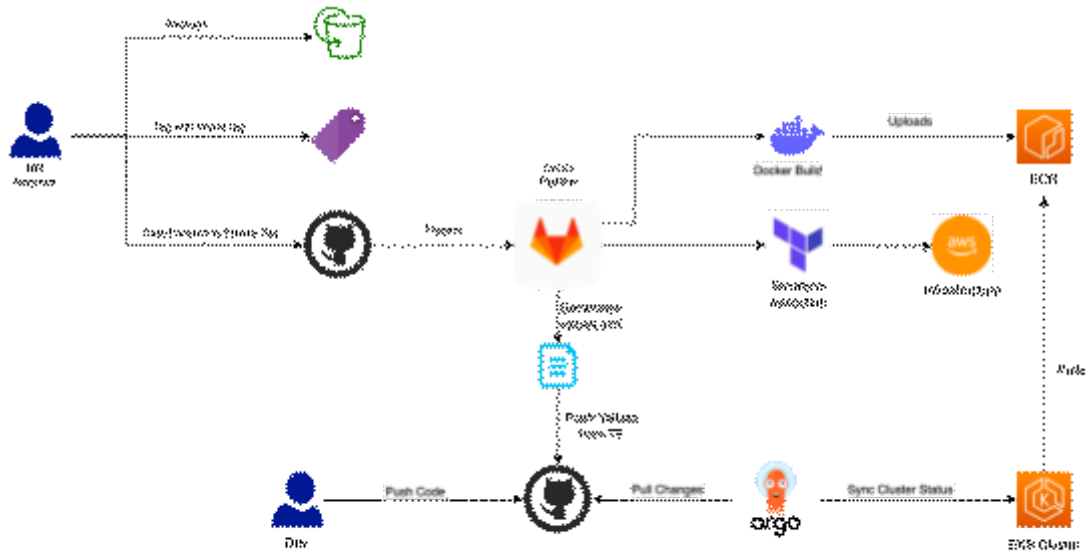


Diagrama implementación de despliegue con Argo CD

Para aplicar los cambios, simplemente se agrega un comando en el script de CI para que sincronice los cambios con los valores actuales.

5.3. Observabilidad

Cualquier plataforma productiva requiere mecanismos que permitan conocer el funcionamiento de cada carga de trabajo. Kubernetes no es una excepción, y por ello propone un monitoreo basado en la estrategia pull (Pull Strategy, 2023), donde las métricas se exponen desde cada contenedor de cada pod, y de esta forma es posible medir el uso de recursos, cantidad de requerimientos, latencias, errores y cualquier dato que sirva para determinar la salud de la plataforma. Estos datos son llamados métricas, y como veremos a continuación, existen servicios que las almacenan y administran, otros que permiten visualizarlas y otros generan alertas.

Por otro lado, cada aplicación genera logs. Los logs de una aplicación muchas veces son fácilmente de seguir, siempre que la aplicación tenga poco tráfico y su escala sea de uno. Cuando el tráfico crece y la escala supera la

individualidad, seguir los logs no es tarea simple. Por ello, siguiendo los lineamientos de los 12 factores, los logs simplemente se escriben en la salida y error estándar. Veremos que estos logs pueden ser manipulados en kubernetes para ser enrutados hacia un concentrador de logs que simplifique su almacenamiento, análisis y visualización.

También es importante analizar las trazas (Tracing, 2023) de una aplicación distribuída, pero este punto requiere que las aplicaciones consideren este requisito desde el código, por lo que no consideraremos esta característica en esta instancia.

En el paso inicial de la migración, se mantuvo la forma de acceso a los logs en AWS Cloudwatch para que los desarrolladores no deban adaptarse a nuevas herramientas. Para ello, se utilizó Fluent Bit (Fluent Bit, 2022) para recolectar los logs de cada contenedor y publicarlos en Cloudwatch, por lo que con un simple agregado, se mantuvo el acceso a los logs de la forma habitual.

En cuanto a las métricas, al utilizar contenedores en pods dentro de kubernetes, AWS ya no podía proveer métricas internas sobre uso de cpu, memoria o disco de cada contenedor. Sólo ofrece métricas a nivel de nodo, por lo que buscamos una herramienta que nos dé mayor granularidad para entender, por ejemplo, por qué un nodo se saturaba. Esta herramienta es Prometheus (Prometheus, 2022), una aplicación que permite almacenar y gestionar métricas, generar alertas y, combinándolo con Grafana (Grafana, 2022), se obtiene un panel de visualización muy completo. Prometheus es un proyecto de código abierto, graduado en la CNCF, que ofrece un soporte muy amplio de la comunidad e integraciones con todo tipo de aplicaciones.

Para recolectar y almacenar métricas en Prometheus, una aplicación debe exponerlas mediante exporters. Un exporter simplemente expone un conjunto de métricas de diferentes tipos, que luego Prometheus recoge y almacena en una base de datos temporal.

Para la instalación de Prometheus en kubernetes usamos un *chart* de Helm llamado kube-prometheus-stack (kube-prometheus-stack, 2023), que incluye

una colección de manifiestos de Kubernetes que integran las capacidades de autodiscovery con Prometheus, permitiendo además definir reglas a través de CRD, despliega además Grafana con una serie de tableros prediseñados y un operador que permite una gestión mucho más simple de las configuraciones.

En la instalación inicial, con sus configuraciones de fábrica, ya se tiene acceso a una cantidad de métricas, que permiten visualizar consumos de recursos básicos, y generar alertas.

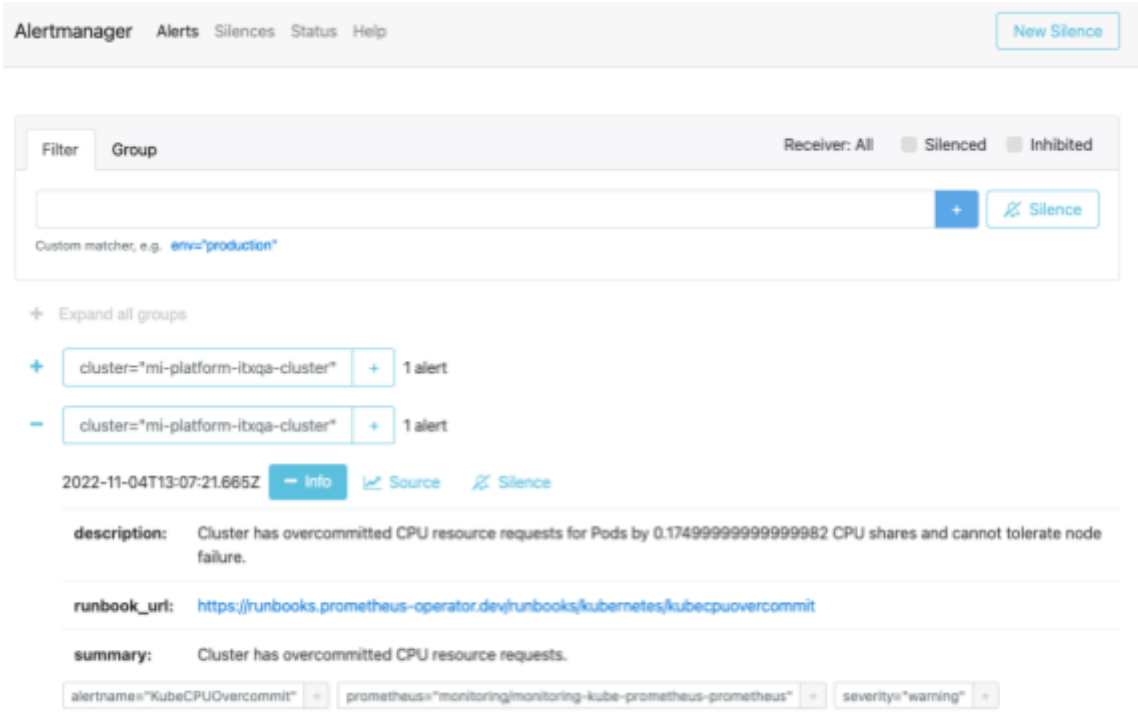
Panel de control de Prometheus

La versión instalada de Grafana ya provee una serie de dashboards muy útiles:

Panel de control de Grafana

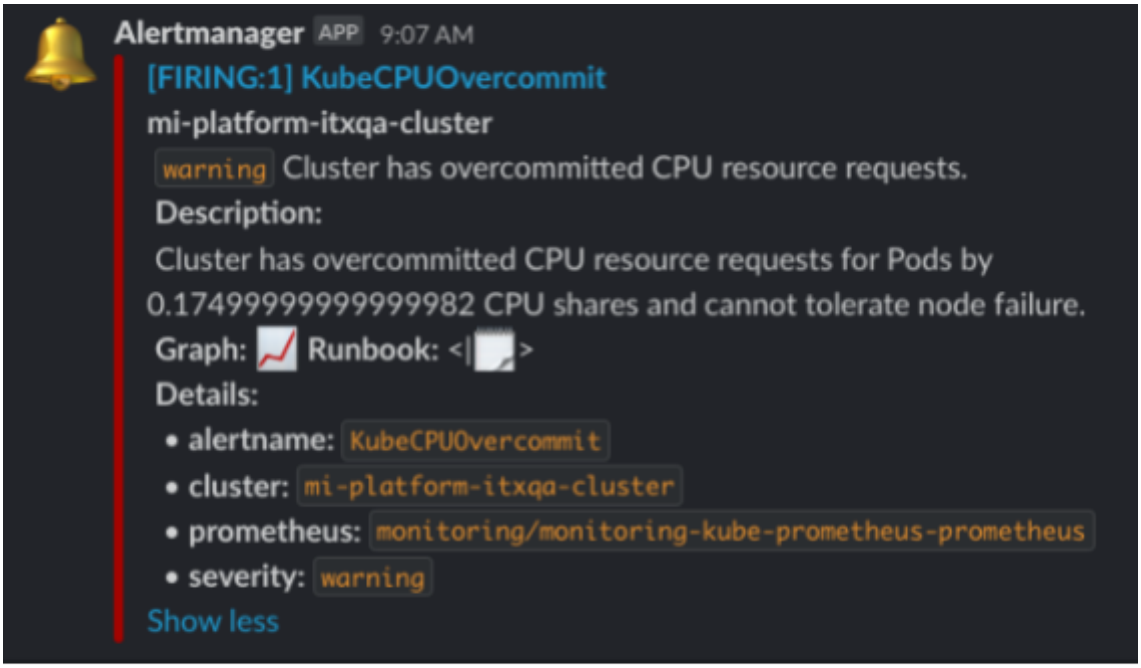
El sistema de alertas en Prometheus se denomina AlertManager (AlertManager, 2022) y su funcionamiento se divide en dos partes. Las alertas se definen desde Prometheus, y las mismas son enviadas al AlertManager, el cual las maneja dependiendo de una serie de reglas que permiten silenciar, inhibirlas, combinarlas y enviar notificaciones mediante diversos medios como email, Jira, Slack y otros sistemas asistidos. Además, AlertManager mantiene una arquitectura tolerante a fallas, para evitar la pérdida de alertas.

