

Despliegue y Observabilidad

Software Developers playbook

Transformación desarrollo de software - #ONE

Control de versiones

Versión	Fecha de Creación	Responsable	Descripción
1.0	{17-3-2023}	Global Software Development - GSD	Primera versión oficial de los playbooks

Índice

1. Introducción	4
1.1. Acerca de este Playbook	4
1.2. Principios básicos	5
1.3. Niveles de exigencia	5
2. Prácticas	6
2.1 Aprobar y coordinar el despliegue para desplegar frecuentemente	7
2.1.1 Detectar automáticamente las dependencias e impactos con otros servicios	7
2.1.2 Asegurar despliegues desacoplados	8
2.1.3 Elegir de forma autónoma la ventana de despliegue	9
2.1.4 Automatizar y racionalizar controles y requisitos en un despliegue	10
2.2 Automatizar el proceso de despliegue	13
2.2.1 Utilizar pipelines para orquestrar el despliegue y el rollback	13
2.2.2 Utilizar Infraestructura como código (IaC)	16
2.2.3 Gestionar la configuración como código	17
2.2.4 Gestionar las bases de datos para asegurar retrocompatibilidad	19
2.3 Seleccionar la estrategia de despliegue y aplicar mejores prácticas para eliminar indisponibilidad	23
2.3.1 Elegir la estrategia de despliegue basada en coste, control de impacto y complejidad	23
2.3.2 Despliegue Blue-green	26
2.3.3 Despliegue Canary	27
2.3.4 Rolling Deployment	28
2.3.5 Despliegue de features mediante Feature Toggle	29
2.3.6 Despliegue de features mediante Dark Launches	32
2.4. Mantener completa observabilidad antes, durante y después del despliegue	34
2.4.1 Asegurar trazabilidad completa de la release de forma automática	34
2.4.2 Comunicar el estado y resultado del despliegue a terceros	35
2.4.3 Establecer la monitorización de eventos y alertas	37
2.4.4 Realizar el seguimiento y reaccionar a eventos y alertas	39
2.4.5 Aprender y mejorar de despliegues pasados	40
3. Proceso	43

1. Introducción

1.1. Acerca de este Playbook

Este playbook (parte de una serie de seis) es un componente fundamental para la práctica de desarrollo de software en el Banco que condensa las prácticas y directrices que los ingenieros de software debemos llevar a cabo durante el proceso de despliegue de código y su observabilidad.

El playbook debe ser actualizado regularmente basándose en las prácticas y el panorama tecnológico del Banco. El equipo de GSD (Global Software Development) será el responsable de actualizarlo.



Figura 1: Ciclo de vida del desarrollo de software

1.2. Principios básicos

Este playbook se guía por un conjunto de principios que todos los desarrolladores deben seguir:

- **Estandarización:** Usar una taxonomía global y formas de trabajo alineadas entre los equipos y geografías permitirá a los desarrolladores colaborar mejor, compartir mejor el conocimiento y rotar entre los equipos.
- **Calidad sobre cantidad:** Escribir código de calidad debe ser nuestra prioridad ya que la mayor parte de nuestro tiempo lo empleamos manteniendo software heredado y abordando deuda técnica, cumplir con esto mejorará la velocidad de desarrollo de los equipos a medio y largo plazo.
- **Transparencia y trazabilidad:** El total control y visibilidad de los sistemas y el código del Banco mejora la experiencia de los desarrolladores para navegar en los distintos entornos, responder a incidentes y garantizar el cumplimiento de los requisitos regulatorios.
- **Propiedad:** La alta demanda de calidad requerida en los servicios críticos del banco implica un control detallado de responsabilidad y propiedad de los activos de código. Debemos ser propietarios del código que desarrollamos y sentirnos orgullosos de ello.
- **Pequeños incrementos frecuentes frente a grandes cambios:** Grandes cantidades de código pueden poner en riesgo la estabilidad de nuestros sistemas ya que son más difíciles de probar y deshacer. Además, la mayoría de los sistemas de software evolucionan en función de los comentarios de los usuarios finales y los requisitos son difíciles de anticipar de antemano. Por tanto, es importante desarrollar rápidamente pequeños incrementos para aprender rápidamente y rectificar.
- **Simplicidad:** Los sistemas de software rara vez se escriben una vez y no se modifican. Más bien, muchas personas trabajan con el mismo sistema a lo largo del tiempo. Por eso es importante buscar las soluciones más sencillas que puedan comprenderse fácilmente y evolucionar en el futuro.
- **Automatización:** Los procesos manuales que son repetitivos deben automatizarse para mejorar nuestra productividad, enfocarnos en el desarrollo y reducir los errores mediante la intervención manual.

1.3. Niveles de exigencia

Este playbook utiliza intencionadamente las siguientes tres palabras para indicar los niveles de exigencia según la norma [RFC2119](#):

- **Debe** - Significa que la práctica es un requisito absoluto, salvo las excepciones que se autoricen de forma específica.
- **Debería** - Significa que pueden existir razones válidas en circunstancias particulares para ignorar una práctica, pero deben comprenderse todas las implicaciones y sopesar cuidadosamente antes de elegir un camino diferente.
- **Puede** - Significa que la práctica es realmente opcional.

