UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Programa de Apoyo al Egreso para Alumnos con Práctica Profesional Supervisada

TÍTULO: Análisis de caso de éxito en la implementación de la técnica “Infrastructure as Code” para la creación de ambientes de desarrollo distribuidos

AUTOR: Pedro Martin Bengoa

DIRECTOR/A ACADÉMICO: Matías Urbieta

DIRECTOR/A PROFESIONAL: Mario Papayannis

CARRERA: Licenciatura en Sistemas 2015

RESUMEN

La presente tesis tiene como propósito el análisis del caso de éxito en el desarrollo y posterior adopción de un nuevo paradigma para la creación de entornos distribuidos de desarrollo, basado en los conceptos de Infrastructure as Code (IaC) y contenedorización, destacando cómo este framework permite reducir los riesgos asociados con el desarrollo en entornos centralizados, al tiempo que permitió mejorar la flexibilidad y confiabilidad en todo el ciclo de vida de software. El enfoque elegido para guiar esta investigación se centra en la experiencia personal como líder de dos proyectos pioneros en la implementación de dicho patrón, y los cambios necesarios en el ciclo de vida del desarrollo de software.

Palabras Claves

Infrastructure as Code, docker, contenedorización, OpenShift, Kubernetes, XL Deploy, XL release, etcd, Core and Satellite applications.SDLC,

Conclusiones

La implementación de las herramientas de automatización necesarias para crear infraestructura on-demand ha sido un proceso extenso y demandante en términos de recursos (en el orden de varios millones de dólares). Si bien hasta el momento no se ha alcanzado un retorno de inversión positivo esa tendencia ya muestra signos alentadores, ya que los sistemas están demostrando una estabilidad mucho mayor en comparación con unos meses anteriores.

Trabajos Realizados

Liderar 2 equipos de desarrollo pioneros en la adopción del nuevo paradigma y gestionar con el área de Arquitectura, impulsora de la iniciativa, la resolución de problemas que excedían nuestro alcance.

Trabajos Futuros

Alcanzar el nivel 3 de madurez en la adopción del nuevo paradigma, eliminando la dependencia de los entornos de desarrollo centralizados y completando todo el ciclo de vida de desarrollo de software (SDLC) utilizando únicamente satélites, core y, finalmente, producción.

Fecha de la presentación: SEPT 2024

Índice

Introducción

Objeto de estudio

El objetivo de mi tesis de Práctica Profesional Supervisada (PPS) es presentar un caso de éxito en la implementación de un modelo automatizado para la creación de entornos de desarrollo de software. Este caso de estudio no tiene como fin proponer un nuevo patrón de diseño de aplicaciones, sino demostrar cómo una solución innovadora basada en tecnologías **[Poulton, N (2017) y Gavant & Dumpleton 2018]** y patrones preexistentes [**Morris (2016)**] puede resolver problemas reales de altísima complejidad y aportar valor a una empresa, incluso en empresas muy grandes y con un amplísimo espectro de tecnologías distintas. A través de este caso de éxito personal, se destaca cómo la aplicación del modelo permitió mitigar riesgos asociados con entornos de desarrollo centralizados y redujo el impacto financiero de las interrupciones de los ambientes de desarrollo, proporcionando un entorno más flexible y estable para los equipos de desarrollo.

También se analizan las particularidades de la implementación a gran escala y los desafíos que presenta, ya que la misma debe adaptarse a la variedad de tecnologías y equipos de desarrollo que trabajan en una empresa de IT de gran tamaño donde suelen coexistir un gran número de tecnologías distintas. El estudio analiza cómo se gestionaron la configuración y el versionado en este contexto diverso y complejo siguiendo lineamientos que han sido previamente documentados ampliamente [**Geerling, J. (2015), Petit, P. (2021)**].

Finalmente, el análisis del proceso de implementación y los costos asociados describe la estrategia para desplegar este sistema en una organización tan grande. Esto incluye aspectos como la capacitación de los equipos, la creación de documentación y el análisis de inversiones en tecnología y recursos humanos. El estudio también explora los costos recurrentes para el mantenimiento del sistema automatizado y proporciona información sobre cómo se abordaron estos costos para garantizar un despliegue exitoso y sostenible.

Contexto

Durante más de cinco años, he estado prestando servicios para una empresa estadounidense líder en el sector de la gestión de recursos humanos y nóminas de pago. Esta empresa, con aproximadamente 16,000 empleados y más de 100 equipos de desarrollo de software, gestiona cientos de sistemas informáticos que sustentan sus operaciones a gran escala. Estos equipos, generalmente organizados bajo metodologías ágiles, trabajan en sprints sincronizados y, a menudo, dependen de los

mismos servicios e infraestructuras clave, como bases de datos relacionales y no relacionales, colas de mensajes, Active Directory, web services y aplicaciones específicas a distintas líneas de negocio. Estas aplicaciones proporcionan servicios transversales que son fundamentales para el funcionamiento eficiente de toda la empresa.

Para asegurar la calidad y la confiabilidad del software desarrollado, la empresa ha establecido una serie de entornos de testing bien definidos:

- **n2a (y dev):** Es el entorno de testing más básico y, por tanto, el más inestable. Aquí es donde se realizan las pruebas iniciales de los desarrollos. Dado su propósito, es un entorno donde se espera encontrar errores y fallos, ya que es el primer punto de contacto entre el código nuevo y un entorno controlado.
- **n0**: Este entorno es el primer paso hacia un entorno más estable. Aquí se prueba el código que ya ha superado las pruebas en Tier 1 y que se considera candidato para ser lanzado. Aunque es más estable que Tier 1, todavía es un entorno donde los errores pueden ser identificados y corregidos antes de avanzar a entornos más críticos.
- **n1:** Este entorno está diseñado para ser una réplica exacta del entorno de producción, por lo que es comúnmente conocido como preproducción. Tier 3 tiene como objetivo ofrecer una experiencia lo más cercana posible al entorno de producción, permitiendo detectar y corregir cualquier problema que pudiera pasar desapercibido en los niveles anteriores.
- **Perf:** Un entorno especialmente diseñado para realizar pruebas de rendimiento y estrés. Aquí se evalúa cómo el software se comporta bajo condiciones extremas de uso, asegurando que pueda manejar la carga esperada y más allá, sin comprometer su estabilidad o rendimiento.
- **Producción:** El entorno de producción es el ambiente real y productivo al que acceden los clientes. Aquí es donde el software finalmente se despliega para su uso en vivo, y donde cualquier problema podría tener un impacto directo en la experiencia del cliente y en la operación del negocio.

Estos entornos de testing centralizados son únicos y compartidos por todos los equipos de desarrollo, lo que incluye más de 100 equipos, cada uno con un promedio de 10 desarrolladores. Este enfoque centralizado, aunque eficiente en términos de recursos, trae consigo ciertos desafíos, especialmente en los niveles inferiores de testing, como en **N2a**.

Diagrama de los ambientes en cascada utilizados en el SDLC

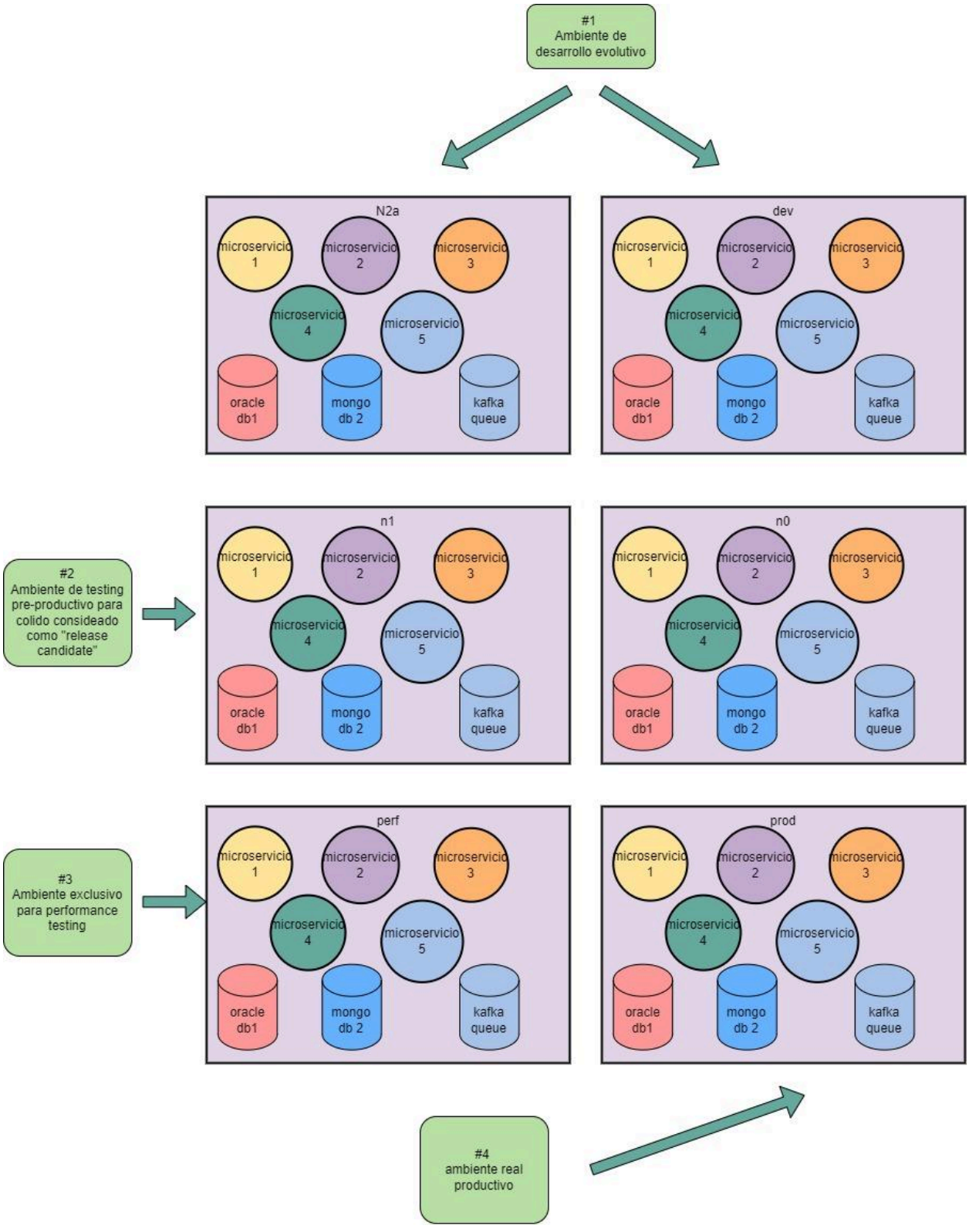

Figura 1: muestra los distintos ambientes de testing, en cascada, centralizados e iterativos