Posee un panel de control muy intuitivo que permite visualizar todas las alertas generadas y silenciarlas temporalmente.



Panel de control de Alertmanager

Una vez configurado algún mecanismo de envío de notificaciones, las mismas se recibirán en un mensaje como el mostrado a continuación:



Ejemplo de alerta en Slack

5.4. Auto escalado

La eficiencia de uso de recursos es una de las características más importantes al planificar el cambio planteado. Para conocer el uso real de recursos, es muy importante mantener métricas de cada carga de trabajo. Además de poder visualizar y alertar a partir de las métricas guardadas en Prometheus, es posible automatizar el escalado de determinadas cargas de trabajo acorde a sus necesidades. Este proceso se llama auto escalado y en kubernetes se divide en tres tipos.

5.4.1. Horizontal

La escalabilidad horizontal consiste en cambiar la cantidad de procesos que ofrecen un servicio en base a diferentes métricas. Observando las métricas de un contenedor dentro un pod, será posible aumentar o disminuir la cantidad de

pods satisfaciendo las necesidades del momento. Para este tipo de autoescalado exploramos dos opciones:

- **Horizontal Pod Autoscaling (*HPA*, 2022):** esta funcionalidad es interna y estándar en Kubernetes. La misma se basa únicamente en métricas de cpu y memoria de los pods y ciertos umbrales configurables. Cuando el valor de la métrica baja del umbral mínimo, se modificará el Deployment asociado para que el ReplicaSet disminuya la cantidad de pods, y cuando el valor sea mayor que el umbral máximo, se modificará el Deployment de la misma forma y creará nuevos pods para bajar la carga de cada uno de los existentes. La cantidad mínima y máxima de pods también son configurables.
- **Kubernetes Event-driven Autoscaling (Keda, 2022):** a diferencia de HPA, keda está basado en eventos, permitiendo tomar métricas más complejas como por ejemplo el tamaño de una cola de pedidos (por ejemplo rabbitmq), una consulta a una base de datos SQL, métricas de prometheus y múltiples posibilidades dado que el escalado se basa en **scalers (Keda Scalers, 2022)**. Una diferencia esencial respecto de HPA es que admite una escala de cero pods, pudiendo apagar completamente un servicio si no se utiliza.

5.4.2. Vertical

Un aspecto que muchas veces pasa por alto, es el seteo de los recursos de cada contenedor dentro de un pod. Su definición y existencia es fundamental para un adecuado funcionamiento del pod, pero además, por un correcto desempeño del cluster completo. Al establecer la cantidad de CPU y memoria requeridos, así como los límites de ellos, el scheduler de kubernetes irá distribuyendo los pods en aquellos nodos con disponibilidad. Si no se establecen los requisitos de CPU y memoria, el scheduler considerará correr en un mismo nodo múltiples pods y de esta forma generaría la saturación de recursos pudiendo dejar al nodo como **Not Ready**.

La escalabilidad vertical consiste en cambiar los recursos seteados en un pod de forma dinámica en base a las métricas. Para ello, el **Vertical Pod Autoscaler (VPA, 2023)** es un recurso de Kubernetes que ayuda a calcular los recursos necesarios por un Deployment dado. VPA analiza los pods corriendo y sugiere cambios para dicho Deployment, que adicionalmente pueden ser aplicados en forma automática. Puede combinarse con HPA, ya que VPA puede usarse para sugerir los recursos que serán asignados a los nuevos pods, cuando escale horizontalmente. Es especialmente útil para casos donde los contenedores requieren grandes cantidades de memoria o procesador por períodos de tiempo cortos, lo que permitiría recuperar los recursos para que corran otros procesos cuando no son necesarios.

5.4.3. Autoscaler

Es interesante analizar cómo se vinculan las diferentes estrategias de escalado mencionadas hasta ahora. Específicamente con el HPA, sucede que si se aumenta la escala y no hay nodos suficientes, los nuevos pods quedarán en estado **pendiente** hasta que se libere espacio en algún Nodo. Este hecho, puede ser contemplado en determinados tipo de clusters (especialmente en aquellos clusters autogestionados por proveedores de cloud), donde al existir presión por pods en estado pendiente, puedan crearse nuevos nodos, escalando así los workers del cluster. Hemos evaluado dos alternativas:

- **Clúster autoscaler (Cluster Autoscaler, 2022)**: escala los nodos de nuestro clúster en forma automática, creando nuevos nodos cuando encuentra pods en estado de pendiente, por no tener lugar en los nodos existentes, y chequeando los nodos existentes, permitiendo que cuando han sido subutilizados durante un tiempo determinado, sea posible reacomodar los pods para que corran en otros nodos existentes, y liberar un nodo para su destrucción.
- **Karpenter (Karpenter, 2022)**: es una herramienta que permite definir distintos aprovisionadores y en base a la definición de afinidad en los pods, permite distribuir pods de una forma más eficiente. Karpenter

permite definir qué tipos de máquinas virtuales se van a crear, y elige las más adecuadas para la carga existente, teniendo en cuenta también los costos del proveedor de la nube que se esté usando.

5.5. Balanceadores de carga

Si bien Kubernetes provee a los objetos service que realizan el balanceo de carga entre los pods que se encuentran corriendo, un balanceador de carga en la nube permite adicionalmente manejar una serie de reglas, que permite dirigir el tráfico dependiendo del tipo de solicitud y sus características. Para hacer uso de estas posibilidades, Kubernetes se integra perfectamente con los balanceadores de carga en la nube:

- **Service LoadBalancer**: los objeto service de tipo LoadBalancer permiten mapear un Service determinado a un Load Balancer en la nube. En AWS, se crea un Application Load Balancer (AWS ALB, 2022), o un Network Load Balancer (AWS NLB, 2022), los cuales funcionan como balanceadores de carga de capa 7 o 4 respectivamente.
- **Ingress**: provee balanceo de carga de capa 7 (capa de aplicación), manejado íntegramente desde kubernetes. Para ello se integran con AWS ALB o NLB según sea necesario.

En la nueva infraestructura se utilizó ingress para crear un Application Load Balancer por aplicación, y mediante reglas relacionadas con el nombre de dominio utilizado para acceder a cada servicio o aplicación, se realiza el redireccionamiento al servicio de Kubernetes correspondiente. A su vez estos servicios de Kubernetes distribuyen la carga entre los pods que se encuentren corriendo en cada momento.

La decisión de compartir un Application Load Balancer entre varios servicios estuvo basada en el ahorro, ya que cada uno tiene un costo individual, y un adicional por uso de tráfico de red, ahorrando el costo de instancia y compartiendo el costo de uso.

5.6. Permisos

Kubernetes provee distintos modos de autorización, pero el más usado es el llamado Role-Based Access Control (RBAC, 2022). El mismo es un método para regular el acceso basado en roles de los usuarios individuales dentro de una organización. En este contexto, el acceso es la habilidad de un usuario de realizar una tarea específica, como por ejemplo ver, crear o modificar un archivo.

Un rol de RBAC contiene reglas, que representan permisos. Las reglas tienen una forma aditiva, no habiendo reglas que denieguen permisos. Por definición, todo lo que no está expresamente permitido, no está permitido.

Los roles de usuarios representan los procesos que realizan los empleados de una empresa, por lo que no existe una única división, sino que depende de la organización que tiene cada empresa. Antes de aplicar este método, hay que realizar un análisis exhaustivo de qué tareas realizará cada grupo de usuarios, y en base a eso se definirán los roles necesarios para distinguirlos.

Amazon conecta en forma nativa IAM con EKS y permite que se mapeen roles de IAM con roles de RBAC, de esta forma, si un usuario se encuentra autenticado en AWS, podrá tener los accesos que le correspondan en el clúster de Kubernetes de forma automática.

Estos métodos de autorización se usan para la administración del clúster y sus objetos, manteniendo el mismo esquema de autenticación y autorización a nivel aplicaciones y servicios.