2. Prácticas

Todos los equipos de software (desarrollo, arquitectura, operaciones...) **deben** seguir las **cuatro prácticas** siguientes relativas a despliegue y observabilidad, independientemente de la tecnología utilizada:

1. [Aprobar y coordinar el despliegue para desplegar frecuentemente](#)
2. [Automatizar el proceso de despliegue](#)
3. [Seleccionar la estrategia de despliegue y aplicar mejores prácticas para eliminar indisponibilidad](#)
4. [Mantener completa observabilidad antes, durante y después del despliegue](#)

Estas prácticas tienen como objetivo que los equipos de desarrollo automaticen e incrementen la frecuencia de los despliegues de modo que se reduzcan los errores en el entorno de producción (los entornos previos y su propósito se detallan en el [playbook de testing](#)). Los equipos cuyos despliegues cumplan con las prácticas detalladas a continuación, deben tener la posibilidad de desplegar de forma habitual y con menor cantidad de restricciones (e.g. *fast track* debería contemplar sobre todo las prácticas mencionadas en este playbook).

A continuación se definen cada una de las prácticas de este playbook siguiendo la siguiente estructura:

- Sus beneficios (que conseguimos).
- Precondiciones (condiciones para poder aplicar la práctica).
- Adopción (cómo implementarla).
- Herramientas necesarias.
- Indicadores para los desarrolladores (como la medimos).
- Enlaces de interés.

2.1. Aprobar y coordinar el despliegue para desplegar frecuentemente

Los equipos de desarrollo de software deben seguir las siguientes prácticas con el objetivo de desplegar lo más frecuentemente posible, de un modo ágil y desacoplado minimizando de este modo los riesgos asociados al despliegue. De esta forma, no deberían existir bloqueos que no sean propios de la naturaleza del servicio o del negocio sobre el que estén trabajando. Para conseguir esto se identifican cuatro líneas de trabajo:

1. Detectar automáticamente las dependencias e impactos con otros servicios
2. Asegurar despliegues desacoplados
3. Elegir de forma autónoma la ventana de despliegue
4. Automatizar y racionalizar controles y requisitos en un despliegue

Adopción

2.1.1. Detectar automáticamente las dependencias e impactos con otros servicios

Beneficios

- Planificar la estrategia de despliegue y pruebas acorde a las dependencias identificadas
- Mayor confiabilidad a la hora de realizar cambios
- Facilitar la detección de errores o incidencias
- Anticipar dependencias en evolutivos
- Facilitar la detección de componentes obsoletos
- Anticipar la reserva de capacidad de esos equipos para tareas como pruebas de regresión o actualización de versión de un componente o servicio
- Coordinar posibles downtimes
- Facilitar siempre que sea posible el fast track en los despliegues

Precondiciones

- Las arquitecturas y aplicaciones deben facilitar la información de trazabilidad acorde a la implementación de la herramienta.
- Los equipos de desarrollo deben disponer de un servicio único y global de publicación de cambios planificados, accesible para todos los desarrolladores y contextualizado por geografía.

Adopción

La detección automática de dependencias entre servicios es importante a la hora de conseguir automatizaciones a lo largo del proceso de despliegue:

- Los desarrolladores **deben** disponer de una herramienta capaz de mantener la información de dependencias de forma actualizada, no declarativa, distinguiendo por entorno
- La herramienta **debe** tener un nivel de detalle suficiente para identificar el componente de software de forma inequívoca (por ejemplo, mapeado al repositorio).
- La información **debe** ser fiable, completa y actualizada para evitar falta de confianza.

- La herramienta **debería** incluir también información sobre la infraestructura

2.1.2. Asegurar despliegues desacoplados

Beneficios

- Agilizar los despliegues al necesitar menos coordinación.
- Mejorar el time-to-market de los distintos aplicativos reduciendo las dependencias en tiempo de despliegue.

Adopción

Para desplegar más frecuentemente y con poco riesgo es importante plantear las implantaciones de forma desacoplada e independiente, evitando a toda costa los temidos *BigBang*, donde todos los equipos deben subir los cambios de los diferentes componentes al mismo tiempo al no ser viable la actualización desacoplada de los componentes.

Gracias a este punto, los equipos pueden planificar su **despliegue de forma autónoma** siempre que prioricen aquellas acciones que mitiguen o reduzcan a cero el impacto en otros equipos.

Dada la complejidad de ciertas aplicaciones habrá situaciones en las que será complicado garantizar un desacoplamiento total del aplicativo. En dichos casos los responsables del aplicativo **deben** dedicar tiempo a **desacoplar los despliegues cuando sea posible** y detallar los casos en los que no sea posible comunicando el tipo de acoplamiento o dependencia (e.g., datos, software, infraestructura) para facilitar los despliegues e incluso el proceso de rollback si fuera necesario.