N2a es un entorno de testing particularmente sensible y crítico debido a su naturaleza evolutiva. Este entorno está diseñado para estar en constante cambio, permitiendo a los equipos de desarrollo desplegar, probar y modificar su código de manera continua. Sin embargo, debido a que es un entorno compartido por tantos equipos, con desarrolladores que a menudo trabajan en paralelo y realizan múltiples cambios simultáneos, la inestabilidad es un problema recurrente. Esta inestabilidad se manifiesta de diversas maneras, desde pequeñas interrupciones en partes del entorno hasta caídas completas del sistema.

Cada vez que uno de estos cambios provocaba la caída de todo el entorno o incluso de una pequeña parte, los efectos se sentían en toda la organización. La naturaleza interdependiente de los sistemas significaba que una pequeña falla en un componente podía tener un efecto dominó, afectando a otros equipos y a sus capacidades para avanzar en sus sprints. Esto no solo causaba retrasos en la entrega de productos, sino que también generaba pérdidas económicas significativas para la empresa.

Para comprender el impacto financiero de estas caídas, consideremos lo siguiente: con un promedio de 40 dólares por hora para cada desarrollador, y dado que más de 1,000 desarrolladores podrían estar trabajando en este entorno compartido, un sólo día de inactividad podía resultar en pérdidas que superan los 320,000 dólares. Estas pérdidas se acumulaban rápidamente, afectando no solo la rentabilidad de los proyectos, sino también la moral de los equipos y la eficiencia general del proceso de desarrollo.

La magnitud de estas pérdidas llevó a la empresa a enfrentar una difícil disyuntiva: cómo mantener entornos de desarrollo completos y estables, que permitan a los equipos avanzar de manera rápida y eficiente, al tiempo que se minimiza el riesgo de inestabilidad y se controlan los costos asociados a los tiempos de inactividad. La solución que se propuso fue la creación de entornos de desarrollo personales, donde cada desarrollador tendrá la posibilidad de desplegar y probar sus desarrollos de manera aislada. Estos entornos personales incluyen tanto los servicios que el desarrollador está modificando (satélites) como todos los componentes comunes necesarios para el correcto funcionamiento de estos servicios (aplicaciones centrales).

Este enfoque no solo prometía mejorar la estabilidad general de los entornos de testing, sino que también permitiría a los desarrolladores experimentar y probar nuevas ideas sin el riesgo de afectar a otros equipos o al progreso general de la empresa. Como resultado, la empresa podría reducir significativamente las pérdidas económicas derivadas de la inestabilidad de los entornos compartidos, mientras que los equipos de desarrollo pudieron mantener su ritmo de trabajo y cumplir con los objetivos de sprint de manera más eficiente.

Marco teórico

Siguiendo el objetivo mencionado con anterioridad, el equipo de Arquitectura desarrolló un nuevo entorno de desarrollo (al que llamaremos por el nombre ficticio "Tier-Core" para proteger cualquier información potencialmente confidencial) que, utilizando los principios delineados por la práctica conocida como "**Infrastructure as Code**" (IaC) [**Morris, K. (2016) y Jourdan, S., & Pomes, P. (2020)**], permitió generar de manera totalmente automatizada nuevos entornos de desarrollo. Estos entornos fueron divididos en dos categorías: ambientes "**Core**", que son compartidos por todos los equipos de desarrollo, y ambientes "**Satellite**", cuyo uso es individual para cada equipo de desarrollo, o incluso para cada desarrollador dentro de un equipo.

Esta nueva arquitectura automatizada ofrece varios beneficios clave. Los ambientes "**Core**" proporcionan servicios y componentes comunes, permitiendo la integración de diferentes equipos. Por su parte, los ambientes "**Satellite**" otorgan flexibilidad y aislamiento, permitiendo a los equipos o desarrolladores individuales probar sus innovaciones sin interferir con otros equipos. Gracias a este enfoque automatizado basado en **IaC**, el tiempo para crear nuevos entornos de testing se reduce significativamente, permitiendo a los equipos moverse con mayor agilidad y responder más rápido a las necesidades cambiantes del proyecto o del negocio.

Ambos tipos de aplicaciones, tanto las que están en el núcleo ("**core**") como las que están en los satélites ("**satellite**"), tienen ciertas características clave:

- Creación completamente automatizada: Los procesos para crear y configurar las aplicaciones deben ser totalmente automatizados, garantizando consistencia y velocidad.
- Externalización de credenciales y secretos: Para mayor seguridad, las credenciales y secretos (como contraseñas y claves API) se almacenan externamente y se accede a ellos de manera segura.
- Tiempo de vida definido: Las aplicaciones tienen un tiempo de vida limitado, lo que ayuda a mantener el entorno fresco y libre de residuos. En el caso de las aplicaciones core, hay dos entornos que se alternan cada dos meses, lo que implica que uno se destruye y se recrea por completo (conocido como "core swap"). Para las aplicaciones satélite, el tiempo de vida es de dos semanas.
- Aislamiento de red: El entorno Tier-Core tiene su propio subdominio de red, lo que significa que no tiene acceso al dominio de otras aplicaciones, garantizando un mayor nivel de seguridad y evitando interferencias.

El nuevo entorno "**Tier-Core**" proporciona un entorno flexible y seguro para el desarrollo y testing, permitiendo a los equipos y desarrolladores trabajar de manera más ágil, mientras se mantiene un alto nivel de estabilidad y seguridad. Al dividir las

aplicaciones entre core y satellite, y con procesos de automatización y gestión de credenciales seguros, este enfoque puede ser una solución eficaz para abordar los problemas de entornos inestables y el alto acoplamiento entre sistemas.

La solución llamada **Tier-Core** se basa principalmente en dos conceptos ampliamente estudiados: la **contenedorización** [Turnbull, J. (2014)] y la **infraestructura como código** (Morris, K. (2016)). Estos enfoques han sido adoptados por muchas empresas debido a sus ventajas para gestionar la infraestructura de manera más eficiente y flexible.

La contenedorización permite empaquetar aplicaciones y sus dependencias en unidades autocontenidas llamadas contenedores. Esto facilita la portabilidad entre entornos y reduce los problemas de compatibilidad, ya que cada contenedor contiene todo lo necesario para ejecutar la aplicación. Con contenedores, es posible crear entornos consistentes de desarrollo, testing y producción, reduciendo el riesgo de conflictos entre diferentes componentes del sistema.

Por otro lado, la infraestructura como código (IaC) permite definir y gestionar la infraestructura mediante archivos de configuración. Con IaC, se puede automatizar la creación, modificación y eliminación de recursos de infraestructura, como servidores, bases de datos, redes y otros componentes. Al tratar la infraestructura como código, es posible aplicar prácticas de desarrollo de software, como control de versiones, pruebas automatizadas y despliegues consistentes.

Tier-Core combina estos conceptos para proporcionar un sistema automatizado de creación de entornos de desarrollo. La contenedorización garantiza que cada entorno sea independiente y aislado, permitiendo a los desarrolladores trabajar sin interferir con otros equipos. IaC facilita la automatización de la creación y configuración de estos entornos, permitiendo reproducirlos y escalarlos según sea necesario.

Esta combinación de contenedorización e infraestructura como código ha demostrado ser efectiva para resolver desafíos de estabilidad y escalabilidad en entornos de desarrollo complejos. La solución **Tier-Core** proporciona flexibilidad para los desarrolladores y, al mismo tiempo, garantiza la consistencia y seguridad necesarias para mantener la calidad del software en una empresa grande y diversa.

Infrastructure as Code

Los principios de Infrastructure as Code (IaC) establecidos por Kief Morris en su libro *"Infrastructure As Code: Managing Servers in the Cloud"* proporcionan un marco claro para la gestión de infraestructura a través del código. Estos principios ayudan a las organizaciones a crear y mantener infraestructuras consistentes, fiables y adaptables. A continuación, se describen cada uno de estos principios en detalle:

- **Reproducibilidad**:
  Este principio se centra en la capacidad de crear entornos de infraestructura de manera consistente, asegurando que se obtenga el mismo resultado cada vez que se ejecuten las configuraciones. Con IaC, puedes capturar la definición de la infraestructura en archivos de configuración y usar esos archivos para reproducir entornos en diferentes momentos y ubicaciones. La reproducibilidad es clave para garantizar consistencia, especialmente cuando se trabaja con infraestructuras complejas o en entornos distribuidos.
- **Idempotencia**:
  Impotencia significa que una operación aplicada repetidamente debe producir el mismo resultado que aplicada una sola vez. En el contexto de IaC, la idempotencia garantiza que los cambios en la infraestructura no causen efectos secundarios inesperados o inconsistencias. Por ejemplo, si un script IaC se ejecuta varias veces, el resultado debe ser siempre el mismo. Esto contribuye a la estabilidad y confiabilidad de la infraestructura.

