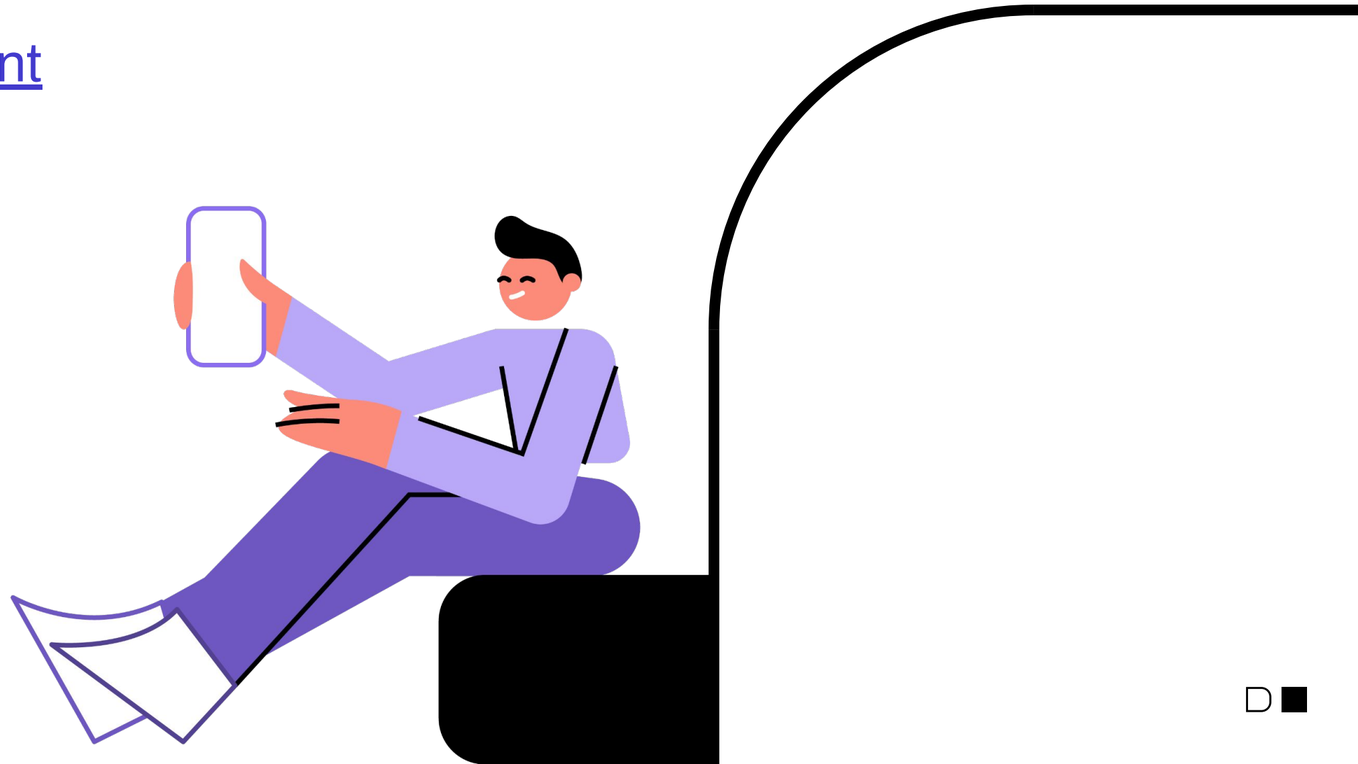
DIGITALHOUSE

Implementamos GitOps a fondo

Certified Tech Developer
The Ultimate Degree

Índice



01

Introducción a Environment Management

Introducción a Environment Management

En la implementación de software, un entorno es donde se implementa y ejecuta el código. Diferentes entornos sirven para diferentes propósitos en el ciclo de vida del desarrollo de software.

Por ejemplo, un entorno de desarrollo local (una computadora) es donde los ingenieros pueden crear, probar y depurar nuevas versiones de código. Una vez que los ingenieros completan el desarrollo del código, el siguiente paso es confirmar los cambios en Git y comenzar una implementación en diferentes entornos para realizar pruebas de integración y, finalmente, lanzar la producción. Este proceso se conoce como integración continua/implementación continua (CI/CD) y generalmente consta de los siguientes entornos: QA, E2E, Stage y Prod.