5.7. Costos

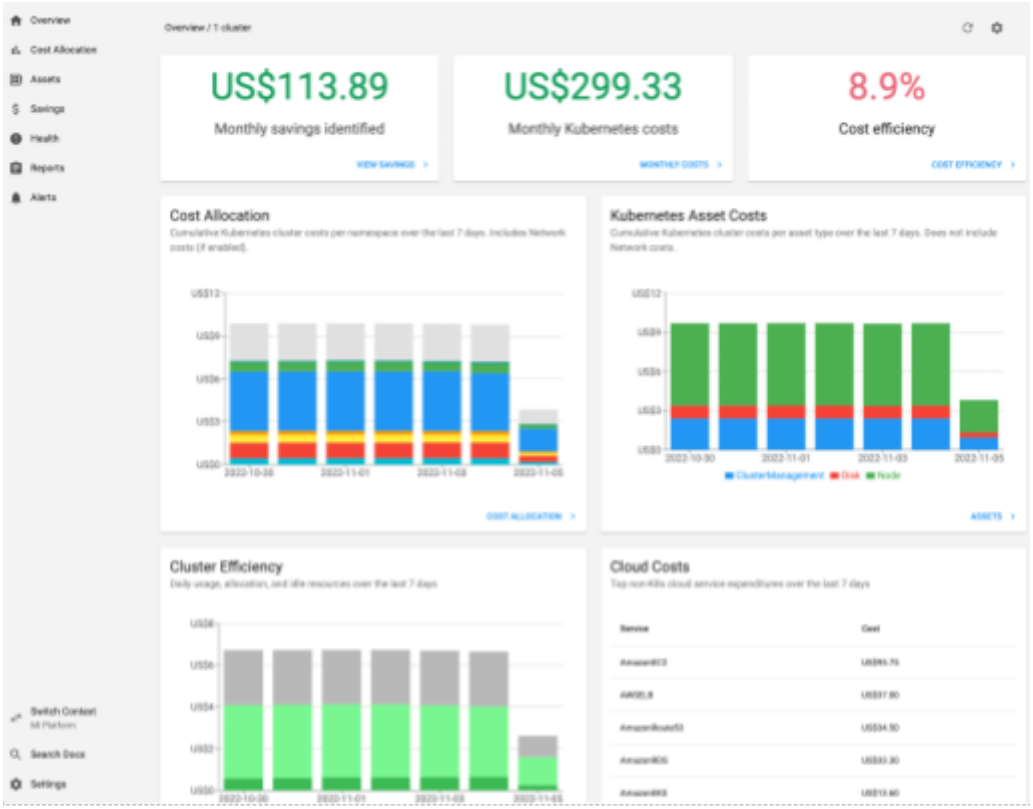
Al sumar aplicaciones a la plataforma, también surgió la necesidad de analizar los costos de la infraestructura utilizada, separando el uso entre los diversos proyectos. Para lograr esto se instaló una herramienta llamada KubeCost (Kubecost, 2022). La misma analiza el uso de recursos de cada objeto de

Kubernetes, y se conecta a AWS para entrecruzar los datos de costo de las instancias utilizadas. Los mismos pueden filtrarse por etiquetas, espacios de nombres o casi cualquier atributo de los objetos. El resultado es una serie de reportes que permiten no sólo ver los costos asociados, sino también una serie de recomendaciones para generar ahorros.

KubeCost comenzó siendo un proyecto de código abierto llamado OpenCost (OpenCost, 2022), soportado por la CNCF, y fuertemente integrada con Prometheus y Grafana. Este proyecto sigue utilizando el proyecto de código abierto para su núcleo principal, pero han agregado funcionalidades muy valiosas en su versión comercial que hacen una buena elección en muchos casos.

Provee un panel de control con gráficos con su configuración de fábrica, lo que permite un uso inmediato.

La pantalla inicial es un resumen que incluye los costos totales, ahorros identificados y la distribución de costos y eficiencia.



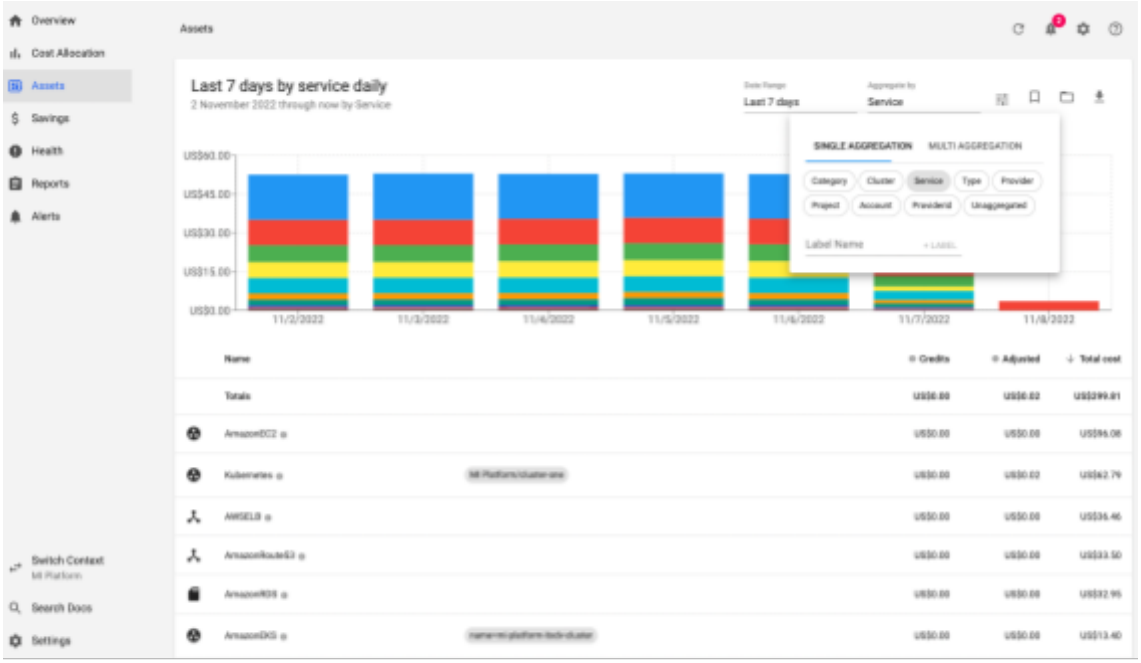
Panel de control de KubeCost

El reporte de distribución de costos provee un detalle del historial de costos, permitiendo observar la información desde distintos puntos de vista, filtrando por espacio de nombre, pod, container, entre otros.

Reporte de distribución de costos

Esto permite identificar fácilmente cuales son los componentes más costosos, para poder encauzar un análisis más detallado y dirigir correctamente los esfuerzos para mejorar costos.

Adicionalmente posee un reporte que muestra los servicios de la nube utilizados, con sus costos asociados. El mismo discrimina los costos por recursos, agrupando los mismos teniendo en cuenta diversos criterios como servicio utilizado, tipo de recurso, categoría o cluster, muy útil en caso de utilizar más de uno.



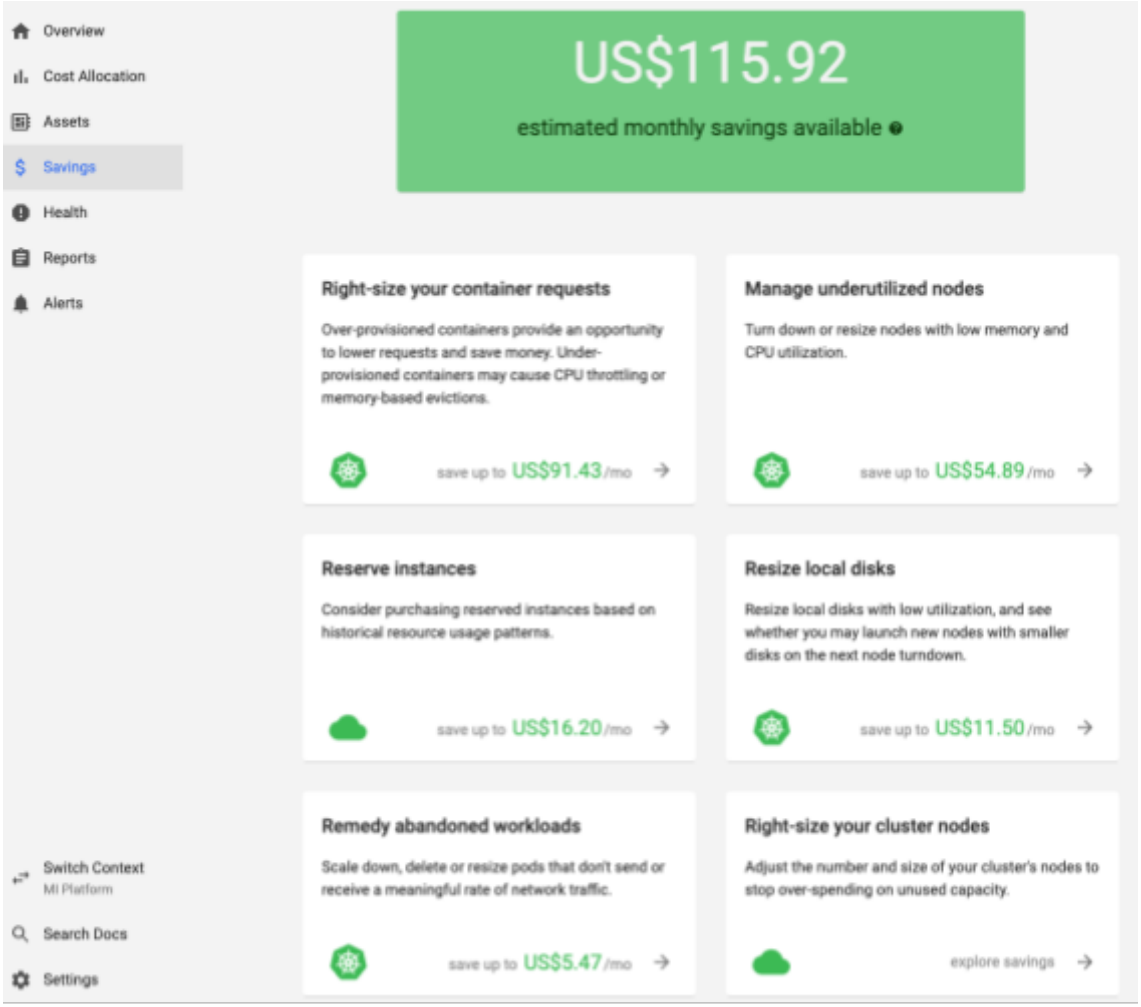
Reporte de costos totales dentro de AWS

Se proveen diversas formas de presentar la información, agrupando por períodos de tiempo.

Diferentes formas de visualizar costos

Finalmente encontramos un reporte con los ahorros sugeridos, los mismos se generan teniendo en cuenta la utilización del cluster, y el costo de los servicios

de la nube. Siguiendo estos consejos básicos se puede conseguir ahorros, sólo modificando tamaño y tipo de instancias de nodos, para poder realizar un mejor aprovechamiento de los recursos.



Reporte de posibles ahorros

5.8. Información sensible

En nuestro sistema, existe una cantidad de valores necesarios dentro de los contenedores que son considerados sensibles, estos son claves de acceso, tokens o datos similares necesarios para la conexión con otros servicios.

Es deseable que esta información sea sólo accesible a un grupo de usuarios reducidos, que muy pocas veces coincide con el grupo de desarrollo, con

acceso al repositorio git. Es por esto que esta información nunca debe ser almacenada en dichos repositorios.