- **Componibilidad**:
  La componibilidad implica la capacidad de ensamblar componentes de infraestructura de forma modular y eficiente para construir sistemas más grandes y complejos. En IaC, este principio permite crear componentes reutilizables que pueden combinarse de diferentes maneras para formar infraestructuras más grandes. La composición facilita la adaptabilidad y la extensión de la infraestructura a medida que cambian las necesidades del negocio.

- **Evolucionabilidad**:
  Este principio se refiere a la capacidad de la infraestructura para adaptarse y evolucionar con el tiempo para satisfacer las demandas cambiantes del negocio. Con IaC, la infraestructura puede ser modificada, actualizada y ampliada sin causar interrupciones significativas en los servicios existentes. Esto es posible

porque el código de infraestructura puede ser fácilmente modificado y aplicado de manera controlada.

Contenedorización

La contenedorización es una tecnología que permite empaquetar aplicaciones y sus dependencias en unidades autocontenidas conocidas como contenedores. Estos contenedores pueden ejecutarse de manera consistente en diferentes entornos, lo que facilita la portabilidad y reduce problemas de compatibilidad. A continuación, se presentan los conceptos clave de la contenedorización:

- **Contenedores**:
  Un contenedor es una unidad autocontenida que incluye una aplicación y todas sus dependencias, como bibliotecas, archivos de configuración y otros recursos necesarios para ejecutarla. Los contenedores comparten el mismo núcleo del sistema operativo, pero están aislados unos de otros, lo que proporciona una mayor eficiencia y escalabilidad en comparación con las máquinas virtuales.
- **Imágenes de Contenedores**:
  Una imagen de contenedores es un archivo estático que contiene todos los componentes necesarios para crear y ejecutar un contenedor. Las imágenes son inmutables, lo que significa que no se pueden modificar una vez creadas. Para cambiar una imagen, se crea una nueva versión a partir de la imagen base y se redistribuye.
- **Aislamiento**:
  La contenedorización proporciona aislamiento entre aplicaciones y procesos, lo que significa que un contenedor no puede afectar a otros contenedores en el mismo sistema. Este aislamiento ayuda a mejorar la seguridad y la estabilidad, permitiendo ejecutar múltiples contenedores en un solo host sin interferencias entre ellos.
- **Portabilidad**:
  Los contenedores son portátiles, lo que significa que pueden ejecutarse en diferentes entornos sin cambios en el código. Esto facilita el despliegue de aplicaciones en distintas plataformas, como servidores locales, nubes públicas o privadas, y entornos híbridos.
- **Orquestación**:
  La orquestación es el proceso de gestionar y coordinar múltiples contenedores en un entorno. Herramientas como Kubernetes y Docker Swarm permiten automatizar tareas como el escalado, el equilibrio de carga, la gestión de

recursos y la recuperación automática de fallos. La orquestación es esencial para administrar aplicaciones de contenedorización a gran escala.

- **Escalabilidad**:
  Los contenedores son altamente escalables. Debido a su naturaleza liviana, es posible ejecutar múltiples instancias de un contenedor para satisfacer las demandas crecientes de tráfico o procesamiento. La orquestación permite escalar automáticamente los contenedores según las necesidades del sistema.
- **DevOps y CI/CD**:
  La contenedorización es una parte fundamental del enfoque DevOps **[Picozzi, S., & Hepburn, M. (2018)]**, permitiendo una integración y entrega continuas (CI/CD). Los contenedores se pueden usar para automatizar pruebas, despliegues y otras tareas del ciclo de vida del desarrollo de software, mejorando la eficiencia y reduciendo el tiempo para llevar nuevas funciones al mercado.

Estos principios son fundamentales para la práctica efectiva de Infrastructure as Code. Cuando se aplican correctamente, permiten a las organizaciones gestionar la infraestructura de manera eficiente y segura, reduciendo el riesgo de errores humanos y facilitando el mantenimiento y la evolución de los sistemas.

Trabajo realizado

El trabajo elaborado como parte de esta tesis se divide en dos áreas principales. En primer lugar, se ofrece una descripción exhaustiva de mi experiencia y participación personal en la iniciativa, la cual he estado desarrollando de manera intermitente durante más de dos años. Esta sección detalla mi rol y las responsabilidades asumidas, así como el proceso de aprendizaje, capacitación y liderazgo en la adopción de una nueva metodología de desarrollo. Se examina la formación adquirida, la transferencia de conocimiento al equipo, y la implementación práctica de la metodología, incluyendo la configuración de aplicaciones, gestión de secretos y ajuste de pruebas de automatización.

En segundo lugar, se realiza un metaanálisis del proceso seguido por la empresa que condujo a la implementación del sistema Tier-core. Este análisis abarca la investigación y evaluación de tecnologías disponibles en el mercado que podrían resolver los problemas identificados, la propuesta de una implementación concreta para el sistema, y la creación de un modelo de madurez para la adopción de la nueva metodología. Se examina detalladamente la búsqueda de soluciones tecnológicas adecuadas, la

formulación y justificación de la propuesta de implementación, y la propuesta de un modelo de madurez para asegurar una adopción efectiva y escalable de la nueva metodología. Este metaanálisis proporciona una visión integral del proceso de selección, implementación y optimización de la metodología, con el objetivo de ofrecer un marco robusto para futuras iniciativas similares.

Mi participación en esta iniciativa

Mi participación en esta iniciativa comenzó hace aproximadamente tres años, cuando asumí el rol de Líder Técnico en uno de los primeros equipos encargados de la incorporación y adopción de una nueva metodología de desarrollo. Como parte de mis responsabilidades iniciales, mi primer paso fue capacitarme a través de cursos y presentaciones impartidas por expertos en arquitectura, complementados con una amplia gama de recursos proporcionados por la empresa. Durante este proceso, también recibí soporte continuo, lo que fue esencial mientras lideraba uno de los equipos pioneros en la adopción de esta metodología.

Una vez que completé mi capacitación, mi siguiente objetivo fue transferir el conocimiento adquirido al resto del equipo, abarcando tanto a los desarrolladores como a los miembros no técnicos, incluyendo analistas funcionales y el equipo de aseguramiento de calidad (QA). Esta capacitación no solo cubrió los fundamentos del nuevo paradigma, sino también aspectos específicos relacionados con su implementación práctica.

Con el equipo ya capacitado en los principios básicos de la nueva arquitectura, asumí la responsabilidad de liderar la integración de todas las aplicaciones dentro de la línea de trabajo de nuestro equipo, enfocado en un cliente específico. Personalmente configuré las primeras seis aplicaciones, lo que implicó la creación de archivos que definían la infraestructura necesaria para cada una, incluyendo las conexiones a componentes downstream como bases de datos, colas de mensajes y otros microservicios esenciales para el correcto funcionamiento de las aplicaciones. También configuré los elementos clave de cada aplicación, como los proyectos, aplicaciones y archivos de enrutamiento utilizados para la creación de los Application Load Balancers (ALB), comúnmente conocidos como balanceadores de carga.

Además, me encargué de gestionar adecuadamente los OpenShift Secrets, los cuales se utilizan para externalizar las contraseñas de las aplicaciones del cliente. Dado que en este entorno específico se adoptó un nuevo sistema de encriptación diferente al que se utilizaba anteriormente, fue necesario migrar estos secretos a la nueva plataforma. Este proceso implicó no solo la reconfiguración de los OpenShift Secrets, sino también

asegurar que la migración al nuevo sistema de encriptación se realizara de manera segura y sin interrupciones en los servicios.. Finalmente, ajusté las pruebas de automatización para que se ejecutarán durante el despliegue en el entorno "**core**", adaptando la lógica de formación de URLs al formato requerido tanto por las aplicaciones "**core**" como por las "**satellites**". Esto requirió modificaciones en los scripts de prueba existentes para alinearlos con los nuevos estándares.

Después de adquirir la experiencia necesaria, me dediqué a compartir este conocimiento no solo con mi equipo, sino también con otros equipos dentro de la empresa y con equipos de clientes que estaban adoptando esta tecnología. A través de esta transferencia de conocimiento, he contribuido significativamente a la expansión y adopción de la nueva metodología, ayudando a otros a comprender y aplicar los principios y prácticas necesarios para su implementación exitosa.

En resumen, mi papel en esta iniciativa ha sido multifacético: desde aprender y dominar la nueva metodología, hasta capacitar al equipo y liderar la incorporación de aplicaciones, garantizando el cumplimiento de todos los requisitos de configuración y seguridad, y adaptando las pruebas de automatización para asegurar un proceso de despliegue fluido y confiable.

El sistema propuesto

Objetivos

El proyecto **Tier-Core** fue desarrollado por el equipo de arquitectura para mejorar la disponibilidad y flexibilidad de los entornos del ciclo de vida del desarrollo de software (**SDLC**).
Los objetivos fundamentales del proyecto son los siguientes:

- **Mayor disponibilidad de los entornos de desarrollo**: Se busca reducir el impacto de incidentes, fallos de despliegue y problemas de configuración mediante un mayor aislamiento de entornos y flexibilidad en su uso. Esto debería resultar en un SDLC más confiable y estable.
- **Mejor calidad en el desarrollo de software y cambios en la infraestructura**: Proporcionando entornos bajo demanda, automatizados y controlados por versiones, se pretende reducir el tiempo necesario para corregir defectos de software y problemas de configuración.
- **Mayor capacidad para identificar, desarrollar, probar y desplegar funciones y correcciones de errores**: La meta es permitir a los equipos responder más

rápidamente a las necesidades de los clientes, introduciendo características y mejoras en menos tiempo.