Realizar esta práctica requiere aplicar las siguientes directrices:

- Las APIs de los servicios y componentes **deben** ser **retrocompatibles** (para todas las geografías) y estar versionadas. Esto permite que haya posibilidad de realizar despliegues desacoplados y que exista libertad para elegir ventanas de despliegue sin necesidad de coordinación con otros equipos.
- **Debería** existir un **contract testing** adecuado con el resto de componentes (más detalle en el **playbook de Testing**).
- Si hay indisponibilidad en el sistema, cualquier consumidor del servicio o componente puede tener un problema, por tanto, los despliegues **deberían** realizarse sin indisponibilidad siempre que sea posible y el aplicativo lo necesite (e.g., aplicativos usados por el cliente, componentes críticos de alto uso). Para más información sobre estrategias sin indisponibilidad ir a la **práctica 2.3.1**.
- El equipo de desarrollo **puede** usar feature toggles para desacoplar despliegues

Indicadores

Indicador	Descripción
-----------	-------------

Afectaciones a terceros	Indicador que recoge número de número de aplicaciones impactadas por el despliegue
Equipos involucrados en el despliegue	Número de equipos que participan en despliegue. El objetivo es optimizar reduciendo el número de equipos

2.1.3. Elegir de forma autónoma la ventana de despliegue

Beneficios

- Mejorar Time to Market, al poder los equipos elegir cuando desplegar sin acoplamientos temporales externos.
- Mejorar la colaboración con negocio porque al desplegar más a menudo se reduce el tiempo del feedback loop.
- Usar más óptimo del tiempo y recursos haciendo una asignación dinámica del tiempo disponible para despliegues.
- Aumentar la previsibilidad del despliegue, evitando retrasos por motivos ajenos al propio despliegue.
- Automatizar tareas de planificación.

Precondiciones

- El equipo debe disponer de pipelines de despliegue y rollback automatizado. Definido en la [práctica 2.2.1](#).
- Debe existir un proceso de validación y pruebas automatizadas
- Debe existir una monitorización para reconocer horas de pico y valle
- Los equipos de desarrollo deben disponer de un servicio único y global de publicación de cambios planificados, accesible para todos los desarrolladores y contextualizado por geografía."

Adopción

Como previamente se ha comentado, es muy importante conseguir la mayor autonomía posible de los equipos para decidir cuándo realizar los despliegues. Esto implica un cambio de cultura en relación a los mismos.

Realizar esta práctica requiere aplicar las siguientes directrices:

- El equipo que libera una nueva versión de software **debe** responsabilizarse de elegir correctamente la **ventana** de tiempo en la que se liberará la **nueva versión**. Idealmente los desarrolladores solamente deberían tener en cuenta las circunstancias y restricciones propias del servicio para elegir el **momento del despliegue**.
- El Service Owner **debe** aplicar las reglas de **freezing y scope** asegurando que no va a haber problemas en el despliegue por una mala elección del momento de despliegue. Para más información visitar el [Playbook del Service Owner](#).

- **Debe** existir una **coordinación estrecha** con el resto de equipos afectados cuando no pueda garantizarse la disponibilidad.
- Los desarrolladores que van a desplegar **deben** comunicarse antes del despliegue con equipos que realicen la monitorización cuando no sea el propio equipo.
- Todos los despliegues **deben** ser publicados con suficiente antelación (a criterio del equipo de desarrollo y dependiente del impacto del cambio) en una plataforma donde se recojan todos los cambios planificados y organizados, al menos, por servicio y geografía.
- También se **debe** informar a los equipos, responsables y Service Owners que consumen el servicio (no clientes finales) sobre los cambios que supongan modificaciones en su consumo.
- La plataforma de publicación **debe** ofrecer la posibilidad de suscribirse para recibir novedades de los cambios en un servicio o del impacto a sus dependencias.
- Los desarrolladores **deberían** suscribirse para recibir alertas sobre los servicios con los que trabajan habitualmente.
-

Indicadores

Indicador	Descripción
Despliegues autónomos	Diferencia del número de despliegues autónomos entre el mes en curso y el anterior

2.1.4. Automatizar y racionalizar controles y requisitos en un despliegue

Precondiciones

- Se deben conocer de forma precisa los requisitos a satisfacer para los despliegues a priori.

Adopción

En cualquier proceso de release es necesario el cumplimiento de requisitos de stakeholders de diversa naturaleza y la aportación de evidencias de dicho cumplimiento. Los requerimientos **deben**:

- Ser públicos para los equipos que van a desplegar
- Estar actualizados
- Ser conocido su responsable
- Conocerse el propósito del requerimiento, de forma que el equipo entienda en profundidad la necesidad de su solicitud.

Para que la presentación de evidencias sea automática y lo más razonable posible:

- Previo al despliegue, el equipo de desarrollo **debe** recoger la información necesaria susceptible de ser auditable.
- Los desarrolladores **deben** relacionar las evidencias presentadas y el despliegue.
- Se **debe** realizar seguimiento al estado y tiempo que conlleva la aprobación/desaprobación de cada requisito. Esto permite estudiar si hay stakeholders que ralenticen los despliegues.
- Los auditores, reguladores y demás revisores **deben** requerir única y exclusivamente aquello que sea **imprescindible** en términos de normativa legal.
- En caso de que haya algún problema con las evidencias presentadas, los revisores **deben** comunicarlo al equipo encargado del despliegue en la mayor brevedad de tiempo posible.
- Las auditorías que se hagan sobre los despliegues o el software en relación a cumplimiento normativo **deben** utilizar las evidencias recogidas en cada uno de los despliegues de forma automática.
- **Debe** existir un único lugar para recoger la información por ámbito (servicio, componente, paquete a desplegar)
- Aquel stakeholder que proponga un nuevo requisito **debería** sugerir el modo de automatizarlo y su inclusión en el ciclo de vida.
- Los equipos de desarrollo **pueden** incorporar en los pipelines de despliegue la aprobación de aquello necesario incluyendo validación de requisitos legales y funcionales.