Introducción a Environment Management

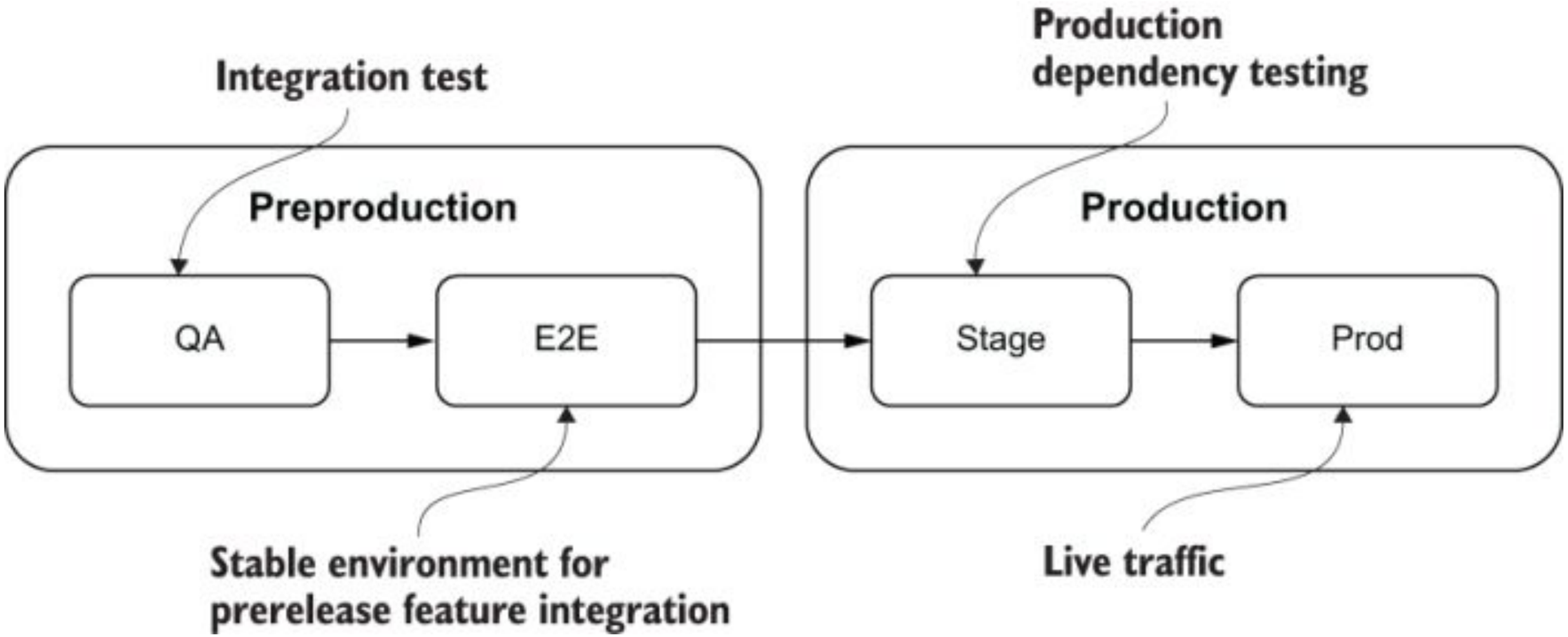
El entorno de control de calidad es donde el nuevo código se probará con el hardware, los datos y otras dependencias similares a las de producción para garantizar la corrección de su servicio. Si todas las pruebas pasan el control de calidad, el nuevo código se promocionará al entorno E2E como un entorno estable para que otros servicios preliminares lo prueben o se integren. Los entornos de control de calidad y E2E también se conocen como entornos de preproducción (preprod) porque no alojan tráfico de producción ni utilizan datos de producción.

Environment Management

- Nueva versión
- Despliegue a Stage
- Pruebas
- Despliegue a producción

Cuando la aplicación requiere una nueva actualización, la misma es desplegada a un entorno Stage para realizar minuciosas pruebas antes del despliegue a producción. Una vez que todas las pruebas garanticen el correcto funcionamiento, el nuevo código finalmente es implementado en producción, donde recibirá el tráfico real de los usuarios.

Introducción a Environment Management



La preproducción tiene un entorno de control de calidad para las pruebas de integración y un entorno E2E para la integración de características preliminares. Los entornos de producción pueden tener un entorno de prueba para las pruebas de dependencia de producción y el entorno de producción real para el tráfico en vivo.

02

Estrategias de despliegues

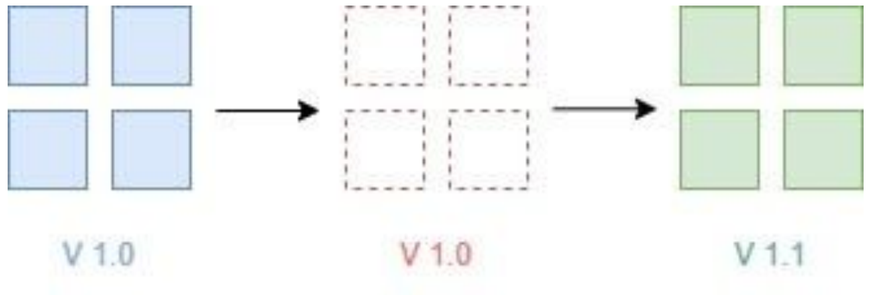
¿Qué es una estrategia de despliegue?

Una **estrategia de despliegue** es parte de un pipeline en el que se define cómo implementaremos las aplicaciones/servicios. Podemos tener un pipeline de despliegue simple, donde la estrategia es para todo el pipeline. Pero podemos tener un pipeline más complejo, donde la estrategia concierne únicamente a una parte del mismo.

Recreate

La estrategia **recreate** es la más simple. Como se menciona en el nombre, esta estrategia se compone de dos pasos:

1. Eliminar la versión actual de la aplicación en el servidor/clúster.
2. Implementar la nueva versión en el servidor/clúster.



Pros

- Simple y rápido de hacer.
- Soportado por una gran cantidad de herramientas de pipeline.