- **Reducción en el tiempo de espera para la construcción de infraestructura**: Con una mayor automatización, se espera que el tiempo de configuración y despliegue se reduzca, acelerando el SDLC.
- **Mayor seguridad**: Con la introducción de técnicas como Data Masking (enmascaramiento de datos), automatización de firewalls, y segregación de identidad y recursos entre entornos de producción y no producción, el objetivo es incrementar el nivel de seguridad de la infraestructura.
- **Mayor capacidad de servicio y soporte**: Ofreciendo soporte y mantenimiento a nivel de producción para entornos no productivos, el proyecto busca mejorar la estabilidad y la confiabilidad para las necesidades del SDLC de la empresa.
- **Refrescos periódicos de datos e infraestructura**: Con un proceso de actualización cada 30 días, se garantiza que los entornos de desarrollo y testing estén actualizados y reflejan mejor la realidad del entorno de producción.

Arquitectura del sistema

El nuevo entorno (al que llamaremos por el nombre ficticio "**Tier-Core**" para proteger cualquier información potencialmente confidencial) dividió a todas los componentes de software en dos categorías:

- Aplicaciones "**Core**", que son compartidas por todos los equipos de desarrollo.
- Aplicaciones "**Satellite**", cuyo uso es individual para cada equipo de desarrollo, o incluso para cada desarrollador dentro de un equipo.

Esta nueva arquitectura automatizada ofrece varios beneficios clave. Los ambientes "**Core**" proporcionan servicios y componentes comunes, permitiendo la integración de diferentes equipos. Por su parte, los ambientes "**Satellite**" otorgan flexibilidad y aislamiento, permitiendo a los equipos o desarrolladores individuales probar sus innovaciones sin interferir con otros equipos. Gracias a este enfoque automatizado basado en IaC, el tiempo para crear nuevos entornos de testing se reduce significativamente, permitiendo a los equipos moverse con mayor agilidad y responder más rápido a las necesidades cambiantes del proyecto o del negocio.

Hay dos entornos principales para el desarrollo, con al menos uno siempre activo. Se realiza un refresco aproximadamente cada 30 días, donde se reinicia toda la infraestructura y se replica una nueva copia de los datos de producción, que se enmascaran para mantener la privacidad. Los procesos están completamente automatizados y orquestados. Para saber qué entorno de desarrollo está activo, se puede consultar ADL-UI, donde el menú desplegable muestra los cores disponibles.

Ambos entornos, TierCoreD y TierCoreE, tienen datos replicados de producción para garantizar que reflejen la configuración y el contenido más recientes.

Diagrama de componentes

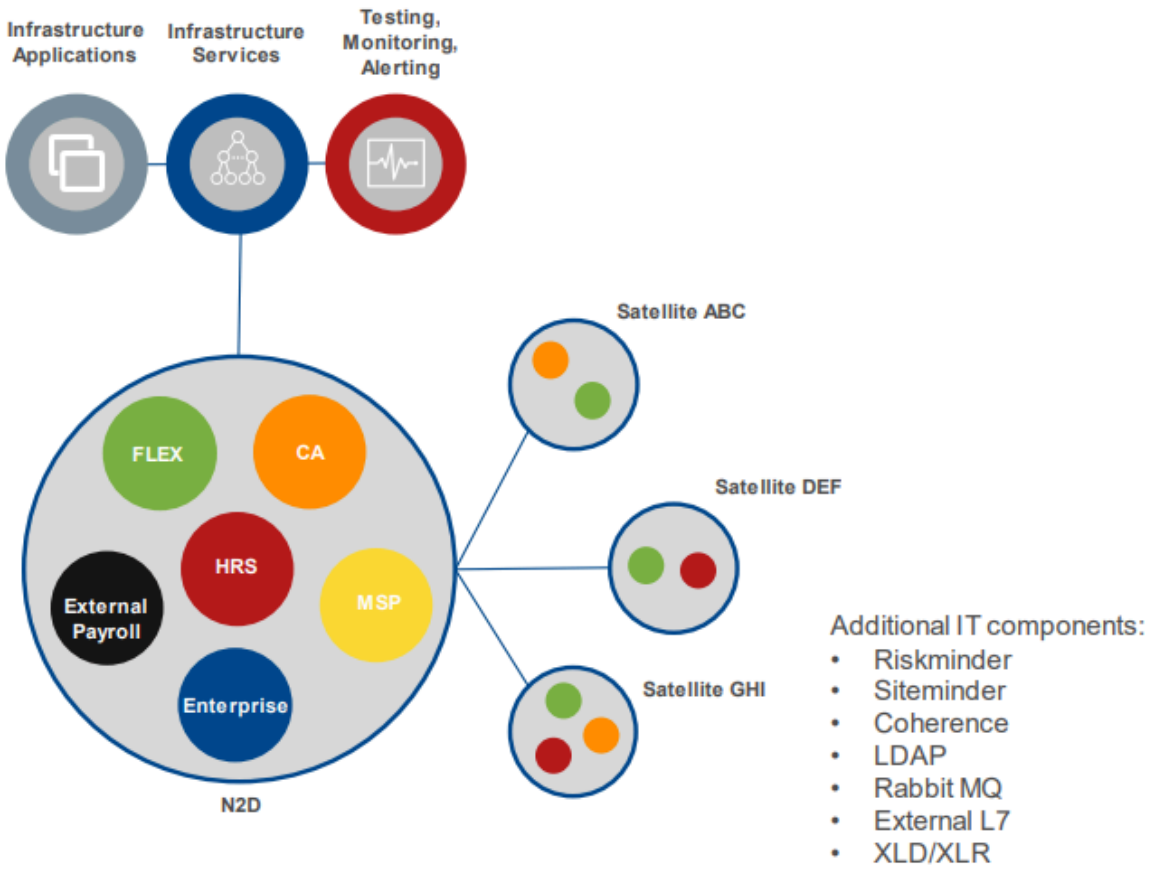

Figura 2: Este diagrama muestra por un lado, el core (N2D) con sus distintas verticales de negocio(flex, CA, HRS,etc) donde cada vertical suele contenedor cientos de servicios y componentes de software, y por otro lado los satélites que pueden contener subsets más pequeños de cada vertical de negocio (Satellite ABC tiene sólo Flex y CA, Satellite DEF tiene Flex y HRS).

Ambos tipos de aplicaciones, tanto las que están en el núcleo ("**core**") como las que están en los satélites ("**satellite**"), tienen ciertas características clave:

- Creación completamente automatizada: Los procesos para crear y configurar las aplicaciones deben ser totalmente automatizados, garantizando consistencia y velocidad.

- Externalización de credenciales y secretos: Para mayor seguridad, las credenciales y secretos (como contraseñas y claves API) se almacenan externamente y se accede a ellos de manera segura.
- Tiempo de vida definido: Las aplicaciones tienen un tiempo de vida limitado, lo que ayuda a mantener el entorno fresco y libre de residuos. En el caso de las aplicaciones core, hay dos entornos que se alternan cada dos meses, lo que implica que uno se destruye y se recrea por completo (conocido como "core swap"). Para las aplicaciones satélite, el tiempo de vida es de dos semanas.
- Aislamiento de red: El entorno Tier-Core tiene su propio subdominio de red, lo que significa que no tiene acceso al dominio de otras aplicaciones, garantizando un mayor nivel de seguridad y evitando interferencias.

Componentes principales

Los principales componentes de **Tier-Core** están diseñados para proporcionar una infraestructura robusta y escalable para entornos de desarrollo y testing. A continuación, se describen los elementos clave que forman la base de esta arquitectura:

Shared Services

- **Infraestructura Compartida**: Esta sección incluye recursos que son comunes para todos los entornos Core. Aquí se encuentran servicios como monitoreo y herramientas de automatización (como Foreman). Esta infraestructura permanece activa de manera constante.
- **Proceso de Refrescos**: El sistema tiene un mecanismo para desmontar y reconstruir la infraestructura periódicamente, permitiendo actualizar datos y configuraciones para mantener la coherencia con la producción.

Cores

- **Entornos Core**: Estos son los entornos de mayor escala, que incluyen todos los sistemas y funcionalidades del entorno de producción. Cada Core tiene un subdominio único (por ejemplo, example-app.n2d-lb.payxdev.com).
- **Funciones Dedicadas**: Cada entorno Core puede tener un propósito específico. Por ejemplo, n2d/e está orientado a Flex, mientras que n2t se centra en herramientas. Esto permite una especialización por entorno.
- **Actualización Programada**: Los Cores se actualizan a partir de un manifiesto controlado por versiones, garantizando que todos los componentes estén actualizados. Tienen una vida útil de alrededor de 60 días o más.

- **Soporte para Múltiples Entornos**: La arquitectura permite que varios entornos Core estén activos al mismo tiempo, brindando soporte para múltiples necesidades.

Satellites

- **Entornos Satélite**: Estos son entornos más pequeños, que contienen sólo un subconjunto de sistemas y funcionalidades de un Core. Los satélites tienen nombres de tres letras y un subdominio propio (por ejemplo, example-app.xyz.n2d-lb.payxdev.com).
- **Vinculación con un Core**: Los Satélites dependen de un Core para ciertos componentes que no contienen, manteniendo una relación 1-a-muchos.
- **Creación Bajo Demanda**: Los Satélites se pueden crear según se necesiten mediante ADL-UI, una aplicación web que facilita el despliegue.
- **Vida Útil Breve**: Los satélites tienen un ciclo de vida más corto, de aproximadamente tres semanas, lo que favorece la limpieza periódica para evitar acumulaciones no deseadas.