Existen varias alternativas válidas para guardar estos datos, y en nuestro caso se hizo uso del servicio de AWS Secret Manager. Por medio de un módulo de terraform, se crea un *secret*, y se le asignan los permisos de lectura y escritura a los roles que lo necesitan. Este módulo permite crear el valor en forma automática, por lo que en ese caso el valor ya se encuentra asignado y las personas con permisos pueden consultarlo para hacer uso del mismo.

Para su uso dentro de los contenedores de Kubernetes se utiliza un controlador, el cual tiene acceso el servicio AWS Secret Manager y verifica los valores de sus *secrets*, y con dichos valores genera *secrets* en Kubernetes, los cuales son inyectados en los pods como variables de entorno.

Hay que asegurar que los permisos de lectura de los *secrets* en Kubernetes sean asignados a los usuarios indicados, para no permitir accesos a usuarios no autorizados, esto configurado con RBAC como se comentó anteriormente.

5.9. Recuperación de desastres

En una infraestructura de esta magnitud se almacena una cantidad de datos muy valiosa, y por ende existe una dependencia del sistema muy alta. Por esta razón los servicios de almacenamiento de información se manejan en forma redundante cuando es posible, utilizando clusters de servidores que permiten que ante fallos no se pierda la posibilidad de acceso a los datos. A su vez, estos datos deben poder ser restaurados ante la posible pérdida de datos, sea por un error humano o por alguna falla de hardware.

Para el último caso es por lo que se han creado sistemas de backups y procedimientos de restauración, que permitan restablecer el sistema en el menor tiempo posible ante una posible pérdida de datos. En nuestro caso, hemos creado un módulo de terraform, que crea un repositorio de Backups en AWS, usando el servicio AWS Backups (Amazon Backups, 2022), y una serie

de reglas que permiten, con solo agregar un tag en los servicios, tener el backup en forma automatizada cada día.

Este backup asegura que la información no se pierda, y poder levantar una copia de cualquiera de estos servicios en pocos minutos. Luego, dependiendo del problema que haya ocurrido, se utilizarán los pipelines de Gitlab CI para restablecer los servicios, y/o pasos manuales para copiar los datos del backup restaurado, a la nueva instancia.

Dado a que estos casos son realmente excepcionales, y los casos de falla tan dispares, no se desarrolló automatización para recuperación de desastres. Sin embargo, el procedimiento de cómo restaurar el sistema completo quedará documentado en la propia documentación de cada repositorio de IaC.

6. Resultados obtenidos

En este capítulo se mostrarán los resultados obtenidos, teniendo en cuenta diversos aspectos, como costos, performance, simplicidad de despliegue y observabilidad.

6.1. Costos

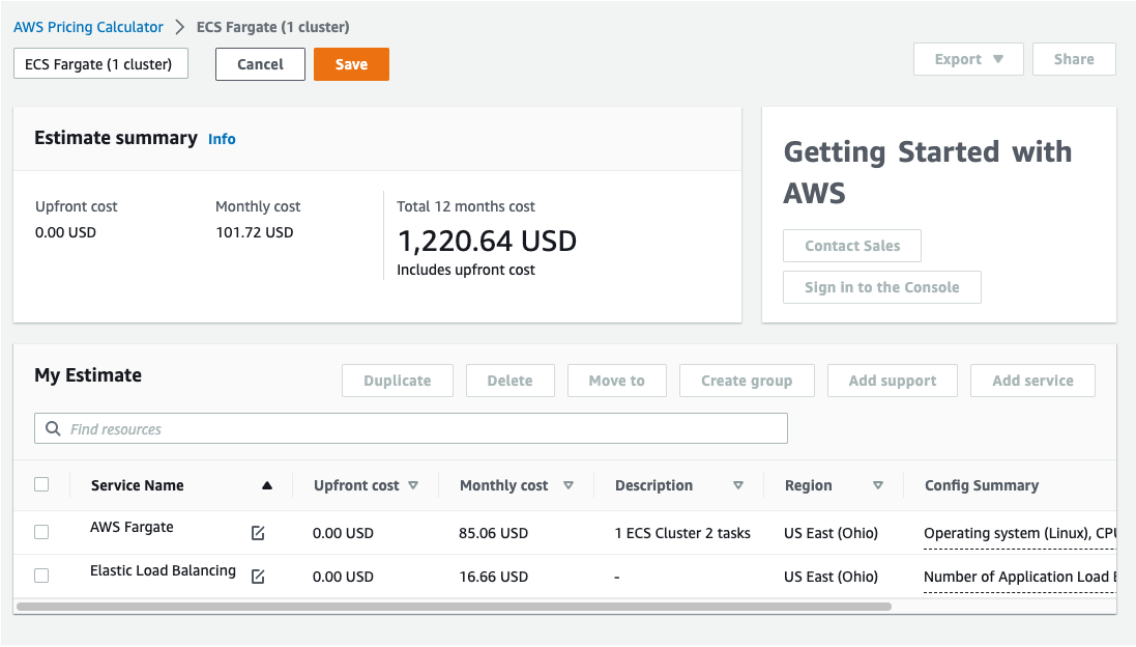
El costo es una de las razones principales que motivaron este cambio, el agregado de nuevas aplicaciones en una arquitectura que crece linealmente ante el incremento de aplicaciones hacía que se previera triplicar los costos en un lapso relativamente corto. En este aspecto comenzamos el proyecto con la hipótesis de que el incremento al migrar a la nueva infraestructura no sería significativo.

6.1.1. Previsión

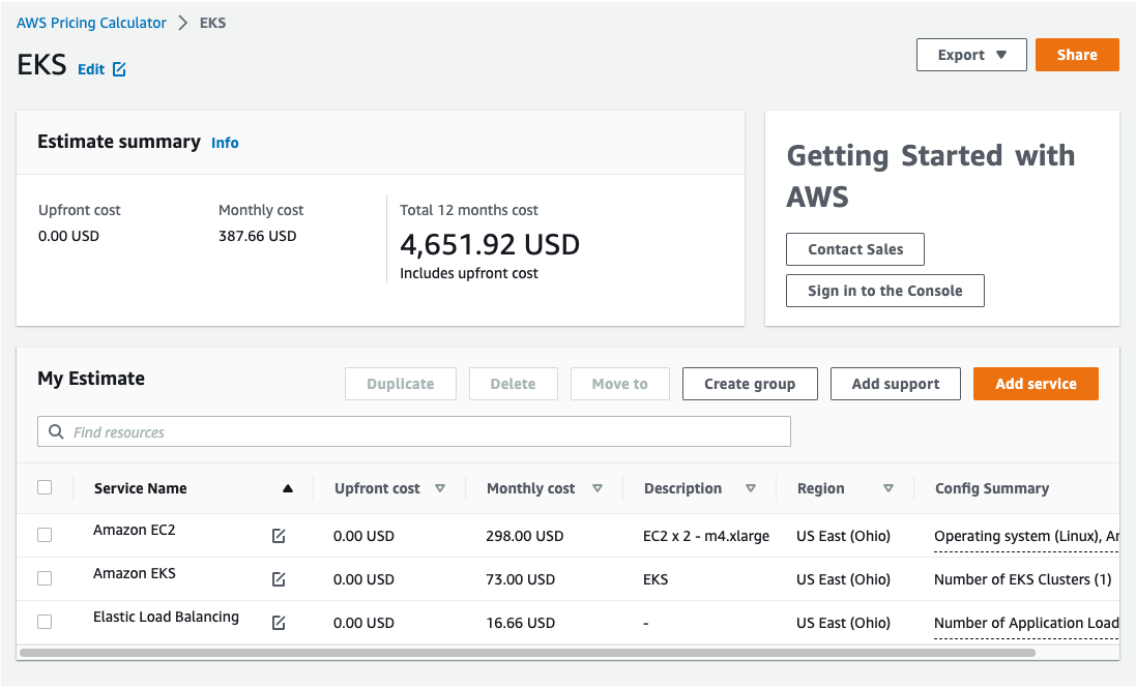
Dado que Amazon provee una calculadora para prever costos asociados a los servicios requeridos, con esta herramienta se realizaron los cálculos de costos asociados a las dos soluciones implementadas con las diferentes infraestructuras, antes y después de esta migración.

Los costos a la solución implementada con ECS se calculan en base a cada servicio o aplicación; en nuestro proyecto utilizamos cuatro instancias.

Cada instancia requiere como mínimo dos tareas corriendo y un Elastic Load Balancer para balancear la carga entre ambas tareas, esto es un cálculo de mínima, ya que si la carga aumenta, el servicio debe agregar tareas, incrementando los costos.



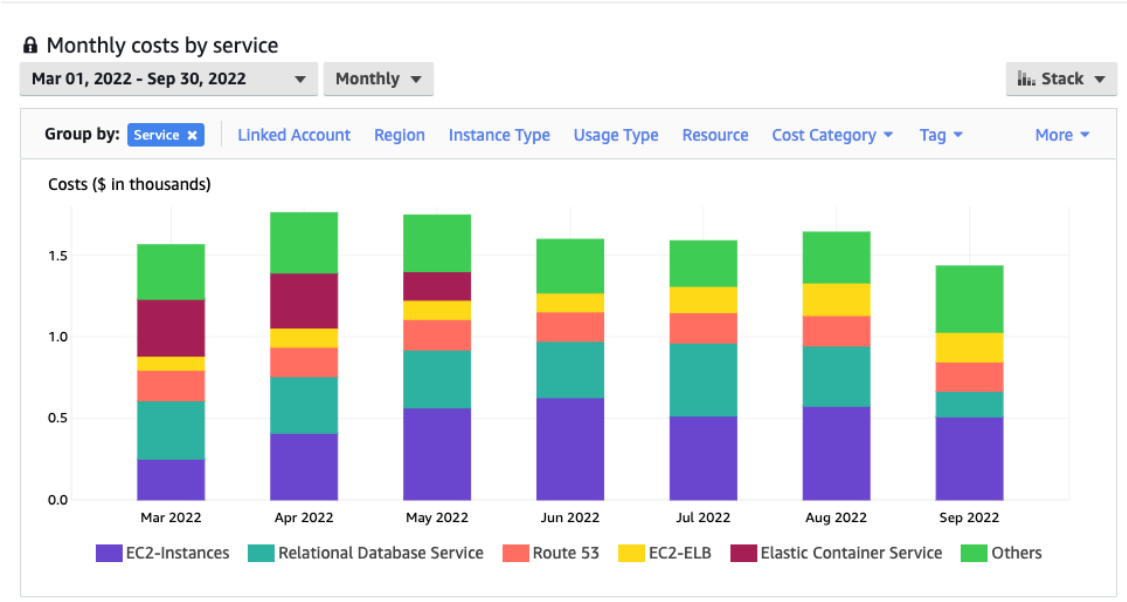
En cuanto a la implementación con EKS, se requiere un cluster EKS, un balanceador de carga que puede compartirse entre los diferentes servicios y al menos dos nodos, que, para realizar una comparativa justa, se utilizaron para el cálculo servidores con el poder de cómputo equivalente a cuatro instancias de las anteriores. En la práctica se utilizan nodos de menor capacidad para los ambientes de desarrollo, ya que las cargas de trabajo son muy inferiores a las de producción.