Contras

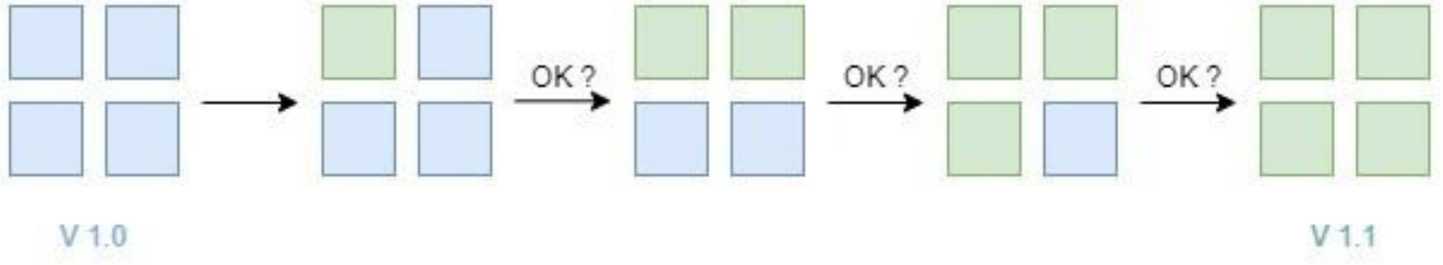
- El servicio no estará disponible durante un período de tiempo. Entonces, si tenemos un SLA alto, claramente no será la mejor opción.
- La nueva versión debe ser compatible con todas las funciones existentes sin interrumpir los cambios (si tenemos algunos cambios importantes, puede ser un problema para nuestros consumidores).

Rolling deployment

Esta estrategia actualizará progresivamente todas las instancias de una aplicación con la nueva versión.

Pasos:

1. Bucle para cada instancia.
2. Matar la instancia existente.
3. Implementar una nueva instancia con la nueva versión.
4. Verificar si la nueva instancia se inició correctamente; si es así, ir a la siguiente instancia.



Pros:

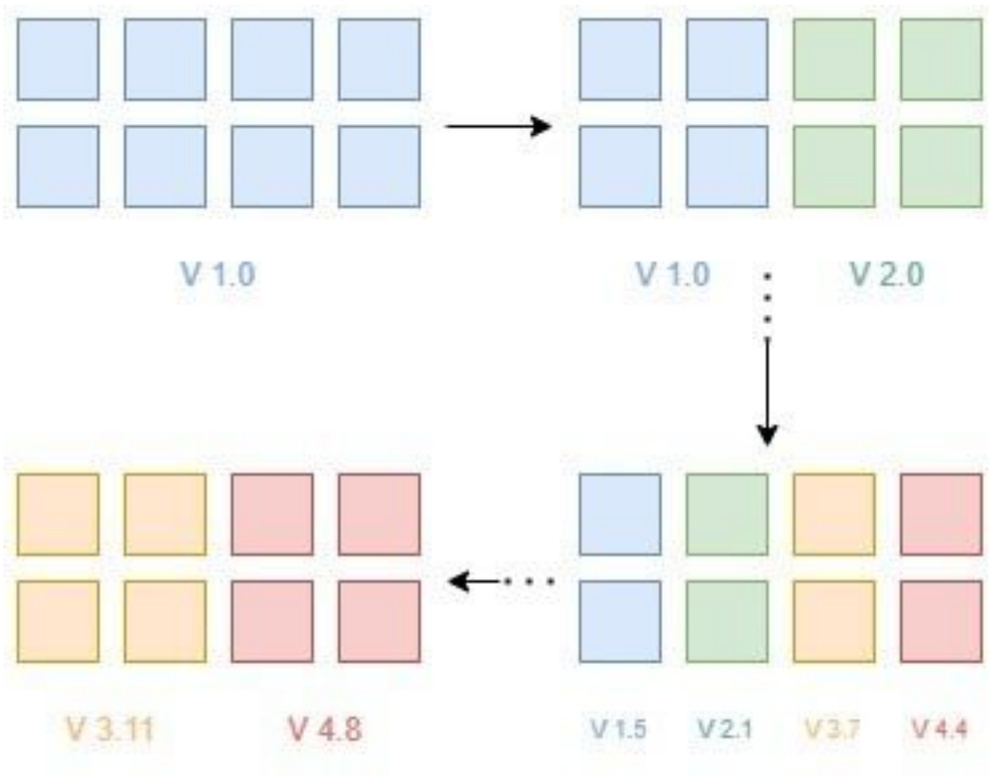
- No afecta la disponibilidad.
- Fácil de implementar en Kubernetes.

Contras:

- La nueva versión debe ser compatible con todas las funciones existentes sin interrumpir los cambios (si tiene algunos cambios importantes, puede ser un problema para nuestros consumidores).

Múltiples versiones

Siguiendo su nombre, esta estrategia intenta admitir múltiples versiones de su aplicación en producción.



Pros:

- Dejará tiempo a los clientes para migrar a la nueva versión.
- La aplicación no necesita admitir las funciones anteriores.

Contras:

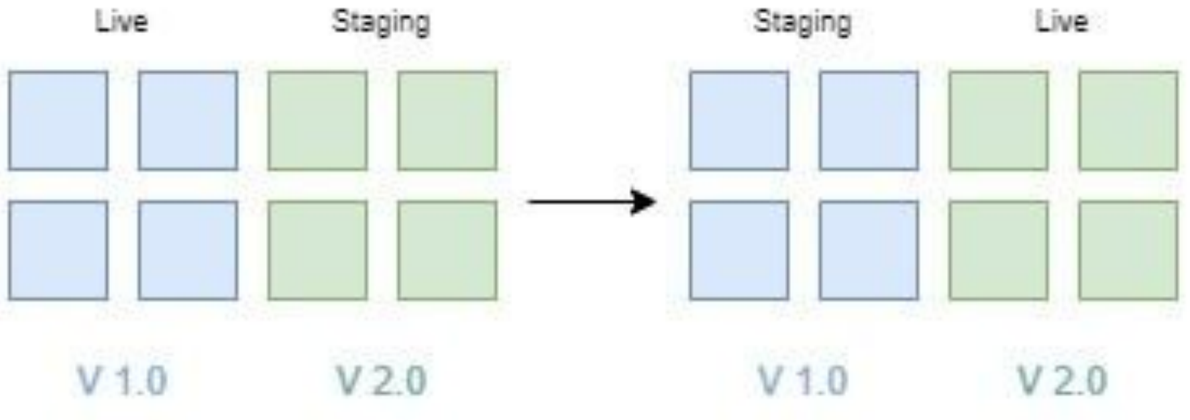
- Lo usamos solo cuando tenemos un cambio importante. Si lo hacemos para cada versión sin interrumpir los cambios, perderemos productividad, especialmente en lo que respecta al soporte.
- Más versiones de producción para soportar. Más versiones tenemos en producción, más versiones tenemos que actualizar/probar cuando ocurre un problema de producción.
- Toma mucho tiempo. Con esta estrategia, tendremos que configurar muchas cosas manualmente (pipelines, configuraciones de enrutamiento).
- Debe pensarse desde la concepción para evitar tener muchos cambios importantes en el futuro.

Esta estrategia puede ser realmente útil en un caso de cambio de última hora. Pero como se explica, puede tener algunas sorpresas. Entonces, si lo usamos, tengamos en cuenta las desventajas y preparemos una estrategia clara para evitar tener 45 cambios importantes al año y admitir 10 versiones simultáneamente. Con estos elementos en mente, deberíamos estar bien.

Blue/Green

Esta estrategia se basa en dos entornos de producción:

- **Entorno productivo** donde se pueden utilizar los servicios.
- **Entorno en staging** donde desplegamos las nuevas versiones de l servicios.



Cuando nuestro entorno de prueba esté listo para la producción (después de las pruebas de calidad), se debe cambiar el tráfico entre los dos entornos.
Si todo está bien en el entorno de prueba, podemos cambiar completamente el tráfico a este entorno, que se convertirá en el nuevo entorno "en vivo".
Si tenemos algunos problemas, volvemos a cambiar todo el tráfico a la versión actual del servicio.

Pros:

- Sencillo de entender.
- Puede ser fácil de configurar con algunas herramientas.
- Rollback simple.

Contras:

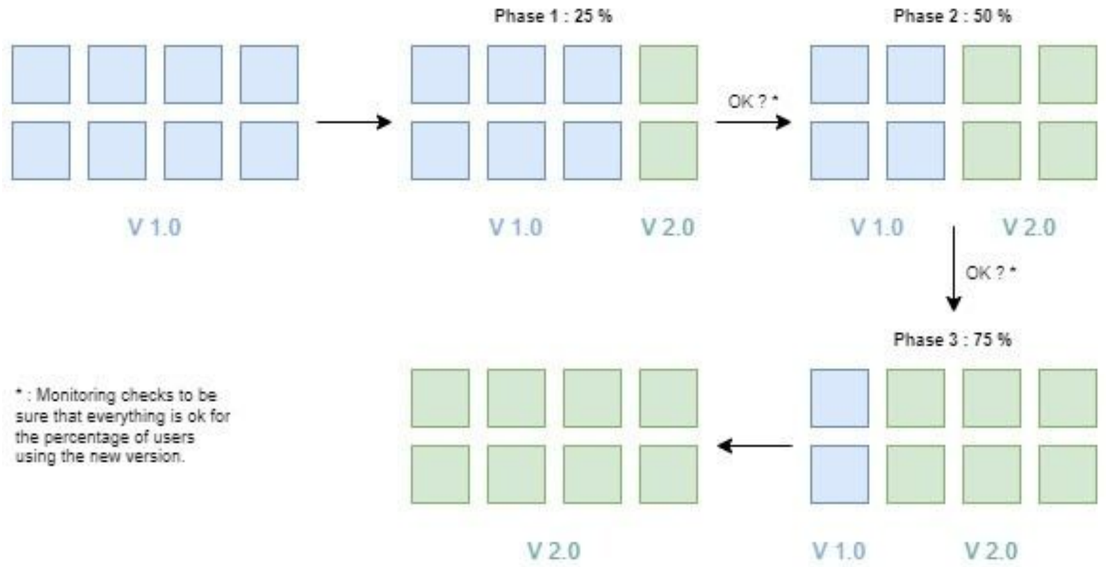
- Puede llevar más tiempo configurarlo con algunas herramientas que no tienen esta función de forma nativa (o si lo estamos haciendo manualmente).
- Puede ser costoso. Tendremos que duplicar al menos las instancias en ejecución del servicio, de modo que tenga todos los recursos disponibles para poder ejecutarlas y volver a implementarlas (puede ser más si estamos utilizando microservicios).
- Puede ser complejo de configurar, especialmente con microservicios, más aún si estamos utilizando algo como Kafka o MQ. Deberemos asegurarnos de que todos los servicios solo se comuniquen con **los otros servicios en el mismo entorno**.

Canary

Esta estrategia consta de migrar progresivamente a nuestros usuarios a la nueva versión.

¿Cómo hacerlo?

Reemplazaremos algunas de sus instancias actuales por la nueva versión y cambiaremos parte del tráfico en ella. Luego podemos verificar si todo está bien para este grupo en la nueva versión. En caso afirmativo, podemos aumentar el porcentaje de usuarios que usan la nueva versión y continuar reemplazando las instancias con la versión "antigua" hasta que se actualicen todas las instancias. Si no, podemos retroceder fácilmente a la versión "antigua".