Componentes de automatización

- **PADL** (Process Application Definition Language): Un lenguaje basado en YAML utilizado para definir infraestructura y aplicaciones en Tier-Core. Los archivos ADL están bajo control de versiones, proporcionando trazabilidad y consistencia.
- **HodgePodge**: Una aplicación backend que valida y gestiona todas las definiciones de ADL, asegurando la consistencia antes del despliegue.
- **ADL-UI**: Es una interfaz gráfica web para solicitar entornos Satélite y obtener información sobre entornos Core y Satélite existentes.
- **etcd**: Una base de datos para representar el estado de un entorno, proporcionando un sistema confiable para gestionar la infraestructura.
- **AWX**: Un motor de flujos de trabajo basado en **Ansible** que permite la automatización, impulsando la creación y configuración de entornos.

Lógica de Enrutamiento y Fallback

En la arquitectura distribuida de la empresa, se implementó un sistema de enrutamiento que optimiza la comunicación entre microservicios en distintos entornos, conocidos como "**satellites**", y un "core" central. Este enfoque garantiza eficiencia y alta disponibilidad.

Dentro de cada "satellite", los microservicios se comunican utilizando URLs que apuntan a otros servicios en el mismo entorno, priorizando la resolución local para reducir la latencia. Si un servicio no se encuentra en el "**satellite**", una lógica de fallback redirige la solicitud al "core",

donde se mantiene una copia central de todos los servicios, asegurando así la continuidad del servicio.

Este mecanismo se asemeja a la gestión de dependencias en un Entorno de Desarrollo Integrado (**IDE**). Al igual que el sistema de enrutamiento, un IDE busca primero las dependencias en el repositorio local del proyecto. Si no se encuentran, se recurre a un repositorio central en línea, garantizando que las dependencias necesarias siempre estén disponibles.Este enfoque híbrido ofrece resiliencia al asegurar que las solicitudes siempre encuentren una respuesta, optimiza los recursos al priorizar el enrutamiento local, y asegura alta disponibilidad al permitir que el sistema recupere servicios desde el "**core**" cuando no están presentes en el "**satellite**".

En resumen, la lógica de enrutamiento y fallback no solo mejora la eficiencia y resiliencia de la infraestructura, sino que también refleja patrones de diseño ampliamente utilizados en la gestión de dependencias en el desarrollo de software, asegurando un servicio continuo y confiable en entornos distribuidos.

Diagrama de flujo de trabajo de la automatización

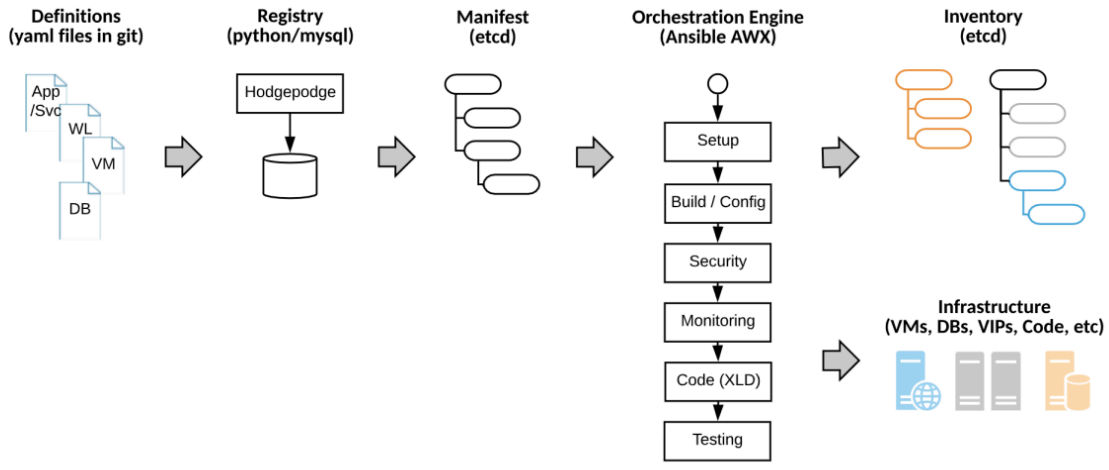

Figura 3: Flujo de tareas automatizadas para coordinar el despliegue de nuevas versiones.

En el diagrama anterior se puede apreciar el flujo de tareas automáticas para el despliegue de nuevas versiones de software:

1. **Publicación en Bitbucket**: Se publica un archivo yml en un repositorio de Bitbucket.
2. **Registro y Actualización en etcd**: La publicación del archivo yml dispara un proceso que registra el nuevo componente en una base de datos etcd, actualizando su definición en la infraestructura.
3. **Ejecución de Scripts en AWX**: Se disparan scripts automatizados en el motor AWX. AWX orquesta la creación del nuevo componente basándose en la definición actualizada en etcd.
4. **Actualización en etcd**: Una vez que el componente ha sido creado, se actualiza la base de datos etcd para reflejar la incorporación del nuevo componente.
5. **Testing**: Finalmente, se realizan tareas de testing, ya sean manuales o automatizadas, para verificar que el nuevo componente funciona como se espera.

Este flujo asegura una integración continua y automatizada, desde la definición y creación de componentes hasta la validación final.

Tal vez el componente más importante sobre el que trabaja regularmente un desarrollador al momento de “onboardear” una nueva aplicación al core es el archivo PADL, a continuación se muestra un ejemplo del mismo

Ejemplo de archivo ADL

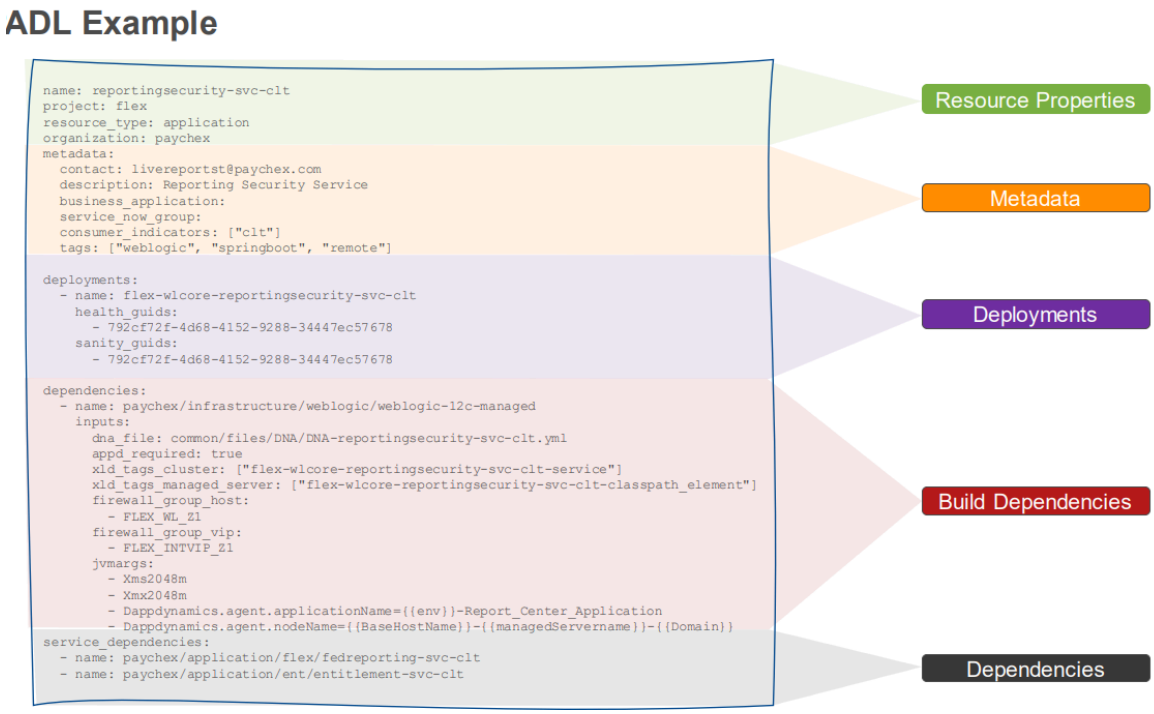

Figura 4: ejemplo de un PADL file describiendo un componente de infraestructura, en este caso un microservicio Springboot que corre sobre Openshift

En el contexto del cliente, muchas de las herramientas de automatización utilizan los conceptos de "Convention over Configuration" (CoC) y "Don't Repeat Yourself" (DRY) para inferir, por ejemplo, las rutas completas de los componentes de software, las URLs de servicios, o la configuración de los Application Load Balancers (ALBs).
En este ejemplo, para construir la URL completa de un servicio, se utilizan diferentes elementos definidos en un archivo de configuración, como la plataforma (por ejemplo, "ose" para OpenShift), el tipo de aplicación (como "svc" para servicios), y la naturaleza de los consumidores (internos o externos). Esto permite generar URLs consistentes y evitar la repetición manual, aprovechando las convenciones preestablecidas.
Con estos enfoques, se logra un proceso de automatización más sencillo y confiable, minimizando la necesidad de configuración manual y permitiendo a los desarrolladores centrarse en aspectos más importantes del desarrollo de software. Además, el uso de CoC y DRY ayuda a mantener la uniformidad en la infraestructura, contribuyendo a la escalabilidad y estabilidad de las aplicaciones y servicios.