En la calculadora de Amazon se ve que las cuatro instancias necesarias de ECS tienen un costo de USD 406,88 al mes, mientras que el costo equivalente de la solución EKS es de USD 387,66 al mes, y esta infraestructura permite correr más servicios en la misma, permitiendo un mayor ahorro.

6.1.2. Validación

Luego de unos cuatro meses se verifican los reportes de costos reales de la plataforma, y se analizan los datos obtenidos. Se utiliza la herramienta de Amazon Cost Management, que permite visualizar los costos reales de los servicios utilizados, de diversas formas, con una interfaz amigable.



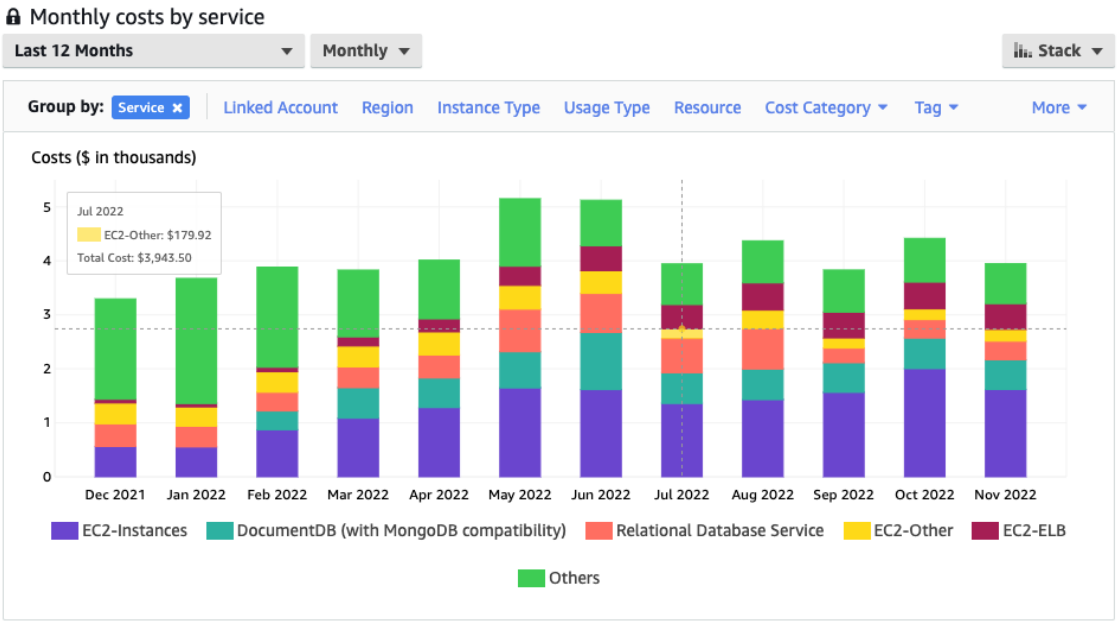
En el gráfico obtenido se verifica claramente que los costos se ven disminuidos desde el primer momento, eliminando el costo de ECS e incrementando el costo de EC2, utilizados por los nodos.

Cabe aclarar que en estos costos se comparte el resto de la infraestructura, que no sufrió modificaciones, pero que en el cluster se encuentran corriendo muchos servicios más, como KubeCost, Prometheus, Grafana, etc.

En resumen, se ve claramente que el costo se vio reducido, y al mismo tiempo se consiguió más valor por este costo.

Adicionalmente se presentan a continuación los costos en producción, luego de agregar las dos nuevas aplicaciones, y triplicar, aproximadamente, el uso de recursos computacionales.

Se ve claramente que el incremento no es lineal, sino que es bastante estable, al escalar los recursos sólo cuando el uso lo requiere, y volviendo a su estado base en cuanto el uso baja nuevamente.



6.2. Performance

Otra de las mejoras notables que hubo con este cambio fue en el escalado horizontal, dado que en la mayoría de los casos hay disponibilidad de cpu y memoria en algún nodo ya creado, los nuevos pods se instancian inmediatamente, en cuestión de milisegundos, cuando el escalado en ecs, demora desde varios segundos, hasta más de un minuto.

Kubernetes mantiene las imágenes de los pods actualizadas, por lo que ante un reinicio no se debe esperar a que baje la última versión.

En nuestro caso, al tener distintas aplicaciones, que se utilizan en forma independiente, también hubo un mejor aprovechamiento de los recursos, ya

que los picos de uso intensivo de una aplicación no coinciden con los de otra, y en pocas ocasiones es necesario levantar una gran cantidad de nodos, ya que los mismos se reutilizan para la aplicación que lo requiera en cada momento.

6.3. Facilidad de despliegue

En el modelo anterior, cada cambio en la definición de un servicio o aplicación implicaba un cambio en los scripts de terraform, y esto es mucho más lento, porque implica cambiar la estructura de un servicio de AWS. En el modelo nuevo, el cambio o agregado de un nuevo servicio sólo implica cambiar o agregar un archivo yaml, y la aplicación es instantánea.

Helmfile también proporciona simplicidad a la aplicación de los manifiestos actuales: con un único comando se permite actualizar todo el estado de cluster.

Otra mejora en los despliegues es que podemos inyectar configuraciones con ConfigMaps, con diversos tipos de manifiestos, y utilizarlas dentro de los pods, como archivos o variables de ambiente, entre otras. Esto agrega versatilidad y simplicidad a la hora de definir imágenes genéricas para los diversos ambientes, permitiendo leer los datos de la forma más conveniente según las tecnologías disponibles.

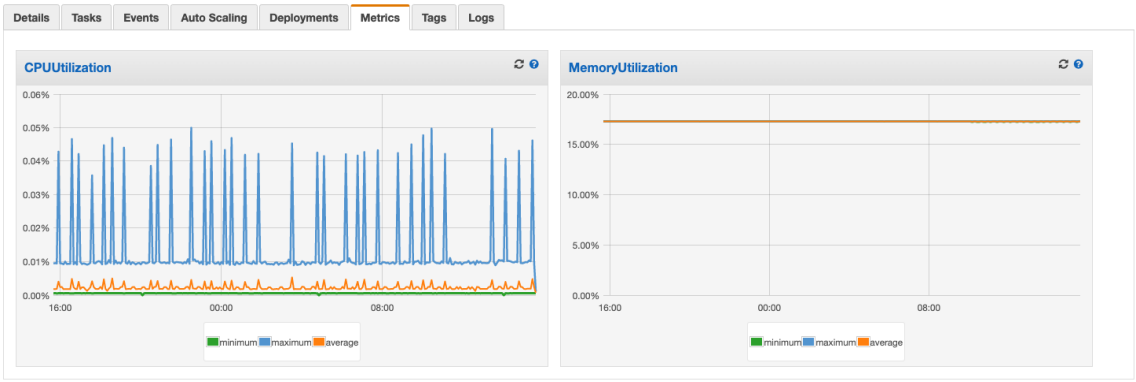
Kubernetes también nos permite realizar distintas estrategias de despliegue, (*rolling*, *recreate*, *ramped slow*, *best effort controlled* y *canary*), permitiendo mejorar la experiencia de usuario de acuerdo con el tipo de cambio implementado, los recursos utilizados o el tiempo que demora un nuevo pod en estar disponible. No siempre la misma estrategia es la mejor para cada despliegue, y elegir en Kubernetes el tipo de despliegue es algo muy sencillo.

6.4. Observabilidad

6.4.1. Recursos

El monitoreo de recursos ha sido otra gran mejora en todo sentido. En ECS teníamos Cloudwatch, lo que permitía visualizar en forma individual para cada servicio información de memoria y CPU, esta información se veía en forma aislada y era difícil de tener una visión general de la plataforma completa.

El siguiente es un ejemplo de visualización en CloudWatch de un servicio corriendo en ECS.



En el nuevo modelo, al poder correr Prometheus con Grafana, conseguimos paneles de control diversos, que muestran el estado general del cluster, con una cantidad importante de información valiosa.

El siguiente es un ejemplo de un panel de control en la nueva plataforma.

Estos paneles de control se aplican con manifiestos, por lo que son instalados automáticamente con el despliegue principal.

6.4.2. Alertas

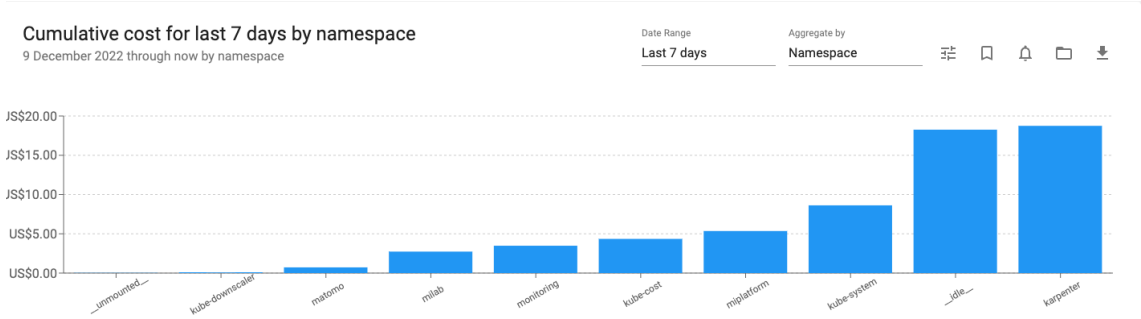
Las alertas que provee CloudWatch en el modelo anterior funcionaban correctamente, la desventaja principal que poseía era que, al no encontrarse centralizadas, eran difíciles de chequear.