Pros:

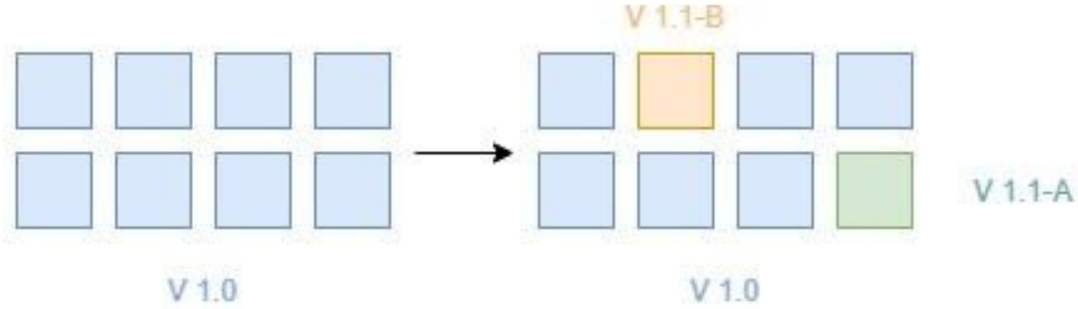
- Más fácil y económico de configurar que una estrategia Blue/Green.
- Solución rápida y segura para migrar en producción.

Contras:

- Debe tener más preparación/instalación/configuraciones en su lugar en caso de un cambio importante.
- Como estrategia Blue/Green, puede ser complejo configurar microservicios, especialmente si estamos utilizando un servicio de cola como Kafka o MQ.
- Se debe implementar una configuración de monitoreo para ver correctamente lo que sucede en ambas versiones.

A/B Testing

A/B Testing es una estrategia técnicamente similar a la estrategia Canary. Implementaremos una nueva versión en un número limitado de instancias y cambiaremos algo de tráfico en ella. Pero aquí el objetivo es hacer algunas experimentaciones con múltiples ideas y ver cuál funciona bien. Es una buena implementación para las aplicaciones frontend, para ver si los usuarios no están aburridos o no abandonan en medio de la secuencia para buscar/hacer algo.



Pros:

- Realmente poderoso para recuperar algunas impresiones del mundo real.
- Fácil de configurar.

Contras:

- Mentalidad experimental. Haremos pruebas y experimentaciones. Algunos elementos pueden romper la aplicación, por lo que, dependiendo de la criticidad de su aplicación, debemos tomar algunas precauciones.
- Como es para experimentaciones, puede ser largo y/o complejo configurar pruebas automatizadas.

03 Secrets

GitOps, Kubernetes y Secrets

Los profesionales de Kubernetes GitOps, invariablemente, se encontrarán con el mismo problema: mientras que los usuarios se sienten perfectamente cómodos almacenando la configuración en Git, cuando se trata de datos confidenciales no están dispuestos a almacenar esos datos en Git, debido a problemas de seguridad. Git se diseñó como una herramienta colaborativa, lo que facilita que varias personas y equipos obtengan acceso al código y vean los cambios de los demás. Pero estas mismas propiedades también son las que hacen que usar Git para mantener Secrets sea una práctica extremadamente peligrosa. Hay muchas preocupaciones y razones por las que no es apropiado almacenar Secrets en Git, que trataremos a continuación.

¿Cuáles son esas razones?

Sin encripción

Kubernetes no proporciona cifrado en el contenido de un Secret, y la codificación Base64 de los valores debe considerarse como texto sin formato. Además, Git por sí solo no proporciona ningún tipo de cifrado integrado.

Repos Git distribuidos

Con GitOps, nuestros colegas clonarán localmente el repositorio de Git en sus computadoras, con el fin de administrar la configuración de las aplicaciones. Al hacerlo, también estaremos distribuyendo Secrets a muchos sistemas, sin una auditoría o un seguimiento adecuados. Si alguno de estos sistemas se viera comprometido, alguien obtendría acceso a todos sus Secrets.

Control de acceso No Granular

Git no proporciona protección de lectura de una subruta o subarchivo de un repositorio de Git. No es posible restringir el acceso a algunos archivos en el repositorio de Git pero no a otros. Cuando se trata de Secrets, idealmente debería otorgar acceso de lectura a los Secrets en función de la necesidad de conocerlos.

Almacenamiento inseguro

Git nunca tuvo la intención de ser utilizado como un sistema de gestión de Secrets por lo que no incorporó al sistema funciones de seguridad estándar como el cifrado en reposo. Por lo tanto, un servidor Git comprometido tendría el potencial de filtrar también los Secrets de todos los repositorios que administra, lo que lo convertiría en un objetivo principal para el ataque.

Historial de commits

Una vez que se agrega un Secret al historial de commits de Git, es muy difícil eliminarlo. Si el Secret se registra en Git y luego se elimina, ese Secret aún se puede recuperar consultando un punto anterior en el historial del repositorio antes de que se elimine el Secret.

Estrategias de gestión de Secrets

Hay muchas estrategias diferentes para manejar Secrets en GitOps que aportan flexibilidad, capacidad de administración y seguridad. Repasemos algunas de las estrategias a nivel conceptual.