Revisión de tecnologías IaC

En el contexto actual de la gestión y creación automatizada de infraestructura, diversas herramientas han emergido para facilitar estos procesos y mejorar la eficiencia operativa. Este capítulo explora en detalle algunas de las principales herramientas disponibles, incluyendo **Pulumi**, **Ansible**, **Chef**, **SaltStack**, **Google Cloud Deployment Manager**, **Azure Resource Manager** y **Kubernetes Operators.** Se examinan sus características, ventajas y desventajas, con el fin de proporcionar un panorama completo sobre cómo estas herramientas pueden ser utilizadas y adaptadas para satisfacer necesidades específicas.

Pulumi

Descripción: Pulumi es una plataforma innovadora de infraestructura como código que permite gestionar y desplegar recursos en la nube utilizando lenguajes de programación convencionales como JavaScript, TypeScript, Python, Go, y .NET. A diferencia de otras herramientas que emplean lenguajes de configuración específicos, Pulumi permite utilizar lenguajes de programación estándar, facilitando la integración con librerías y herramientas de desarrollo existentes.

Lenguajes de Programación: Soporta lenguajes como JavaScript, TypeScript, Python, Go, y .NET.

Integración: Compatible con una amplia variedad de proveedores de nube, incluyendo AWS **[Amazon Web Services. (n.d.)**], Azure, Google Cloud y Kubernetes **[Hightower, K., Burns, B., & Beda, J. (2017)**].

Estado: Mantiene un estado de la infraestructura para gestionar cambios y actualizaciones.

Ventajas:

- Uso de herramientas y librerías de programación existentes.
- Facilita la integración con aplicaciones y sistemas ya en uso.

Desventajas: Curva de aprendizaje alta para usuarios no familiarizados con los lenguajes soportados.Puede ser menos intuitivo comparado con lenguajes de configuración específicos.

Chef

Descripción: Chef es una herramienta de automatización de configuración que utiliza "recetas" y "cookbooks" para definir y gestionar la infraestructura. Está diseñada para manejar la configuración de sistemas a gran escala y se centra en la automatización y el mantenimiento de los sistemas.

Lenguaje: Utiliza Ruby para definir recetas y cookbooks.

Agentes: Requiere un agente Chef instalado en los nodos gestionados.

Gestión del Estado: Utiliza un servidor central (Chef Server) para almacenar el estado de la configuración.

Ventajas:

- Potente para la configuración de sistemas a gran escala.
- Amplia gama de configuraciones y personalizaciones.
- Integración con herramientas de automatización de procesos de TI y despliegue.

Desventajas:

- La necesidad de agentes puede ser una desventaja en entornos más pequeños o menos dinámicos.
- Utiliza Ruby, que puede no ser familiar para todos los usuarios.

SaltStack

Descripción: SaltStack, ahora parte de VMware, es una herramienta de automatización y gestión de configuración que se destaca por su rapidez y escalabilidad. Utiliza una arquitectura de mensajería para permitir la comunicación en tiempo real y la ejecución de comandos.

Lenguaje: Utiliza YAML para definir estados y configuraciones.

Agentes: Requiere un agente llamado "salt minion" en los nodos gestionados.

Arquitectura: Emplea una arquitectura de mensajería para una comunicación rápida y escalable.

Ventajas:

- Alta velocidad y capacidad para manejar grandes cantidades de nodos.
- Arquitectura flexible y escalable.
- Soporta una amplia variedad de configuraciones y gestiona tanto infraestructura como aplicaciones.

Desventajas:

- La complejidad de la arquitectura puede ser un desafío para la configuración y gestión.
- La necesidad de agentes en los nodos puede ser vista como una desventaja en algunos casos.

Google Cloud Deployment Manager

Descripción: Google Cloud Deployment Manager es una herramienta específica para Google Cloud Platform (GCP) que permite definir y gestionar recursos mediante

plantillas en YAML o Python. Está diseñada para facilitar la gestión de la infraestructura en GCP.

Lenguaje: Utiliza YAML o Python para definir plantillas.

Integración: Ofrece integración nativa con Google Cloud Platform.

Plantillas: Permite la creación de plantillas modulares y reutilizables.

Ventajas:

- Integración profunda con Google Cloud Platform.
- Plantillas modulares y reutilizables.
- Gestión eficiente de recursos en GCP.

Desventajas:

- Limitado a Google Cloud Platform, lo que puede ser una desventaja en entornos multi-nube.
- El formato YAML puede ser más complejo para algunos usuarios en comparación con otros lenguajes.

Azure Resource Manager (ARM)

Descripción: Azure Resource Manager (ARM) es la herramienta de infraestructura como código de Microsoft Azure. Permite la gestión y el despliegue de recursos en Azure mediante plantillas en formato JSON.

Lenguaje: Utiliza JSON para definir plantillas.

Integración: Ofrece integración nativa con Microsoft Azure.

Gestión del Estado: Utiliza grupos de recursos y políticas para la gestión de la infraestructura.

Ventajas:

- Integración completa con Microsoft Azure.
- Gestión centralizada de todos los recursos de Azure.
- Plantillas modulares y reutilizables.

Desventajas:

- Limitado a Microsoft Azure, lo que puede ser una desventaja en entornos multi-nube.
- El formato JSON puede ser menos intuitivo y más verboso en comparación con otros lenguajes de configuración.

Kubernetes Operators

Descripción: Los Kubernetes Operators son extensiones del control de clústeres de Kubernetes que permiten gestionar aplicaciones y servicios complejos dentro de un clúster de Kubernetes. Utilizan la API de Kubernetes para automatizar el ciclo de vida de estas aplicaciones.

Lenguaje: Desarrollado en el lenguaje de programación del Operator (como Go o Python).

Integración: Se integra directamente con el clúster de Kubernetes.

Automatización: Facilita la gestión del ciclo de vida de aplicaciones complejas, incluyendo despliegue, escalado y actualizaciones.

Ventajas:

- Permite gestionar aplicaciones complejas dentro de Kubernetes.
- Aprovecha las capacidades nativas de Kubernetes para la orquestación y gestión.
- Facilita la automatización de tareas repetitivas y complejas.

Desventajas:

- Puede ser complejo de desarrollar y gestionar, especialmente para aplicaciones no diseñadas para Kubernetes.
- Requiere un conocimiento profundo de Kubernetes y sus APIs.

Ansible

Descripción: Ansible, desarrollado por Red Hat, es una herramienta de automatización de TI que facilita la configuración de sistemas, el aprovisionamiento de infraestructura y el despliegue de aplicaciones. Utiliza un lenguaje de configuración basado en YAML, conocido como playbooks, para describir el estado deseado de los sistemas.

Lenguaje: Utiliza YAML para definir configuraciones en playbooks.

Agentes: No requiere agentes en los sistemas gestionados; utiliza SSH para la comunicación.

Idempotencia: Las configuraciones son idempotentes, aplicándose múltiples veces sin cambiar el resultado más allá de la primera aplicación.

Ventajas:

- Configuración simple y fácil de aprender.
- No requiere software adicional en los nodos gestionados.
- Extensible con módulos personalizados y roles.

Desventajas:

- Menos enfocado en la gestión de infraestructura como código en comparación con Terraform **[Brikman, Y. (2017)]**.
- Puede ser menos eficiente para grandes infraestructuras debido a la necesidad de SSH para cada acción.

Tecnología elegida

En la búsqueda de una herramienta adecuada para la gestión y creación automatizada de infraestructura, se consideraron varias tecnologías destacadas. A continuación, se presenta un cuadro comparativo que resume las principales características, ventajas y desventajas de cada herramienta evaluada:

Tecnología	Descripción	Características	Ventajas	Desventajas	Lenguaje
Pulumi	Plataforma de infraestructura como código que usa lenguajes de programación.	- Soporta JavaScript, TypeScript, Python, Go, .NET. - Compatible con múltiples proveedores de nube. - Mantiene estado de infraestructura.	- Usa lenguajes de programación estándar. - Facilita la integración con aplicaciones existentes.	- Curva de aprendizaje alta para usuarios no familiarizados. - Menos intuitivo que lenguajes de configuración específicos.	JavaScript, TypeScript, Python, Go, .NET
Ansible	Herramienta de automatización que gestiona configuración y despliegue mediante YAML.	- Usa YAML en playbooks. - No requiere agentes en nodos gestionados (usa SSH). - Configuraciones idempotentes.	- Configuración simple y fácil de aprender. - No requiere software adicional en nodos.	- Menos enfocado en infraestructura como código comparado con Terraform. - Menos eficiente para grandes infraestructuras.	YAML
Chef	Herramienta de automatización de configuración que usa recetas y cookbooks en Ruby.	- Usa Ruby para definir configuraciones. - Requiere agentes en nodos gestionados. - Gestiona estado a través de Chef Server.	- Potente para configuraciones a gran escala. - Amplia gama de configuraciones. - Integración con otras herramientas.	- Requiere agentes en nodos. - Usa Ruby, que puede no ser familiar para todos los usuarios.	Ruby