Prometheus provee un sistema de alertas centralizado, por lo que podemos revisar un único panel de control en búsqueda de problemas, y configurar diversas formas de comunicación ante alertas críticas, en nuestro caso, recibiendo alertas en Slack, en un canal dedicado para esto.

6.4.3. Costos

En cuanto al monitoreo de costos, la centralización de información que posee Kubernetes permite mejoras a niveles muy altos. Con la ayuda de Prometheus. KubeCost nos provee el seguimiento de costos a nivel de cada proceso, namespace, nodo, etc. El servicio provee un nivel de granularidad y detalle muy grande, que permite no sólo visualizar claramente los costos de cada servicio o aplicación, sino también posibles problemas en la asignación de recursos.

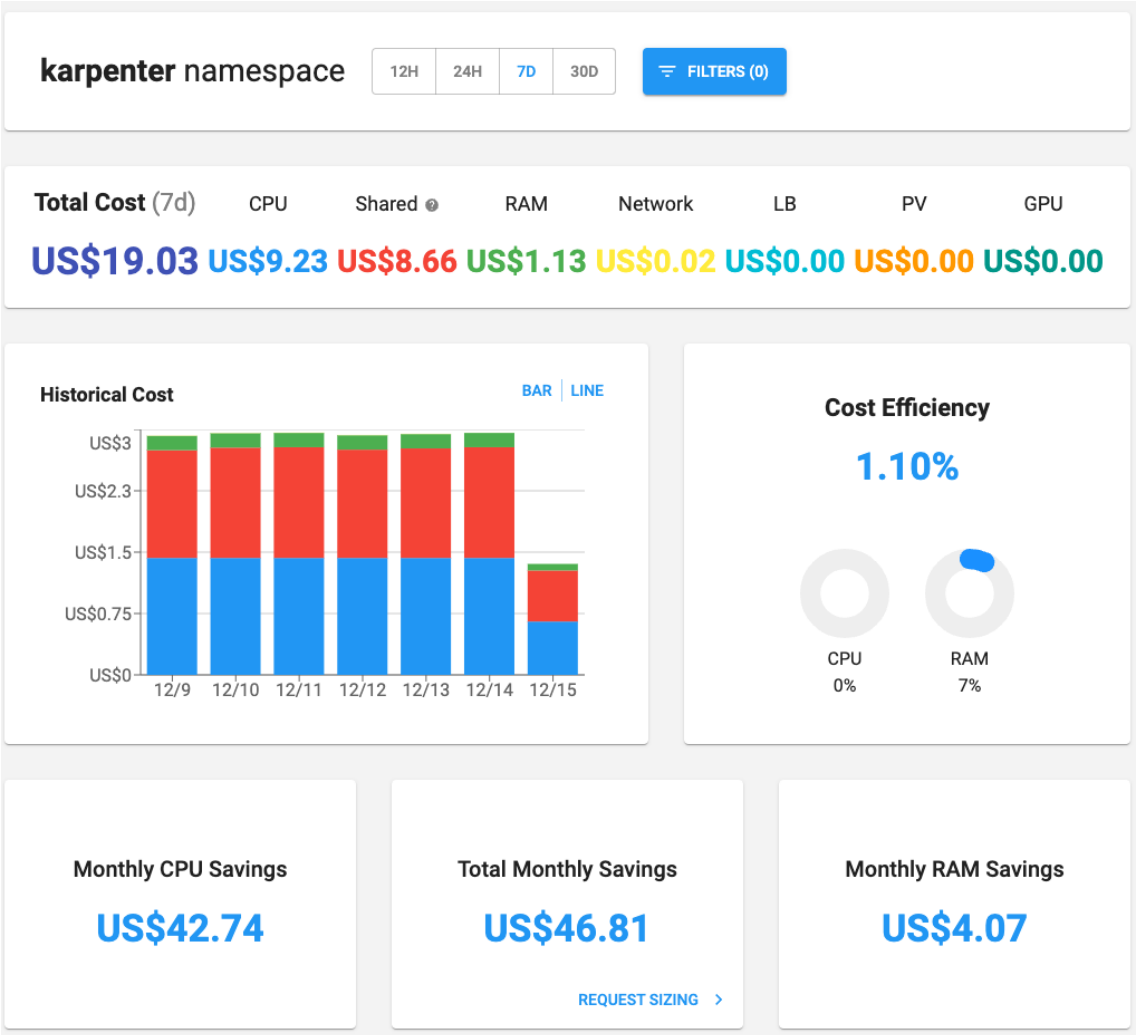
El siguiente es un ejemplo de uno de los reportes básicos de KubeCost, en el que vemos los costos asociados a la asignación de recursos de cada servicio, relativo al costo de los nodos asignados al clúster.



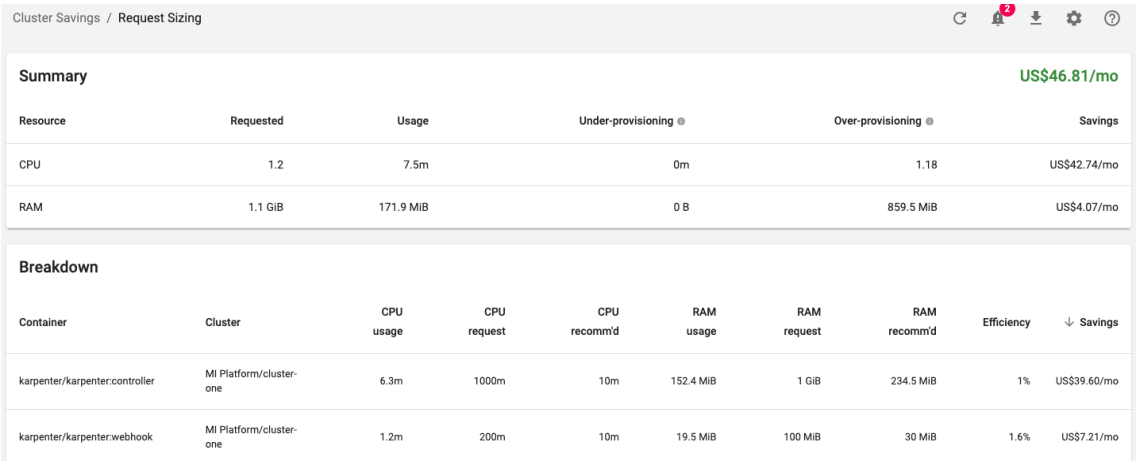
Name	CPU	GPU	RAM	PV	Network	LB	Shared	Efficiency	Total cost
Totals	**US$28.83**	**US$0.00**	**US$11.29**	**US$1.37**	**US$0.83**	**US$0.00**	**US$20.06**	**11.3%**	**US$62.38**
karpenter	US$9.09	US$0.00	US$1.11	US$0.00	US$0.02	US$0.00	US$8.53	1.1%	US$18.75
__idle__	US$11.04	US$0.00	US$7.22	US$0.00	US$0.00	US$0.00	US$0.00	–	US$18.26
kube-system	US$3.38	US$0.00	US$0.53	US$0.00	US$0.79	US$0.00	US$3.92	43.7%	US$8.62
miplatform	US$1.87	US$0.00	US$1.04	US$0.00	US$0.00	US$0.00	US$2.44	1.8%	US$5.35
kube-cost	US$0.80	US$0.00	US$0.27	US$1.28	US$0.02	US$0.00	US$1.98	153.7%	US$4.35
monitoring	US$1.45	US$0.00	US$0.44	US$0.00	US$0.01	US$0.00	US$1.58	101.6%	US$3.48

En el ejemplo anterior podemos ver que Karpenter posee muchos más recursos de los que efectivamente utiliza, y KubeCost posee menos recursos asignados de los necesarios.

Al ir al detalle de Karpenter observamos más información, que muestra claramente dónde se encuentran los problemas y posibles ahorros con optimizaciones.



KubeCost provee varios reportes extra, con información más clara sobre los recursos subutilizados, que permitirán tomar las decisiones adecuadas para la optimización de recursos



7. Conclusiones

Durante el desarrollo de este trabajo, se realizó una investigación exhaustiva sobre Kubernetes y herramientas para su gestión, así como el despliegue de aplicaciones.

Este trabajo llevó más de cuatro meses y su implementación fue gradual, a través de los distintos ambientes, para evitar dejar sin servicio a los usuarios finales.

El proceso indicado en este informe demuestra que es posible realizar una migración de servicios de una plataforma de contenedores, como es el caso de ECS a kubernetes, más específicamente EKS, sin necesidad de modificar las imágenes de los contenedores. Esto maximiza la reutilización del trabajo previamente realizado en tanto a la construcción de imágenes de contenedores. La mayor complejidad estuvo asociada al entendimiento del funcionamiento propio de kubernetes, la elección de las herramientas, junto con sus respectivas pruebas para verificar si eran las más adecuadas para nuestro caso.

El uso de recursos de procesamiento ha sido optimizado significativamente respecto al empleado en la infraestructura anterior. Y la consecuencia fundamental de ello, fue el disponer de métricas que evidencien dónde estaban mal asignados dichos recursos. El escalado frente a la demanda se hace de una forma inmediata y la liberación de recursos es igualmente eficiente. Esto permite una mejor respuesta del sistema en general, sin tener un gran incremento en los costos.

La adopción de Kubernetes como plataforma para correr nuestros contenedores ha sido muy beneficiosa también al momento de realizar cambios, ya que los métodos provistos para la gestión de despliegues ahora son más eficientes y seguros, lo cual contribuye además a la confianza para afrontar el crecimiento de la plataforma con nuevas aplicaciones, dado que con

este esquema, no se requiere ningún cambio a nivel infraestructura, sólo agregar manifiestos que despliegan los recursos necesarios.

7.1. Trabajos futuros

Este informe es solo una tarea dentro del extenso desarrollo de una plataforma, por lo que todo lo mostrado en él seguirá evolucionando según los nuevos requerimientos y productos que aparezcan.