Almacenamiento de Secrets en Git

La primera estrategia de GitOps en cuanto a Secrets es no tener ninguna estrategia. En otras palabras, simplemente comprometeríamos los Secrets en Git como cualquier otro recurso de Kubernetes y aceptaríamos las consecuencias de seguridad.

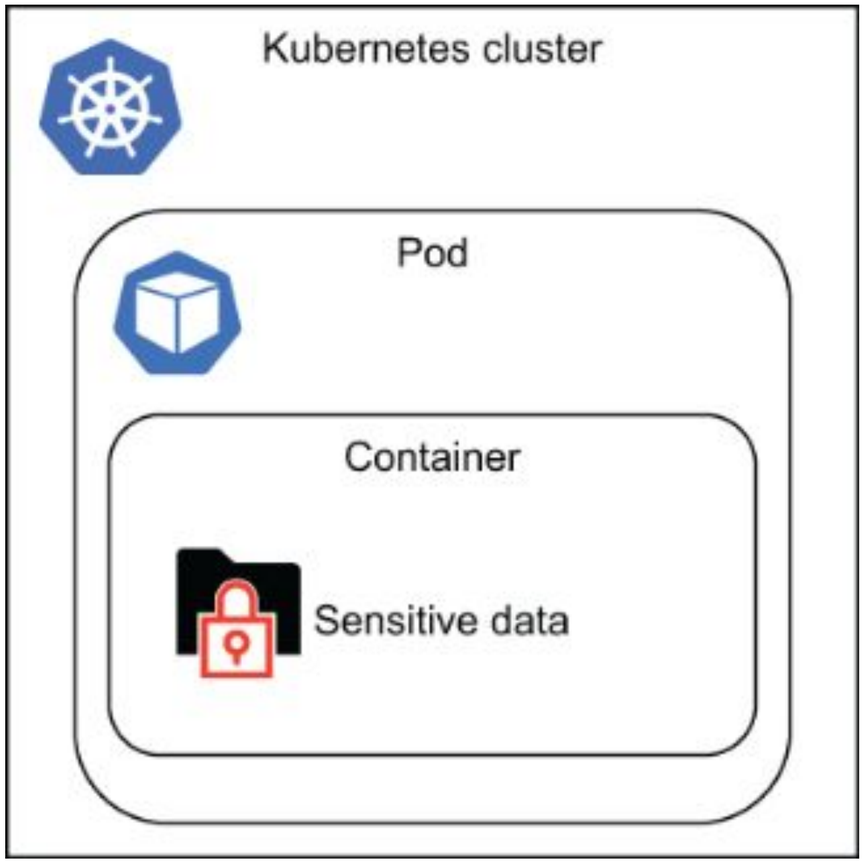
Podríamos estar pensando: ¿Qué tiene de malo almacenar Secrets en Git? Incluso si tenemos un repositorio privado de GitHub, al que solo pueden acceder los miembros de confianza de nuestro equipo, es posible que deseemos permitir el acceso de terceros al repositorio de Git: sistemas CI/CD, escáneres de seguridad, análisis estáticos, etc. Al proporcionar el repositorio Git de Secrets a estos sistemas de software de terceros, a su vez les estamos confiando los Secrets.

Por lo tanto, en la práctica, los únicos escenarios realmente aceptables en los que los Secrets se pueden almacenar, tal como están en Git, son cuando los Secrets no contienen datos verdaderamente confidenciales, como los entornos de desarrollo y prueba.

Secrets en la imagen del contenedor

Una estrategia confiable para evitar almacenar Secrets en Git es tenerlas directamente en la imagen del contenedor. En este enfoque, los datos Secrets se copian directamente en la imagen del contenedor como parte del proceso de compilación de Docker.

- Cualquier persona o cosa que tenga acceso a la imagen del contenedor ahora puede copiar y recuperar el Secret.
- Si las Secrets están integradas en la imagen, siempre el proceso de rotación de Secrets será muy engorroso.
- La imagen del contenedor no es lo suficientemente flexible para adaptarse cuando la misma imagen necesita ejecutarse utilizando diferentes Secrets.