Indicadores

Indicador	Descripción
Promedio de tiempo aprobación por stakeholder	Indicador que recoge el tiempo medio de aprobación de requisitos por stakeholder. Esto permitirá mejorar los tiempos de despliegue
Promedio de tiempo total de la fase de aprobación de requisitos normativos	Este indicador recoge por un lado el tiempo total que tardan los stakeholders en aprobar los requerimientos y, en caso de que no se puedan aprobar, el tiempo que se tarda en resolver los problemas
Requisitos requeridos	Diferencia intermensual del número de requisitos normativos requeridos
Requisitos en pipeline	Diferencia intermensual del número de requisitos normativos incluidos en el pipeline de despliegue

Herramientas (todas las prácticas)

- **Gestor de dependencias (herramienta pendiente de definición):** **Debe** ser un componente custom que debería recoger el análisis de dependencias entre componentes y/o arquitecturas de las diferentes capas de aplicación. Para ello se puede basar en la información de trazabilidad distribuida, en el análisis estático o una mezcla entre ellas (por ejemplo, SCKS¹ o Némesis²).
- **Herramienta de publicación de despliegues** (pendiente de definición): herramienta global en la que los equipos se pueden suscribir a los cambios planificados y ver el estado de la implantación. Ejemplos de herramientas locales parciales usadas son:
 - o https://lookerstudio.google.com/u/0/reporting/11K09gnrZe5iMYKhXxmB7JzO6DQ8X_zig/page/Busn
 - o [Interventions](#)
 - o https://docs.google.com/spreadsheets/d/1x9wK6Zf-ACemVFEIJoFtysJC1fEEQf2_i_mzRRsbdTNM/edit?usp=sharing
- **Servicio integral** para la gestión de proyectos (e.g. **Jira**).
- Herramienta o “Calendario” público de despliegues: la herramienta de gestión de dependencias se utilizará como input, entre otros elementos, para que se puedan planificar despliegues.

Indicadores

Los indicadores que apliquen, **deben** ser proporcionados a los equipos para conocer el grado de madurez respecto a la práctica:

Indicador	Descripción
Porcentaje de despliegues exitosos	Despliegues exitosos / despliegues no exitosos Se definen despliegue exitosos como aquellos que se han desplegado en producción y no han tenido que ser corregidos en producción (ya sea utilizando rollback o hotfix)
Tiempo de coordinación	Tiempo desde que se inicia el proceso de preparación del despliegue hasta que se está listo para desplegar
Tiempo de despliegue	Tiempo de despliegue desde el release en Artifactory hasta Producción
Cambios por despliegue	Variación de número de cambios por despliegue

Enlaces

- <https://martinfowler.com/articles/devops-compliance.html#Patterns>
- <https://martinfowler.com/articles/consumerDrivenContracts.html>

¹ Herramienta usada por el equipo de Turquía para detectar dependencias

² Herramienta de despliegue utilizada en entornos host que permite realizar análisis de impacto, dependencias entre otras cosas

- [https://es.wikipedia.org/wiki/Acoplamiento_\(inform%C3%A1tica\)](https://es.wikipedia.org/wiki/Acoplamiento_(inform%C3%A1tica))
- <https://es.wikipedia.org/wiki/Retrocompatibilidad>

2.2. Automatizar el proceso de despliegue

El proceso de despliegue **debe** ser automatizado para evitar comportamientos inesperados, errores en los despliegues y procesos no reproducibles o dependientes del humano que lo realiza. Esta práctica contribuye a incrementar la frecuencia y facilidad de los despliegues en producción. Para automatizar el despliegue hay cuatro líneas de trabajo que se abordan en esta práctica:

1. Utilizar pipelines para orquestar el despliegue y rollback
2. Utilizar infraestructura como código
3. Gestionar la configuración como código
4. Gestionar las bases de datos para asegurar retrocompatibilidad

Beneficios

- Ejecutar y testear rápidamente en entornos de desarrollo y producción, reduciendo el riesgo de cada despliegue
- Ayudar a dividir y desplegar el código en pequeños incrementos.
- Reducir el time-to-market
- Reducir el error humano al convertirse los despliegues en procesos repetibles
- Simplificar despliegues complejos

Precondiciones

- Se **debe** contar con las herramientas adecuadas y la infraestructura necesaria para soportar la implementación de pipelines

Adopción

2.2.1. Utilizar pipelines para orquestar el despliegue y el rollback

Precondiciones

- **Debe** existir un entorno previo donde se puedan probar estos pipelines

Adopción

Las soluciones técnicas de despliegues no deben impedir en ningún caso y deben estar alineadas para evolucionar hacia la entrega continua o despliegue continuo. La idea es que cualquier nueva versión de código (commit) en el repositorio es susceptible (dependiendo del branching model adoptado) de ser construida y desplegada en producción desde un único pipeline o pipelines enlazados. En este sentido, los desarrolladores **deben** adecuarse al principio de despliegue continuo: **Se construye una vez y se despliega N veces**. Es decir, se construye un solo artefacto y se despliega ese mismo artefacto en todos los entornos (e.g. despliegues multipaís o despliegues en entornos pre-productivos).