Kubernetes mantiene una gran comunidad que da soporte y constantemente se desarrollan nuevas funcionalidades y servicios, mientras que las funcionalidades actuales evolucionan y maduran, convirtiéndose en candidatas para implementarse en ambientes productivos, por lo que es fundamental mantenerse actualizados.

Un enfoque interesante para un desarrollo posterior, es la incorporación de trazas como parte de la observabilidad, un tema que quedó pendiente porque implica modificar el código de las aplicaciones. Se propone evaluar opentelemetry, debido a su desacoplamiento de una estrategia única de gestión de trazas, independizando el código de esta elección y configuración externa al desarrollo de las aplicaciones.

En otro orden, la seguridad provee múltiples líneas de investigación. Por un lado, podemos mencionar cuestiones de seguridad inherentes a la ejecución de contenedores, donde se propone investigar más sobre imágenes distroless, rootless, siempre en línea con minimizar los privilegios utilizados para correr cada carga de trabajo. Además, el análisis constante de vulnerabilidades dentro del cluster, asociados a manifiestos inseguros, excesivos privilegios, imágenes vulnerables, etc. En otro orden, asegurar las comunicaciones dentro del cluster, usando Network Policies y en el caso de microservicios utilizar Service Mesh para asegurar comunicaciones entre los contenedores utilizando mTLS e incluso obtener como valor agregado métricas RED de cada comunicación en el mesh.

Aún promoviendo minimizar costos, creemos que hay mucho camino por recorrer. Si bien hemos mencionado keda como escalamiento horizontal que admite escalar a cero, existen diferentes alternativas serverless que pueden correrse en kubernetes, como Knative o Kubeless, ofreciendo no solamente costos reducidos, sino simplicidad en el desarrollo de nuevos microservicios independientes entre sí.

Quedan justificadas las ventajas respecto de ofrecer un entorno más versátil, con mayor control por parte del administrador, lo que redunda en una optimización de los recursos en cuanto al control de gastos y búsqueda de soluciones adecuadas.

8. Referencias Bibliográficas

AKS. (2022). *Documentación de Azure Kubernetes Service*. Microsoft Learn. Retrieved February 2, 2023, from https://learn.microsoft.com/es-es/azure/aks/

AKS. (2023, January 29). *Introducción a Azure Kubernetes Service - Azure Kubernetes Service*. Microsoft Learn. Retrieved February 2, 2023, from https://learn.microsoft.com/es-es/azure/aks/intro-kubernetes

AlertManager. (2022). *Alertmanager. Prometheus*. Retrieved February 22, 2023, from https://prometheus.io/docs/alerting/latest/alertmanager/

AllegroGraph. (2022). *AllegroGraph*. Retrieved February 1, 2023, from https://allegrograph.com/

Amazon Backups. (2022). *Copia de seguridad como servicio - Copias de seguridad centralizadas - AWS Backup*. AWS. Retrieved February 2, 2023, from https://aws.amazon.com/es/backup/

Amazon Fargate. (2022). *Motor informático sin servidor – AWS Fargate – Amazon Web Services*. Amazon AWS. Retrieved February 1, 2023, from https://aws.amazon.com/es/fargate/

Ansible. (2022). *Ansible is Simple IT Automation*. Retrieved February 1, 2023, from https://www.ansible.com/

Argo CD. (2022). *Argo CD - Declarative GitOps CD for Kubernetes*. Retrieved February 2, 2023, from https://argo-cd.readthedocs.io/en/stable/

Argo CD CDR. (2022). *ApplicationSet Controller*. Retrieved February 22, 2023, from https://argocd-applicationset.readthedocs.io/en/stable/

Argo CD Webhook. (2022). *Git Webhook Configuration - Argo CD - Declarative GitOps CD for Kubernetes*. Argo CD. Retrieved February 22, 2023, from https://argo-cd.readthedocs.io/en/stable/operator-manual/webhook

AWS. (2022). *What is AWS*. *Cloud computing with AWS*. Retrieved February 1, 2023, from https://aws.amazon.com/what-is-aws/

AWS ALB. (2022). *Application Load Balancer | Elastic Load Balancing | Amazon Web Services*. Amazon AWS. Retrieved February 1, 2023, from https://aws.amazon.com/es/elasticloadbalancing/application-load-balancer/

AWS Cloud Computing. (2022). *What is cloud computing?* AWS. Retrieved February 1, 2023, from https://aws.amazon.com/what-is-cloud-computing/

AWS CloudFormation. (2022). *Aprovisionamiento de infraestructura como código - AWS CloudFormation*. AWS. Retrieved February 1, 2023, from https://aws.amazon.com/es/cloudformation/

AWS CloudWatch. (2022). *Monitoreo de infraestructuras y aplicaciones – Amazon CloudWatch – Amazon Web Services*. AWS. Retrieved February 1, 2023, from https://aws.amazon.com/es/cloudwatch/

AWS Cognito. (2022). *Gestión de identidades y autenticación de usuario en la nube*. AWS. Retrieved February 1, 2023, from https://aws.amazon.com/es/cognito/

AWS EBS. (2022). *Elastic block store (EBS) para almacenamiento persistente*. AWS. Retrieved February 9, 2023, from https://aws.amazon.com/es/ebs/

AWS EC2. (2022). *Elastic compute cloud (EC2) de capacidad modificable en la nube*. Amazon Web Services. Retrieved February 1, 2023, from https://aws.amazon.com/es/ec2/

AWS ECS. (2022). *Gestión de contenedores (ECS) compatible con los de Docker*. AWS. Retrieved February 1, 2023, from https://aws.amazon.com/es/ecs/

AWS EFS. (2022). *Amazon EFS*. AWS. Retrieved February 9, 2023, from https://aws.amazon.com/es/efs/

AWS EKS. (2022). *Servicio de Kubernetes administrado - Amazon EKS - Amazon Web Services*. AWS. Retrieved February 2, 2023, from https://aws.amazon.com/es/eks/

AWS ElastiCache. (2022). *Amazon ElastiCache - Servicio de almacenamiento en caché administrado - Amazon Web Services*. AWS. Retrieved February 1, 2023, from https://aws.amazon.com/es/elasticache/

AWS IAM. (2022). *Administración de identidades | IAM*. AWS. Retrieved February 1, 2023, from https://aws.amazon.com/es/iam/

AWS IAM Prácticas. (2022, Julio 14). Prácticas recomendadas de seguridad en IAM - AWS Identity and Access Management. Retrieved February 1, 2023, from https://docs.aws.amazon.com/es_es/IAM/latest/UserGuide/best-practices.html#lock-away-credentials

AWS NLB. (2022). *Network Load Balancers - Elastic Load Balancing*. Retrieved February 22, 2023, from

https://docs.aws.amazon.com/es_es/elasticloadbalancing/latest/network/network-load-balancers.html

AWS PIT. (2022). *Restauración de una instancia de base de datos a un momento especificado*. Retrieved February 1, 2023, from https://docs.aws.amazon.com/es_es/AmazonRDS/latest/UserGuide/USER_PIT.html

AWS RDS. (2022). *Servicio de bases de datos relacionales (RDS)*. AWS. Retrieved February 1, 2023, from https://aws.amazon.com/es/rds/

AWS SD. (2022). *Service discovery - Implementing Microservices on AWS*. AWS Documentation. Retrieved February 1, 2023, from https://docs.aws.amazon.com/whitepapers/latest/microservices-on-aws/service-discovery.html

AWS Secret Manager. (2022). *Administración de contraseñas de credenciales*. AWS. Retrieved February 1, 2023, from https://aws.amazon.com/es/secrets-manager/

AWS STS. (2022, August 10). *AWS Security Token Service*. Retrieved February 1, 2023, from https://docs.aws.amazon.com/es_es/STS/latest/APIReference/welcome.html

AWS WAF. (2022). *Protección de aplicaciones web*. *AWS*. Retrieved February 1, 2023, from https://aws.amazon.com/es/waf/

Borg. (2015, April 23). *Borg: The Predecessor to Kubernetes*. Kubernetes. Retrieved February 8, 2023, from https://kubernetes.io/blog/2015/04/borg-predecessor-to-kubernetes/

Cluster Autoscaler. (2022). *Kubernetes Cluster Autoscaler. GitHub*. Retrieved February 22, 2023, from https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler

CNCF. (2018, 06 01). *CNCF Cloud Native Definition v1.0*. GitHub. Retrieved February 1, 2023, from https://github.com/cncf/toc/blob/main/DEFINITION.md

CNCF. (2022). *Who We Are*. *Cloud Native Computing Foundation*. Retrieved February 2, 2023, from https://www.cncf.io/about/who-we-are/

containerd. (2022). *containerd – An industry-standard container runtime with an emphasis on simplicity, robustness and portability*. Retrieved February 11, 2023, from https://containerd.io/

CRI-O. (2023). *cri-o/cri-o: Open Container Initiative-based implementation of Kubernetes Container Runtime Interface*. GitHub. Retrieved February 11, 2023, from https://github.com/cri-o/cri-o

Crontab. (2018). *crontab*. *The Open Group Publications Catalog*. Retrieved February 8, 2023, from https://pubs.opengroup.org/onlinepubs/9699919799/utilities/crontab.html

CSI. (2022). *Introduction - Kubernetes CSI Developer Documentation*. Kubernetes-CSI. Retrieved February 9, 2023, from https://kubernetes-csi.github.io/docs/

CSI EBS. (2023). *kubernetes-sigs/aws-ebs-csi-driver: CSI driver for Amazon EBS https://aws.amazon.com/ebs/*. GitHub. Retrieved February 9, 2023, from https://github.com/kubernetes-sigs/aws-ebs-csi-driver

CSI EFS. (2023). *kubernetes-sigs/aws-efs-csi-driver: CSI Driver for Amazon EFS https://aws.amazon.com/efs/*. GitHub. Retrieved February 9, 2023, from https://github.com/kubernetes-sigs/aws-efs-csi-driver

Docker. (2022). *Docker overview*. *Docker Documentation*. Retrieved February 1, 2023, from https://docs.docker.com/get-started/overview/

Dockercon. (2014, June 20). *Dockercon keynote: Eric Brewer (Google)*. YouTube. Retrieved February 8, 2023, from https://www.youtube.com/watch?v=YrxnVKZeqK8