SaltStack	Herramienta de automatización y gestión con alta velocidad y escalabilidad.	- Usa YAML para definir configuraciones. - Requiere agentes (salt minions). - Arquitectura basada en mensajería.	- Alta velocidad y capacidad de manejar muchos nodos. - Arquitectura escalable. - Amplia gama de configuraciones.	- Complejidad en la arquitectura. - Necesita agentes en los nodos.	YAML
Google Cloud Deployment Manager	Herramienta de infraestructura como código para Google Cloud Platform (GCP).	- Usa YAML o Python para plantillas. - Integración nativa con GCP. - Plantillas modulares.	- Integración profunda con GCP. - Plantillas reutilizables. - Gestión eficiente en GCP.	- Limitado a Google Cloud Platform. - Formato YAML puede ser complejo para algunos usuarios.	YAML, Python
Azure Resource Manager (ARM)	Herramienta de infraestructura como código para Microsoft Azure.	- Usa JSON para plantillas. - Integración nativa con Azure. - Gestión mediante grupos de recursos y políticas.	- Integración completa con Azure. - Gestión centralizada de recursos. - Plantillas reutilizables.	- Limitado a Azure. - Formato JSON puede ser menos intuitivo y más verboso.	JSON
Kubernetes Operators	Extensiones para Kubernetes que gestionan aplicaciones y servicios complejos.	- Desarrollado en Go o Python. - Integración directa con Kubernetes. - Automatiza el ciclo de vida de aplicaciones.	- Gestión de aplicaciones complejas en Kubernetes. - Aprovecha capacidades nativas de Kubernetes. - Automatización de tareas.	- Complejidad en desarrollo y gestión. - Requiere conocimiento profundo de Kubernetes.	Go

Tras evaluar estas herramientas en función de sus capacidades y compatibilidad con los requisitos específicos de la empresa, se concluyó que la solución empaquetada más adecuada fue Ansible. Esta elección se basó en su flexibilidad y capacidad para ser extendida, permitiendo adaptarse a las necesidades particulares que no estaban completamente cubiertas **"out-of-the-box"** por otras soluciones y se le agregaron, en esta implementación "***custom***" realizada ciertas ideas tomadas de algunas de las otras herramientas, como por ejemplo la idea de "***recetas***" tomada de **Chef**.

Ansible se destacó no solo por su simplicidad y facilidad de configuración, sino también por su capacidad para integrarse sin necesidad de agentes adicionales y su idempotencia en la aplicación de configuraciones. Aunque otras herramientas, como Pulumi y Terraform **[Kennedy, B. (2019) y HashiCorp. (n.d.)]**, ofrecían características robustas, Ansible demostró ser la más alineada con las necesidades y restricciones específicas del entorno de la empresa, facilitando una implementación eficaz y adaptable a largo plazo.

Modelo de madurez

En el marco de la evolución tecnológica y la necesidad de optimizar procesos dentro de la empresa, se ha propuesto un modelo de madurez específico para la adopción de la metodología Tier-Core. Este modelo ha sido diseñado para medir y guiar tanto a los equipos de desarrollo como a las aplicaciones en su transición hacia un entorno más ágil y eficiente, alineado con las mejores prácticas de la industria.

Etapas del Modelo de Madurez

El modelo de madurez para la adopción de **Tier-Core** se estructura en cuatro etapas secuenciales que representan el progreso desde una fase inicial de familiarización hasta un estado avanzado de integración. Cada etapa tiene características definidas y objetivos claros que deben ser alcanzados para avanzar al siguiente nivel.

Nivel 1 - Engaged
En la primera etapa del modelo de madurez, denominada "**Engaged**", los equipos y aplicaciones están en las primeras fases de interacción con la metodología **Tier-Core**. Este nivel se caracteriza por la familiarización inicial con los conceptos y prácticas de Tier-Core, pero aún no se han integrado completamente en los flujos de trabajo diarios. Durante esta fase, los equipos comienzan a interactuar con la metodología, explorando sus beneficios y adaptándola lentamente a su entorno operativo.

Nivel 2 - Onboarded
Una vez que los equipos han superado la etapa de familiarización, se procede al nivel "**Onboarded**". En esta etapa, los equipos han completado el proceso de incorporación formal a **Tier-Core** y han comenzado a aplicar activamente sus principios en los proyectos. Aunque la metodología se utiliza de manera regular, aún no ha alcanzado un grado de madurez donde sea la norma en todos los procesos y decisiones. Esta etapa representa una adopción activa pero parcial, con un enfoque en la consolidación de prácticas y el ajuste continuo de procesos.

Nivel 3 - Tier-Core First
En el tercer nivel de madurez, conocido como "**Tier-Core First**", la metodología **Tier-Core** se convierte en el enfoque principal para la gestión de proyectos y desarrollo dentro de la empresa. En este punto, **Tier-Core** se aplica de manera consistente y es la primera opción en la toma de decisiones, planificación y ejecución de proyectos. Este nivel indica que la metodología ha sido completamente adoptada y es un pilar central en la operativa diaria de los equipos. La empresa, en su conjunto, ha alcanzado un grado significativo de madurez en la adopción de **Tier-Core.**

Nivel 4 - Ready for N2A Decom
La etapa final del modelo, "**Ready for N2A Decom**", señala un nivel avanzado de madurez en la adopción de **Tier-Core**. Aquí, tanto los equipos como las aplicaciones están preparados para una transición y descomposición hacia N2A, una arquitectura más avanzada que representa el siguiente paso en la evolución de la metodología. Este nivel implica no solo una adopción total de **Tier-Core,** sino también una disposición y capacidad para evolucionar las estructuras existentes hacia una arquitectura más ágil y adaptable. Alcanzar este nivel demuestra que la organización está lista para abordar desafíos más complejos y adaptarse a cambios tecnológicos futuros.

Situación Actual y Perspectivas Futuras

En la actualidad, el nivel promedio de adopción de **Tier-Core** a nivel empresarial se sitúa en el Nivel 3, "**Tier-Core First**". Esto refleja que, en términos generales, la metodología se ha convertido en el enfoque predominante dentro de la organización. Sin embargo, es importante destacar que existen variaciones entre las distintas verticales de negocio, con algunas áreas aún ubicadas en la etapa 2, "**Onboarded**". Estas diferencias indican que, aunque se ha logrado un avance significativo, todavía queda trabajo por hacer para alcanzar una adopción completa y uniforme.

De cara al futuro, la empresa se ha propuesto un objetivo ambicioso: alcanzar el Nivel 4, "**Ready for N2A Decom**", en los próximos 12 a 24 meses. Este objetivo representa no solo la culminación del proceso de adopción de Tier-Core, sino también la preparación para una evolución hacia arquitecturas más avanzadas. El plan incluye iniciativas estratégicas que buscan cerrar las brechas actuales en la adopción y asegurar que todos los equipos y aplicaciones estén alineados con las metas de madurez establecidas.

El avance hacia el Nivel 4 será un hito significativo en la transformación digital de la empresa, marcando el punto en el que la organización no solo habrá adoptado plenamente **Tier-Core,** sino que también estará lista para enfrentar los desafíos del futuro con una base tecnológica sólida y adaptativa.

Conclusiones

La transición del ciclo de vida de desarrollo de software (**SDLC**) de la empresa desde una estructura de ambientes centralizados y "perennes" hacia un esquema de ambientes completamente automatizados ha representado un cambio monumental en la manera en que gestionamos nuestra infraestructura. En el modelo anterior, los entornos no tenían un plazo de expiración definido y sólo experimentaban "data refreshes" una o dos veces al año. Estos procesos implicaban la realización de "dumps" de los datos de producción en los ambientes de desarrollo, previa ofuscación de datos sensibles (**PII**). Además, gran parte de la creación y mantenimiento de estos entornos estáticos requería una intervención manual significativa por parte del equipo de "IT Operations".

Este enfoque tradicional no solo resultaba en ineficiencias operativas, sino que también presentaba desafíos significativos en términos de escalabilidad, agilidad y seguridad. La dependencia de procesos manuales limitaba nuestra capacidad para responder rápidamente a las demandas del negocio y a los cambios en el entorno tecnológico.

La implementación de un esquema de ambientes cuya creación está totalmente automatizada marca un hito en la evolución de nuestras prácticas de desarrollo. En este nuevo modelo, hemos introducido medidas estrictas para evitar la dependencia de intervenciones manuales, asegurando que cualquier tarea que requiera interacción humana sea insostenible a largo plazo debido a la destrucción periódica y frecuente de los ambientes. Por ejemplo, los ambientes satelitales son destruidos cada tres semanas, mientras que las aplicaciones del "**core**" tienen un ciclo de vida de tres meses. Además, hemos eliminado la posibilidad de migrar solo una parte de los microservicios al "**core**" y apuntar a otros servicios en un ambiente no core, ya que ambos operan en dominios de red separados sin conectividad entre sí.

Esta estrategia no solo optimiza nuestros procesos internos, sino que también refleja un compromiso firme por parte de la empresa de alinearse con las mejores prácticas y tendencias tecnológicas globales. Al adoptar un enfoque basado en Infrastructure as Code (**IaC**), nos posicionamos al nivel de las principales empresas tecnológicas del mundo, como Amazon, Google, Netflix, Microsoft, Airbnb, Spotify, Facebook, Red Hat y Salesforce. Estas organizaciones han implementado iniciativas similares, tanto para su

funcionamiento interno como para los servicios que ofrecen a sus clientes, demostrando la eficacia y la relevancia de IaC en el contexto empresarial actual.

En resumen, la transición hacia una infraestructura automatizada y efímera no sólo ha mejorado nuestra eficiencia operativa, sino que también ha fortalecido nuestra capacidad para innovar y adaptarnos rápidamente a las necesidades del mercado. Esta iniciativa subraya el compromiso de la empresa de mantenerse a la vanguardia tecnológica, adoptando prácticas que no solo son sostenibles y escalables, sino que también reflejan un entendimiento profundo de las tendencias globales en el desarrollo de software.

Mis conclusiones personales

La implementación de las herramientas de automatización necesarias para crear infraestructura on-demand ha sido un proceso extenso y demandante en términos de recursos. Este proyecto ha requerido más de dos años de desarrollo y estabilización, involucrando la colaboración de cientos de equipos de desarrollo y una inversión significativa de varios millones de dólares. A pesar de que este esfuerzo ha sido considerable tanto en tiempo (más de tres años) como en recursos financieros, hasta el momento no se ha alcanzado un retorno de inversión positivo. Sin embargo, la tendencia muestra signos alentadores, ya que los sistemas están demostrando una estabilidad mucho mayor en comparación con unos meses atrás. De hecho, muchos equipos han avanzado hacia la etapa 2 de madurez y adopción, conocida como "**Tier-Core first**".