Los desarrolladores **deben** contar con [pipelines de despliegue](#) para llevar los desarrollos a producción de una manera repetible, fiable y automatizada.

El rollback **debe** estar automatizado en el pipeline para que, si se detectan errores después de haber hecho la liberación, se pueda volver atrás con seguridad y rapidez. En aquellos casos en los que los errores en producción no puedan arreglarse mediante un rollback, se arreglarán mediante un hotfix, el cual lleva asociado un procedimiento específico³.

Al utilizar pipelines para orquestar despliegue y rollback se requiere que el equipo de desarrollo adopte las siguientes directrices:

- **Deben** contar con un pipeline como código (o crearlo en su defecto) que permita el despliegue de software desde el propio pipeline orquestando las llamadas a las APIs de la plataforma o herramienta correspondiente.
- **Deben** tener la posibilidad de crear desde cero o extender/componer un pipeline para adecuarlo a las necesidades específicas del equipo. Por ejemplo, se **pueden** añadir stages o incluso realizar orquestaciones "complejas" que involucren uno o múltiples componentes de software de la misma o distintas tecnologías.
- El proceso automatizado para realizar un despliegue de software y el proceso para realizar el rollback **debe** ser muy similar, idealmente el mismo. El rollback **debe** dejar el sistema exactamente en el estado anterior.
- El proceso de despliegue y rollback **debe** ser testeado previamente para confirmar que funciona como se espera.
- En los despliegues **debe** haber un lugar indicado y perfectamente accesible por los equipos responsables del despliegue en dónde se guarden los logs del mismo.
- Los despliegues **deben** ser seguros y lo más rápido posibles.
- Todos los miembros del equipo **deberían** ser capaces de hacer un despliegue autónomamente.
- Los equipos *owners* del pipeline **deberían** tener un **versionado de los pipelines**. Si se produce un error de despliegue debido al pipeline de despliegue se podría restaurar la versión anterior al tener éste como código.
- **Debe** haber cobertura amplia de pruebas automatizadas (más detalle en el [Playbook de testing](#)).
- Los equipos que proveen los entornos conjuntamente con el equipo encargado de los pipelines **deben** asegurar que los entornos de preproducción sean lo más parecidos posibles a producción. Más concretamente, los entornos de despliegue de preproducción y producción **deberían** ser idénticos en términos de arquitectura, topología de red, configuración del sistema operativo (incluyendo *patches*) y stack de la aplicación entre otros. No obstante debido al elevado coste que puede suponer duplicar entornos, los entornos de preproducción **pueden** tener capacidad de procesamiento, de disco menores que el entorno de producción.
- Los equipos de desarrollo **deben** usar el mismo pipeline de despliegue para los entornos de preproducción y los entornos de producción. Adicionalmente, si hay réplica de datos en el entorno de preproducción éste **debería** estar en un estado válido.

³ Descrito en la sección [3. Proceso](#)

- Los entornos de preproducción **pueden** ser efímeros. Para mayor detalle, se puede consultar el [playbook de testing](#).
- Si se han pasado los tests de despliegue en preproducción y el despliegue ha sido exitoso en dicho entorno, **debería** comenzar el despliegue en el entorno de producción bien de forma automática o, en aquellos casos que se requiera, después de una aprobación manual. Sin embargo, el rollback en preproducción no debería hacerse de forma automática cuando se realice el rollback en producción.

Indicadores

Los indicadores que apliquen, deben ser proporcionados a los equipos para conocer el grado de madurez respecto a la práctica:

Indicador	Descripción
Despliegues a producción desde pipelines	Porcentaje de los despliegues realizados a producción que se han realizado desde un pipeline automático de despliegue.
Despliegues sin errores	Despliegues sin errores / despliegues totales (esta métrica hace referencia al propio proceso de despliegue y no a bugs en el aplicativo necesariamente)
Manuales para realizar un despliegue	Se tendrán en cuenta el número de pasos manuales para realizar un despliegue
Incidentes resueltos haciendo rollback	Número de incidencias resuelto con rollback automáticos / rollback totales
Efectividad de rollbacks	Número de rollback que terminan correctamente / Número totales de rollbacks
Tiempo medio de despliegue	Variación del tiempo medio en minutos que dura el pipeline de despliegue intermensual
Mean time to Repair (MTTR)	Tiempo de recuperación del servicio entre despliegue y el rollback cuando se detecta un error
Errores detectados en pre-producción	Errores entornos pre-productivos / (Errores entornos productivos + Errores entornos pre-productivos)
Versionado de pipelines de despliegue	Pipelines de despliegue como código y versionadas / pipelines de despliegues totales de equipo
Despliegues parciales	Despliegues parciales / despliegues totales (cuanto menor sea el porcentaje, mejor será la adopción)

2.2.2. Utilizar Infraestructura como código (IaC)

Precondiciones

- La infraestructura implicada **debe** exponer APIs como sucede en las cloud públicas y privadas, sistemas de virtualización, balanceadores, sistemas de almacenamiento etcétera

Adopción

La infraestructura, se define como todos los recursos y servicios necesarios para ejecutar nuestra aplicación (por ejemplo: redes, reglas de seguridad, balanceadores de carga, servidores, almacenamiento, bases de datos, serverless computing, orquestadores de contenedores etcétera).