Evans, K. (2018, August 29). *Cloud Native Computing Foundation Receives $9 Million Cloud Credit Grant from Google Cloud to Fund Kubernetes Development, Empower Community*. Linux Foundation. Retrieved February 8, 2023, from https://www.linuxfoundation.org/press/press-release/cncf-receives-9-million-from-google-to-fund-kubernetes

Fluent Bit. (2022). *What is Fluent Bit? Fluent Bit*. Retrieved February 22, 2023, from https://docs.fluentbit.io/manual/about/what-is-fluent-bit

GitLab. (2022). *GitLab CI/CD* | *GitLab*. GitLab Documentation. Retrieved February 2, 2023, from https://docs.gitlab.com/ee/ci/

GitLab Flow. (2022). *Introduction to GitLab Flow* | *GitLab*. GitLab Documentation. Retrieved February 1, 2023, from https://docs.gitlab.com/ee/topics/gitlab_flow.html#environment-branches-with-gitlab-flow

GKE. (2022). *Kubernetes: Google Kubernetes Engine (GKE) | Google Kubernetes Engine (GKE)*. Google Cloud. Retrieved February 2, 2023, from https://cloud.google.com/kubernetes-engine?hl=es-419

Google. (2022). *Google - Site Reliability Engineering*. Retrieved February 1, 2023, from https://sre.google/

Google Cloud. (2022). *¿Qué es cloud computing?* Google Cloud. Retrieved February 1, 2023, from https://cloud.google.com/learn/what-is-cloud-computing?hl=es

Grafana. (2022). *Grafana: The open observability platform | Grafana Labs*. Retrieved February 2, 2023, from https://grafana.com/

Helm. (2022). *Helm*. Helm. Retrieved February 2, 2023, from https://helm.sh/es/

Helmfile. (2022). *Helmfile - Deploy Kubernetes Helm Charts*. Retrieved February 2, 2023, from https://helmfile.readthedocs.io/en/latest/

Helm Dependency. (2022). *Helm Dependency*. Helm. Retrieved February 22, 2023, from https://helm.sh/docs/helm/helm_dependency/

HPA. (2022, November 26). *Kubernetes Horizontal Pod Autoscaling*. Retrieved February 22, 2023, from https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

IBM. (2019). *DevOps*. IBM Cloud Learn Hub. Retrieved January 31, 2023, from https://www.ibm.com/cloud/learn/devops-a-complete-guide

IBM Obs. (2022). *¿Qué es la observabilidad?* IBM. Retrieved February 1, 2023, from https://www.ibm.com/es-es/topics/observability

Jones, C., Murphy, N. R., Petoff, J., & Beyer, B. (Eds.). (2016). *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Incorporated.

Karpenter. (2022). *Karpenter*. Retrieved February 22, 2023, from https://karpenter.sh/

Keda. (2022). *KEDA | Kubernetes Event-driven Autoscaling*. Retrieved February 22, 2023, from https://keda.sh/

Keda Scalers. (2022). *Scalers. KEDA*. Retrieved February 22, 2023, from https://keda.sh/docs/2.9/scalers/

Kerrisk, M. (2022, December 18). *signal(7) - Linux manual page*. man7.org. Retrieved January 31, 2023, from https://man7.org/linux/man-pages/man7/signal.7.html

Kind. (2022, November 7). kind. Retrieved February 8, 2023, from https://kind.sigs.k8s.io/

kube-prometheus-stack. (2023). *kube-prometheus-stack 45.2.0 · prometheus/prometheus-community. Artifact Hub*. Retrieved February 22, 2023, from https://artifacthub.io/packages/helm/prometheus-community/kube-prometheus-stack

Kubecost. (2022). *Kubecost | Kubernetes cost monitoring and management*. Retrieved February 22, 2023, from https://www.kubecost.com/

Kubernetes. (2023, January 3). *Overview*. Kubernetes. Retrieved February 2, 2023, from https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

Litvin, D. (2022, August 10). *Graceful shutdown in a cloud environment (the example of Kubernetes + Spring Boot)*. Maxilect. Retrieved January 31, 2023, from https://maxilect-company.medium.com/graceful-shutdown-in-a-cloud-environment-the-example-of-kubernetes-spring-boot-f922b41adaa0

Minikube. (2022). *minikube start | minikube*. Minikube. Retrieved February 2, 2023, from https://minikube.sigs.k8s.io/docs/start/

Montedoro, L. (2021, April 1). *Deploying Containers in AWS Fargate using Terraform - Part 1*. My Blog. Retrieved February 2, 2023, from https://lmontedoro.github.io/aws/2021/04/01/Deploying-Containers-in-AWS-Fargate-using-Terraform-Pt1.html

Morris, K. (2016). *Infrastructure as Code: Managing Servers in the Cloud*. O'Reilly Media.

Neto, M. D. (2014, March 18). *A brief history of cloud computing - Cloud computing news*. IBM. Retrieved January 31, 2023, from https://www.ibm.com/blogs/cloud-computing/2014/03/18/a-brief-history-of-cloud-computing-3/

NFS. (n.d.). Linux NFS faq. Retrieved February 9, 2023, from https://nfs.sourceforge.net/

NIST. (2011). The NIST Definition of Cloud Computing. Retrieved January 31, 2023, from https://csrc.nist.gov/publications/detail/sp/800-145/final

OpenCost. (2022). *OpenCost — open source cost monitoring for cloud native environments*. Retrieved February 22, 2023, from https://www.opencost.io/

OpenID. (2022). *Welcome to OpenID Connect*. OpenID. Retrieved February 1, 2023, from https://openid.net/connect/

OpenShift. (2022). *Red Hat OpenShift makes container orchestration easier*. Red Hat. Retrieved February 2, 2023, from https://www.redhat.com/en/technologies/cloud-computing/openshift

O'Reilly. (2009, June 25). *10+ Deploys Per Day: Dev and Ops Cooperation at Flickr*. John Allspaw (Flickr/Yahoo!) and Paul Hammond (Flickr). Retrieved February 1, 2023, from https://www.youtube.com/watch?v=LdOe18KhtT4

Ouroboros. (2019). *pyouroboros/ouroboros: Automatically update running docker containers with newest available image*. Ouroboros. Retrieved February 1, 2023, from https://github.com/pyouroboros/ouroboros

OWASP. (2022). *OWASP Top Ten*. OWASP Foundation. Retrieved February 1, 2023, from https://owasp.org/www-project-top-ten/

Parecki, A. (2022). *OAuth 2.0 — OAuth*. OAuth. Retrieved February 1, 2023, from https://oauth.net/2/

PostgreSQL. (2022). *PostgreSQL: The world's most advanced open source database*. Retrieved February 1, 2023, from https://www.postgresql.org/

Prometheus. (2022). Prometheus - Monitoring system & time series database. Retrieved February 2, 2023, from https://prometheus.io/

Pull Strategy. (2023). *FAQ. Prometheus*. Retrieved February 22, 2023, from https://prometheus.io/docs/introduction/faq/#why-do-you-pull-rather-than-push

RBAC. (2022, October 19). *Using RBAC Authorization. Kubernetes*. Retrieved February 22, 2023, from https://kubernetes.io/docs/reference/access-authn-authz/rbac/

RedHat. (2017). *PRINCIPLES OF CONTAINER-BASED APPLICATION DESIGN*. Red Hat. Retrieved January 31, 2023, from https://www.redhat.com/cms/managed-files/cl-cloud-native-container-design-whitepaper-f8808kc-201710-v3-en.pdf

RedHat. (2022, May 11). *What's a Linux container?* Red Hat. Retrieved February 1, 2023, from https://www.redhat.com/en/topics/containers/whats-a-linux-container

RedHat CI/CD. (2022, May 11). *Temas El concepto de DevOps La integración y la distribución continuas (CI/CD)*. Red Hat. Retrieved February 2, 2023, from https://www.redhat.com/es/topics/devops/what-is-ci-cd

SAML. (2007, October 23). *About SAML*. SAML XML.org. Retrieved February 1, 2023, from http://saml.xml.org/about-saml

Schlesinger, K. (2022, June 1). *GitOps to Automate the Setup, Management and Extension a K8s Cluster - Kim Schlesinger, DigitalOcean*. YouTube. Retrieved February 10, 2023, from https://www.youtube.com/watch?v=rra7TkYOnko

Schwaber, K. (2004). *Agile Project Management with Scrum*. Microsoft Press.

Schwaber, K. (2022). *What is Scrum?* Scrum.org. Retrieved February 1, 2023, from https://www.scrum.org/resources/what-is-scrum

SCM. (2022). *Source Code Management*. Atlassian. Retrieved February 1, 2023, from https://www.atlassian.com/git/tutorials/source-code-management

Terraform. (2022). *Terraform by HashiCorp*. Retrieved February 1, 2023, from https://www.terraform.io/

TopQuadrant. (2022). *TopQuadrant | Enterprise Models for Data Governance*. Retrieved February 1, 2023, from https://www.topquadrant.com/

Tracing. (2023). *Tracing (software). Wikipedia*. Retrieved February 22, 2023, from https://en.wikipedia.org/wiki/Tracing_(software)

Udovicic, S. (2022, February 10). *Logstash, Fluentd, Fluent Bit, or Vector? How to choose the right open-source log collector*. Cloud Native Computing Foundation. Retrieved January 31, 2023, from https://www.cncf.io/blog/2022/02/10/logstash-fluentd-fluent-bit-or-vector-how-to-choose-the-right-open-source-log-collector/

Verma, A., Pedrosa, L., Korupolu, M. R., Oppenheimer, D., Tune, E., & Wilkes, J. (2015). *Large-scale cluster management at Google with Borg – Google Research*. Google Research. Retrieved February 8, 2023, from https://research.google/pubs/pub43438/

VPA. (2023). *Vertical Pod Autoscaler. GitHub*. Retrieved February 22, 2023, from https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler

White, S. (2001). *A Brief History of Computing*. Retrieved February 1, 2023, from

https://web.archive.org/web/20010604175515/http://www.ox.compsoc.net/~swhite/timeline.html

Wiggins, A. (2017). The Twelve-Factor App (Traducción de la versión original en Inglés). Retrieved February 1, 2023, from https://12factor.net/es/