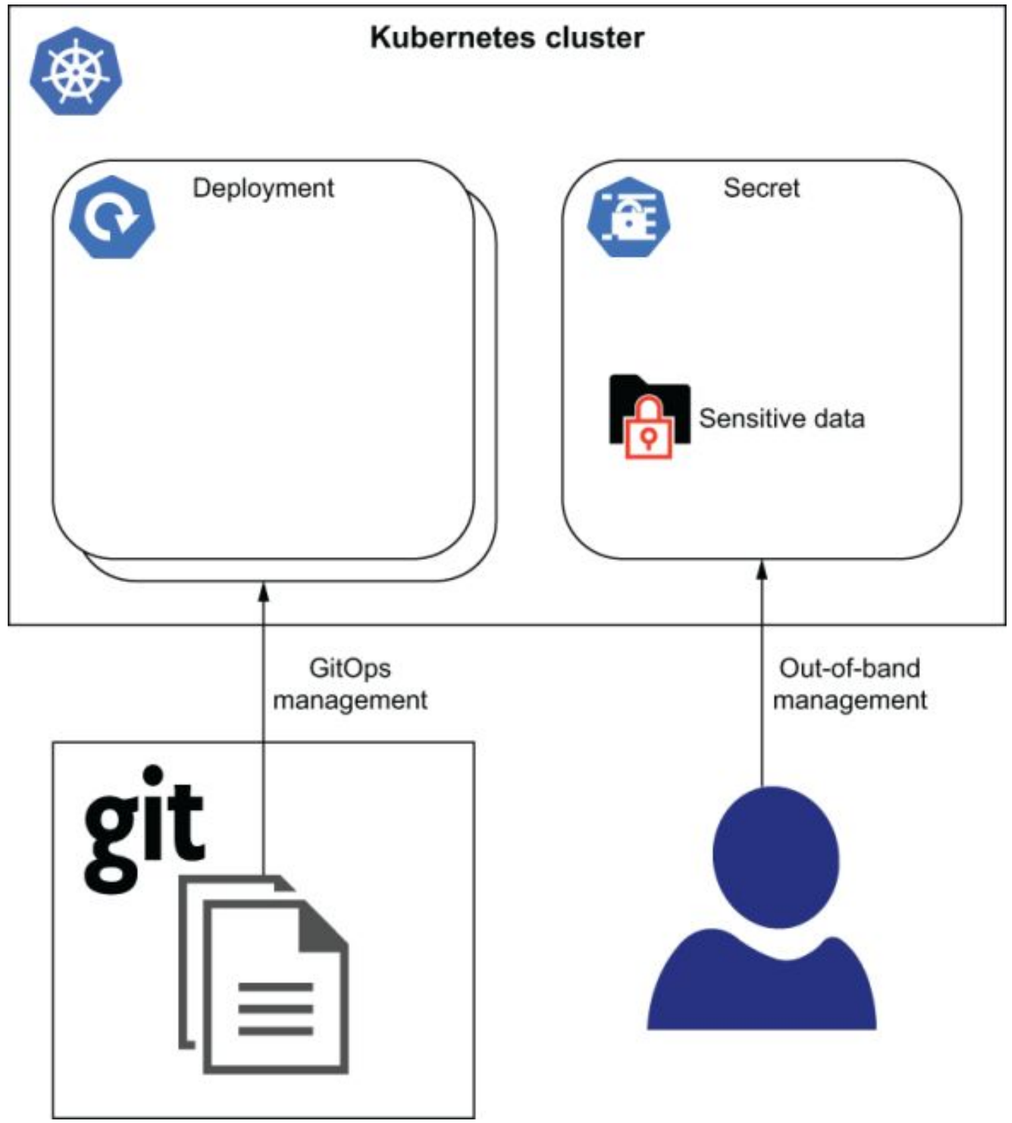


Evitemos almacenar datos sensibles en las imágenes

Gestión externa de Secrets

Otro enfoque para manejar Secrets en GitOps es administrarlas completamente fuera de GitOps, de esta manera todo, excepto los Secrets de Kubernetes, se define en Git y se implementaría a través de GitOps, pero se usaría algún otro mecanismo para implementar Secrets, incluso si fuera manual.

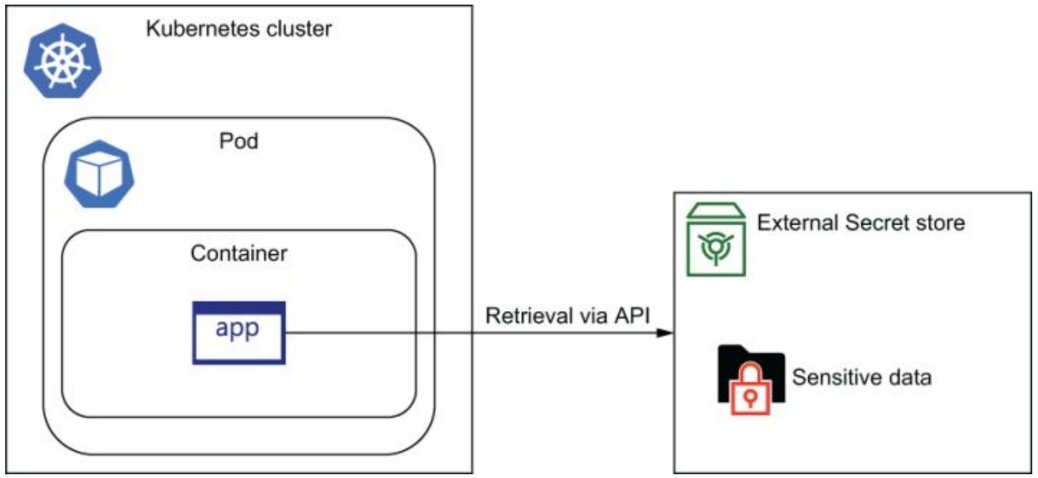
La desventaja obvia es que necesitaría dos mecanismos diferentes para implementar recursos en el clúster: uno para los recursos normales de Kubernetes a través de GitOps y otro estrictamente para Secrets.



Sistemas de gestión de Secrets externos

Otra estrategia para manejar Secrets en GitOps es usar un sistema de gestión de Secrets externo que no sea Kubernetes. En esta estrategia, en lugar de usar las funciones nativas de Kubernetes para almacenar y cargar Secrets en el contenedor, los propios contenedores de la aplicación recuperan los valores Secrets dinámicamente en tiempo de ejecución, en el punto de uso.

Existe una variedad de sistemas de gestión de Secrets, pero el más popular y ampliamente utilizado es HashiCorp Vault. Los proveedores de nube individuales también brindan sus propios servicios de administración de Secrets, como AWS Secrets Manager, Google Cloud Secret Manager y Microsoft Azure Key Vault. Las herramientas pueden diferir en cuanto a capacidades y conjuntos de características, pero los principios generales son los mismos y deben ser aplicables a todos.