A día de hoy se ofrecen API para administrar todos estos recursos, así mismo existen diferentes herramientas de software que permiten interactuar con estas APIs de forma programática, es decir, la propia infraestructura se puede crear, evolucionar y destruir como si fuera una aplicación. “La idea habilitante de Infraestructura como código (IaC) es que los sistemas y dispositivos utilizados para ejecutar software pueden tratarse como si ellos mismos fueran software” (Kief Morris), es decir, “La infraestructura como código (para más información visitar [documento anexo](#)) es el enfoque para definir computación y networking a través del código fuente que puede entonces ser tratado como cualquier otro sistema de software”, dado que la infraestructura puede tratarse como software todas las prácticas del desarrollo de software aplican a la Infraestructura, incluido la práctica de usar los pipelines de construcción y despliegue.

Esta práctica requiere la adopción del equipo encargado de la infraestructura, que **puede** ser el equipo de desarrollo, de las siguientes directrices:

- La infraestructura **debe** estar declarada como código. Por tanto, **debe** seguir todas las buenas prácticas de desarrollo de código mencionadas en los otros playbooks. Esto incluye de forma general, tanto las prácticas de [desarrollo y versionado de código](#), como las de [análisis de código y construcción](#), como las de [testing](#) y las de este playbook.
- Los diferentes recursos de infraestructura **deben** estar relacionados apropiadamente entre ellos en el código.
- El equipo encargado de la infraestructura **debe** identificar *drift* o desalineamiento entre el código o las templates y la configuración de los diferentes entornos.
- El código relativo a la infraestructura **debería** modularizarse en componentes más pequeños que **pueden** ser reusados.
- Adicionalmente a los tests habituales, el equipo de desarrollo encargado del despliegue **debería** hacer pruebas de carga con el volumen de carga que va a soportar ese determinado recurso.

Adicionalmente, si la infraestructura está definida como código, ésta **debería** ser [inmutable](#). Es decir, mientras algún recurso que forma parte de la infraestructura esté

corriendo, éste no se verá modificado y cuando sea necesaria una nueva versión del recurso se creará una versión completa de nuevo.

Siempre que sea posible deben usarse [servidores inmutables](#) y tratar los servidores como [ganado y no como mascotas](#). Las directrices que aplican a la inmutabilidad son las siguientes:

- Se **pueden** instanciar Imágenes *Golden* o contenedores cada vez que se realiza un commit y **deben** tener asociado un número de versión o identificador único. Esto permitirá revisar los cambios fácilmente.
- Se **deben** pasar los tests a la nueva imagen o contenedor y registrar el resultado en logs accesibles para los equipos relevantes antes de registrar el contenedor o imagen.
- Los equipos **deben** asegurar que la infraestructura se testea apropiadamente en cada uno de los entornos. Si se produce un error debido a la infraestructura, éste debería ser provocado por los cambios implementados.
- Se **debería** desplegar la nueva versión en un entorno de preproducción antes de desplegar en producción. Se **puede** desplegar en un entorno de desarrollo primero.

Indicadores

Los indicadores que apliquen, deben ser proporcionados a los equipos para conocer el grado de madurez respecto a la práctica:

Indicador	Descripción
% de despliegues fallidos debido a drift entre entorno staging y producción	Porcentaje de despliegues fallidos en producción debido a que los entornos de staging y producción no son semejantes y no se han podido validar correctamente.
Recursos creados en un despliegue que utilizan IaC	Porcentaje de recursos IaC = recursos infraestructura levantados con IAC/recursos totales de infraestructura
Errores set up de infraestructura	Peticiones de infra que terminan en error / Peticiones totales de infra

2.2.3. Gestionar la configuración como código

Adopción

Los equipos de desarrollo **deben** aprovisionar y modificar ficheros de configuración de forma fiable y repetible en los servidores y demás infraestructura implicada en el despliegue. Esta es una parte importante clave que permite a los desarrolladores lograr la automatización de un despliegue.