Este nuevo paradigma sigue evidenciando un gran potencial para mejorar la disponibilidad de los entornos al reducir incidentes y fallos, optimizando así la calidad del desarrollo a través de entornos completamente automatizados y versionados. Además, ha acelerado significativamente la identificación, desarrollo y despliegue de nuevas funcionalidades, reduciendo el tiempo necesario para la configuración de la infraestructura y mejorando la seguridad mediante prácticas como el enmascaramiento de datos. En resumen, aunque el retorno de inversión aún no ha alcanzado un punto completamente positivo, las mejoras en la estabilidad y las ventajas en términos de soporte, mantenimiento y ciclo de vida del desarrollo de software indican que el verdadero potencial de este nuevo paradigma está en camino de ser plenamente realizado.

Trabajos futuros

El estado actual de la creación y gestión de infraestructura está experimentando una evolución que recuerda a la transformación vivida por los ciclos de vida del desarrollo de software. En el pasado, los modelos tradicionales de desarrollo de software, como el enfoque en cascada, fueron reemplazados por metodologías ágiles que introdujeron sprints y ciclos de iteración más cortos pero más frecuentes. Este cambio permitió una mayor flexibilidad y una entrega más rápida y continua de funcionalidades al tiempo que permitió una adaptabilidad al cambio mucho mayor.

De manera análoga, la gestión de infraestructura está siguiendo una trayectoria de modernización. En el contexto descrito en esta tesis, el proceso de gestión de infraestructura actualmente se desarrolla a través de múltiples ambientes de testing, que incluyen n2a, n1, n0 y finalmente el entorno de pruebas de rendimiento (perf), antes de llegar al entorno de producción. Este enfoque en cascada puede implicar un ciclo que dure aproximadamente un mes, desde que el código ingresa al ambiente más bajo hasta su despliegue en producción. Este proceso extenso refleja una metodología que prioriza la estabilidad y la exhaustividad en las pruebas, pero que puede resultar en ciclos prolongados.

En contraste, el nuevo enfoque propuesto ofrece una alternativa que permite un ciclo de vida mucho más corto. En este nuevo modelo, el desarrollo y la prueba inicial se realizan en un entorno de satélite, el cual está diseñado para evitar cualquier riesgo para la estabilidad de los componentes principales del sistema. Una vez que se completan estas pruebas iniciales, se procede a realizar un ciclo de pruebas de integración en el entorno central o núcleo (**core**). Esta metodología permite ciclos de vida más breves y frecuentes, alineándose con la tendencia global hacia la entrega continua (**Continuous Delivery**), que busca mejorar la eficiencia y la agilidad en la gestión de infraestructura.

Un análisis detallado de esta evolución podría constituir un área de trabajo futuro relevante. Este análisis podría incluir una evaluación exhaustiva de los costos asociados con la migración hacia el nuevo enfoque, así como una comparación de los beneficios y posibles inconvenientes que este nuevo modelo presenta. La comprensión de estos factores proporcionará una perspectiva más clara sobre la viabilidad y el impacto de adoptar ciclos de vida más cortos en la gestión de infraestructura, facilitando la toma de decisiones informadas para futuras implementaciones.

Glosario

SDLC (El Software Development Life Cycle) es un marco que define las fases y etapas del proceso de desarrollo de software, desde la concepción inicial hasta el despliegue y el mantenimiento. El SDLC proporciona un enfoque estructurado para desarrollar software de alta calidad de manera eficiente y controlada. Generalmente, incluye fases como recopilación de requisitos, análisis, diseño, desarrollo, pruebas, implementación y mantenimiento. Cada fase tiene actividades y objetivos específicos, lo que ayuda a los equipos de desarrollo a gestionar el proceso, minimizar errores y garantizar que el software cumpla con los requisitos y expectativas del cliente.

Foreman: Foreman es una plataforma de gestión de infraestructura de código abierto que permite a los administradores de sistemas automatizar tareas relacionadas con la gestión de servidores. Con Foreman, se puede aprovisionar, configurar, monitorear y mantener servidores en entornos físicos, virtuales y en la nube. Ofrece integración con herramientas de automatización como Puppet y Ansible, y facilita la gestión centralizada de la infraestructura y la colaboración entre equipos.

ALB (Application Load Balancer): Es una tecnología desarrollada por Amazon Web Services (AWS). Fue creado como parte de la gama de servicios de balanceo de carga de AWS, diseñada para operar a nivel de aplicación y ofrecer capacidades avanzadas de enrutamiento y balanceo de carga en entornos de nube. El ALB forma parte de la familia de Elastic Load Balancers (ELB) de AWS, y está diseñado para distribuir el tráfico entre múltiples instancias de aplicaciones, proporcionando escalabilidad, alta disponibilidad y tolerancia a fallos, además de permitir enrutamiento basado en contenido y otras funcionalidades avanzadas.

XLD: XL Deploy es una herramienta empresarial de despliegue de software. Permite automatizar, coordinar y gestionar el proceso de despliegue de aplicaciones en diversos entornos, desde desarrollo hasta producción. Con XL Deploy, los equipos pueden definir y controlar procesos de despliegue, reducir errores humanos y garantizar implementaciones consistentes y confiables a través de entornos on-premise o en la nube.

XLR: XL Release es una herramienta empresarial de coordinación de lanzamientos de software. Permite planificar, rastrear y ejecutar planes de lanzamiento desde el código inicial hasta el usuario final. Los equipos pueden gestionar tareas, hitos y dependencias, supervisar el progreso en tiempo real y automatizar procesos para una entrega de software más eficiente y confiable.

Etcd: Almacén de clave-valor open source, distribuido y uniforme que permite la configuración compartida, la detección de servicios y la coordinación del programador de clústeres o sistemas distribuidos de máquinas.

CoC (Convention over Configuration): Este principio implica que, en lugar de requerir configuraciones extensas y detalladas, las herramientas y aplicaciones adoptan convenciones predeterminadas para deducir información de manera automática. Esto

simplifica el proceso de configuración y reduce la cantidad de decisiones que los desarrolladores deben tomar. En el ejemplo citado, las URL completas de los servicios se construyen siguiendo convenciones predefinidas, lo que facilita la consistencia y la implementación sin errores.

PII (Personally Identifiable Information): se refiere a cualquier dato que pueda ser utilizado para identificar de manera única a una persona. Esto incluye información como el nombre completo, la dirección de residencia, el número de teléfono, el número de Seguro Social, el número de tarjeta de crédito, la fecha de nacimiento, el número de pasaporte, la dirección de correo electrónico y los datos de cuentas bancarias. La protección de la PII es crucial para garantizar la privacidad y la seguridad de los individuos, ya que la exposición de esta información puede llevar a riesgos como el robo de identidad, el fraude y otras formas de abuso. En el desarrollo de software y la gestión de datos, se utilizan prácticas como la ofuscación y el cifrado para proteger la PII y asegurar que los datos sensibles sean manejados de manera segura.

DRY (Don‘t Repeat Yourself): Este principio busca eliminar la redundancia en el código y la configuración. Cuando se utiliza DRY, se minimiza la duplicación, lo que facilita el mantenimiento y reduce la probabilidad de errores. En el caso mencionado, el uso de convenciones y plantillas evita la repetición de elementos como paths o configuración de ALBs, lo que contribuye a un desarrollo más eficiente.

Bibliografía

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

Brikman, Y. (2017). Terraform: Up & Running. O'Reilly Media.

Kennedy, B. (2019). Terraform: From Beginner to Advanced. Independently Published.

HashiCorp. (n.d.). Official Terraform Documentation. terraform.io/docs.

Jourdan, S., & Pomes, P. (2020). Infrastructure as Code Cookbook. Packt Publishing.

Brikman, Y. (2020). Mastering Infrastructure as Code. O'Reilly Media.

Hightower, K., Burns, B., & Beda, J. (2017). Kubernetes: Up & Running. O'Reilly Media.

Poulton, N. (2017). The Kubernetes Book. Independently Published.

Lukša, M. (2017). Kubernetes in Action. Manning Publications.

Pressman, R. S. (1982). Software Engineering: A Practitioner's Approach. McGraw-Hill Education.

Shore, J., & Warden, S. (2007). The Art of Agile Development. O'Reilly Media.

Larman, C. (1997). Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. Pearson Education.

Wood, J., & Tannous, B. (2021). OpenShift for Developers: A Guide for Impatient Beginners O'Reilly Media.

Picozzi, S., & Hepburn, M. (2018). DevOps with OpenShift: Cloud Deployments Made Easy. Packt Publishing.

Gavant, J., & Dumpleton, G. (2018). Mastering OpenShift. Packt Publishing.

Amazon Web Services. (n.d.). AWS CloudFormation: User Guide. aws.amazon.com.

Beach, B. (2020). AWS CloudFormation for Developers: A Beginner's Guide. Independently Published.

Petit, P. (2021). AWS CloudFormation: Simplified Infrastructure Management on AWS. Packt Publishing.

Geerling, J. (2015). Ansible for DevOps: Server and Configuration Management for Humans. Independently Published.

Hochstein, L., & Moser, R. (2015). Ansible: Up and Running. O'Reilly Media.

Keating, J. (2015). Mastering Ansible. Packt Publishing.

Turnbull, J. (2014). The Docker book: Containerization is the new virtualization. Independently published.