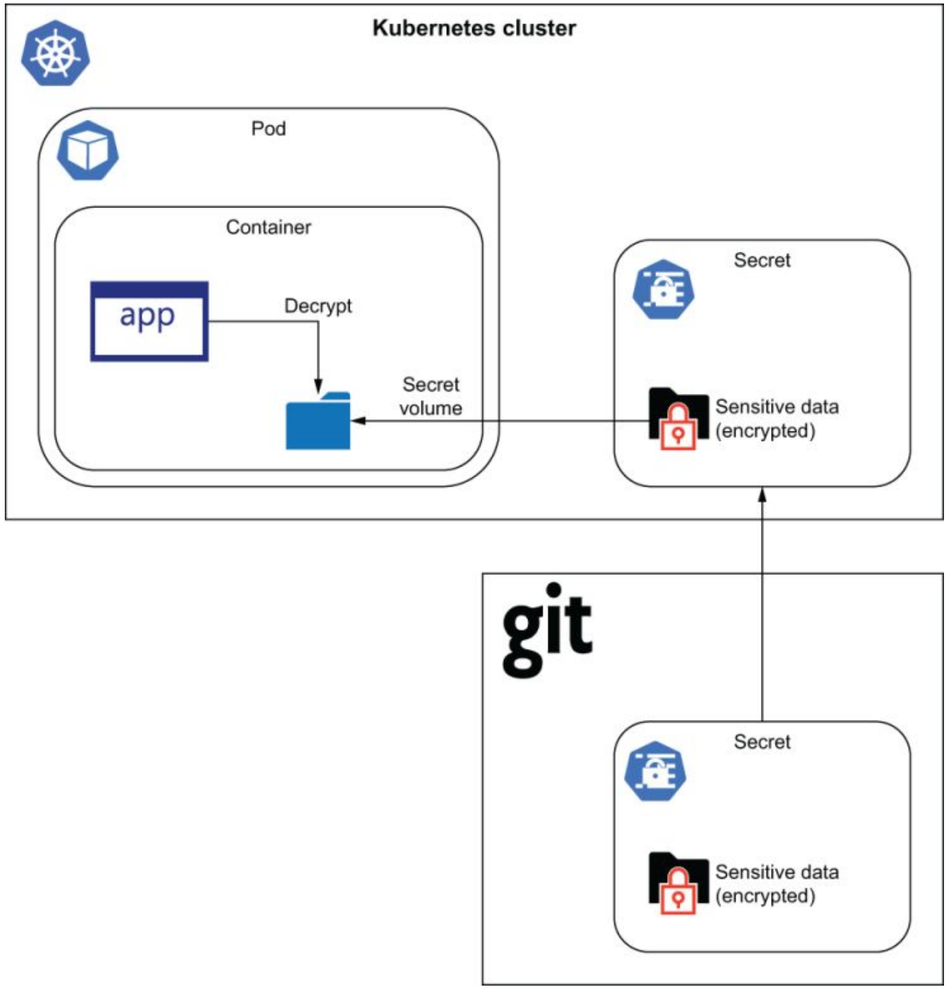
Eligiendo un sistema de manejo de secretos externo (como Vault) para administrar los Secrets, también estamos tomando la decisión de no usar los Secrets de Kubernetes. Esto se debe a que, al utilizar esta estrategia, confiamos en el sistema de gestión de Secrets externo para almacenar y recuperar sus Secrets, y no en Kubernetes. Una consecuencia es que tampoco podrás aprovechar algunas de las ventajas que brindan los Secrets de Kubernetes, como establecer el valor de una variable de entorno de un Secret o asignar el Secret como archivos en un volumen.

Cuando se utiliza una tienda de Secrets externa, es responsabilidad de la aplicación recuperar los Secrets de la tienda de forma segura. Por ejemplo, cuando se inicia la aplicación, podría recuperar dinámicamente los valores Secrets del almacén de Secrets en tiempo de ejecución, en lugar de utilizar los mecanismos de Kubernetes (variables de entorno, montajes de volumen, etcétera). Esto traslada la carga de la custodia de los Secrets tanto a los desarrolladores de aplicaciones que deben recuperar el Secret de forma segura como a los administradores del almacén de Secrets externo.

Otra consecuencia de esta técnica es que debido a que los Secrets se administran en una base de datos separada, no tiene el mismo historial/registro de cuándo se cambiaron los Secrets que tiene para su configuración administrada en Git. Esto podría incluso afectar su capacidad para retroceder de manera predecible. Por ejemplo, durante una reversión, aplicar los manifiestos en la confirmación de Git anterior podría no ser suficiente. Además, tendría que revertir el Secret a un valor anterior al mismo tiempo que la confirmación de Git. Dependiendo de qué tienda secreta se haya utilizado, esto podría ser un inconveniente en el mejor de los casos, o absolutamente imposible en el peor de los casos.

Cifrado de Secrets en Git

Dado que Git no se considera seguro para almacenar Secrets de texto sin formato, una estrategia es cifrar los datos confidenciales para que sea seguro almacenarlos en Git y luego descifrar los datos cifrados más cerca de su punto de uso. El actor que realiza el descifrado tendría que tener las claves necesarias para descifrar el Secret cifrado. Esta podría ser la propia aplicación, un contenedor de inicio que llena un volumen utilizado por la aplicación o un controlador para manejar estas tareas para la aplicación sin problemas.



DIGITALHOUSE

¡Muchas gracias!