Para implementar de forma correcta ese tipo de cambios es necesario seguir las siguientes directrices:

- Los equipos de desarrollo **deben** asegurar que se desacopla el código del software de la configuración del mismo.
- Los ficheros de configuración **deben** tener un sistema de versionado.
- Toda la información sensible **debe** residir en un [Vault](#) al que acceden los pipeline de despliegue y las aplicaciones en caso de que sea necesario.
- Los pipelines de construcción y despliegue **deben** ser los encargados de recuperar la información de configuración y añadirla a los ficheros antes de copiarlos en las máquinas destino.
- Todas las modificaciones en las máquinas que requiere un despliegue **deben** estar automatizadas y los scripts y ficheros de configuración deben residir en un repositorio de código.
- Los equipos encargados del despliegue que utilicen sistemas que no gestionen sus cambios de forma automática **deben** aportar una justificación técnica de por qué no es posible automatizarlo y debe ser conocida y aprobada por el comité de cambios.
- Los cambios manualmente que realicen equipos involucrados en el despliegue por carencia de automatización, **deben** poseer un DoD que debe ser revisado y validado por el autor del cambio y visible para los diferentes equipos.
- Todo cambio de configuración **debe** llevar asociado mecanismo de validación del propio cambio.
- Para evitar las desviaciones de configuración los equipos de desarrollo involucrados en el despliegue no **deben** realizar ningún cambio manualmente. Por tanto, los operadores no deberían tener acceso directo a dichas máquinas para este fin. Las excepciones que sirvan para solventar un problema crítico en producción **deben** ser temporales e inmediatamente después codificarlas y re-desplegadas para que el cambio converja y evitar desviaciones de configuración.
- Los service owners o los tech lead **deben** conocer qué partes del despliegue no cumplen con los requisitos de automatización anteriormente mencionados.
- Los service owners **deben** dar su consentimiento previo a la subida del riesgo que supone la falta de automatización, comunicación y/o agilidad.
- **Debería** existir un sistema que detecte drift entre la configuración en el repositorio y la que hay en producción para avisar de que ha existido una manipulación manual y se corrija.

Indicadores

Los indicadores que apliquen, deben ser proporcionados a los equipos para conocer el grado de madurez respecto a la práctica:

Indicador	Descripción
Ficheros de configuración versionados	Ficheros de configuración versionado / Ficheros de configuración totales

Modificaciones manuales de ficheros de configuración	Número de modificaciones manuales de los ficheros de configuración en producción (e.g. implementar cambios, previos al despliegue, de ficheros de configuración)
Errores por falta de automatización	Número de errores por falta de automatización de configuración / Número de errores totales Falta de automatización: modificación manual / directamente en producción de ficheros de configuración
Vulnerabilidades en config	Número de ocasiones en las que se detectan vulnerabilidades de seguridad en la configuración de las aplicaciones: - Ficheros de la aplicación

2.2.4. Gestionar las bases de datos para asegurar retrocompatibilidad

La creación o modificación de los modelos de datos de la aplicación son procesos especialmente críticos en el ciclo de vida de las aplicaciones. El motivo principal, es que almacenan y gestionan los datos de la aplicación, los cuales son la parte más difícil de recuperar o corregir en caso de error al realizar un despliegue. Hay que tener en cuenta 2 perspectivas en la gestión de las bases de datos en un despliegue: la gestión de los esquemas y la gestión de los datos.

En cuanto a la gestión de los esquemas, se debe asegurar la posibilidad de recuperar un estado anterior. Para ello, los desarrolladores de bases de datos y otros equipos encargados del desarrollo y gestión de bases de datos :

- **Deberían** seguir estrategias de [backward compatibility](#) a nivel de base de datos:
 - Se pueden añadir nuevos campos obligatorios con un valor por defecto
 - Se puede añadir un nuevo campo como campo opcional
 - Se puede cambiar el tipo de un campo a un tipo compatible, más genérico.
 - Se pueden generar nuevas entidades
- Los equipos **deberían** seguir [estrategias](#) para transformar cambios no-retrocompatibles en una secuencia de cambios retrocompatibles.
- **Deberían** usar repositorio que permita el control de versiones y versionar el código.
- **Debería** existir un pipeline de despliegue para los esquemas y/o modificaciones de la BBDD. De forma similar a los despliegues de código, se deberían utilizar los pipelines para realizar los rollbacks en aquellos casos en los que hay retrocompatibilidad.
- En caso de que no sea posible, se **debe** definir e implementar un plan de actualización, que incluirá, no solo los script para modificar el esquema, sino también, procesos de backup de la información, transformación de los datos, restauración del backup entre otros.
- **Deben** mantener registro de las versiones generadas de los esquemas y de los scripts (ddl's) para la actualización de la base de datos y su marcha atrás.

- En el caso de los cambios no retrocompatibles, **deben** de asegurar la definición y creación de un plan para restaurar la situación previa.
- Cuando un sistema, con una evolución del modelo de datos entra a operar, se corre el peligro de que no sea posible retroceder a una situación válida, por lo que se **debería** asegurar en el momento del despliegue la validación de los cambios que no se puedan retroceder.
- Se **pueden** mantener scripts para la generación de “líneas base” de esquema, para evitar reconstrucciones de la historia lejana.

Conjuntamente, los equipos encargados de las bases de datos y los desarrolladores de software:

- Para lograr que la evolución y cambios puedan seguir estrategias de [full compatibility](#) el equipo de desarrollo y el equipo que gestiona la base de datos **deben** trabajar coordinadamente.
- Los equipos **deben** coordinar los despliegues con la evolución del modelo y el software, buscando el desacoplamiento de los despliegues de código y de base de datos.
- Los equipos **deben** mantener el tracking de las versiones de base de datos asociadas a las versiones de código aplicativo (online / batch).

Para otro tipo de bases de datos no-SQL y otros sistemas de almacenamiento (p. ej. almacenamiento distribuido), el modo de gestionarla y conseguir retrocompatibilidad sería distinto.. Para estos casos, los desarrolladores de los aplicativos junto con los desarrolladores de bases de datos:

- **Deben** adecuar los criterios y estrategias para asegurar la retrocompatibilidad en base al tipo de base de datos.
- En caso de modelos como mongo o Elastic donde el esquema no se actualiza sino que se sobrescribe, no aplica el mantenimiento de los script de modificación pero sí se **debe** tener el esquema como código en un repositorio para poder restaurar un esquema anterior.
- En los casos donde la base de datos no tenga esquema, se **debe** asegurar de la retrocompatibilidad se consigue coordinando el sistema que escribe y el que lee.
- En el caso de los modelos expuestos directamente para el consumo (y no a nivel API) no será posible realizar algunas de las estrategias planteadas. Por tanto, los equipos **deben** asegurar la coordinación con los diferentes equipos consumidores.

Indicadores

Los indicadores que apliquen, deben ser proporcionados a los equipos para conocer el grado de madurez respecto a la práctica:

Indicador	Descripción
-----------	-------------

Cambios manuales en bases de datos	Cambios en la base de datos mediante pipelines / Cambios totales de la base de datos (e.g. creación de tabla)
Cambios manuales en tablas	Cambios en una tabla mediante pipelines / Cambios totales de una tabla (e.g. creación de campo)
Cambios no retrocompatibles	Número de cambios que impiden retrocompatibilidad en bases de datos o tablas
Errores en aplicaciones debido a bbdd	Errores de aplicación consecuencia de base de datos / Errores totales (un ejemplo de esto puede ser un cambio no retrocompatible que no se ha avisado)

Herramientas (todas las prácticas)

- Servidor de automatización con el que se harán los pipelines (Jenkins)
- **Orquestador de releases:** Para pipelines complejos donde se necesita de cierta orquestación de pipelines y/o gates que habilitan pasar entre stages.
- Herramientas de software de infraestructura como código: define y crea infraestructura (por ejemplo, CloudFormation, [Terraform](#))
- **Herramientas de gestión de la configuración** que han evolucionado a manejar infraestructura (por ejemplo, Ansible o Puppet).
- Desarrollos ad-hoc que usan las APIs a través de librerías que proporciona la propia cloud (por ejemplo, usando Boto3 en AWS).
- Herramientas para la **gestión de bases de datos** (por ejemplo, [Flyway](#), [Liquidbase](#), Alembic).

Indicadores (todas las prácticas)

Los indicadores que apliquen, **deben** ser proporcionados a los equipos para conocer el grado de madurez respecto a la práctica:

Indicador	Descripción
Aplicativos sin pruebas automatizadas	Número de aplicativos desplegados sin pruebas automatizadas
Despliegues mediante pipelines	Número de despliegues desde pipelines / Número de despliegues totales
Frecuencia de releases	Diferencia intermensual de número de releases

Enlaces

- <https://martinfowler.com/bliki/DeploymentPipeline.html>
- <https://martinfowler.com/bliki/InfrastructureAsCode.html>
- <https://martinfowler.com/bliki/SnowflakeServer.html>
- <https://martinfowler.com/bliki/ImmutableServer.html>
- <https://martinfowler.com/bliki/PhoenixServer.html>
- <https://www.martinfowler.com/articles/evodb.html>
- <http://agiledata.org/essays/databaseRefactoring.html>
- <https://databaserefactoring.com/>

2.3. Seleccionar la estrategia de despliegue y aplicar mejores prácticas para eliminar indisponibilidad

La correcta selección y uso de una estrategia de despliegue sin indisponibilidad contribuye a mitigar el riesgo y permite una mayor frecuencia de despliegues.

Los equipos implicados en la puesta en producción de software **deben** seleccionar/diseñar una **estrategia de despliegue que no produzca indisponibilidad** al cliente o a los usuarios del aplicativo. Para las aplicaciones batch el concepto *zero downtime* es menos restrictivo. Para estos aplicativos solo se considera downtime cuando hay indisponibilidad en el momento en el que se debería estar ejecutando el aplicativo. Por tanto, en el caso de estos aplicativos, podrían existir ventanas de indisponibilidad siempre y cuando no se solapen con su ejecución. Esto implica que se pueden relajar para estas aplicaciones los criterios de indisponibilidad.

La estrategia de despliegue a utilizar **debería** ser discutida y acordada durante las fases previas al momento de despliegue para evitar errores en producción.

Beneficios

Seleccionar adecuadamente la estrategia de despliegue genera los siguientes beneficios:

- Disminuir el impacto de un posible error en releases
- Permite testear las funcionalidades y el despliegue de la manera apropiada según la estrategia escogida.
- Desplegar con cero indisponibilidad ya que los usuarios pueden moverse a la versión antigua en caso de que haya algún problema con el despliegue sin que deba haber indisponibilidad.
- Permite hacer rollbacks más rápidos y seguros
- Cumplir con regulaciones existentes que exigen utilizar una estrategia que mitiga el riesgo de los despliegues.

Precondiciones

- El sistema **debe** disponer de mecanismos para disponibilizar nuevas funcionalidades a un subconjunto de usuarios.
- El servicio/aplicación **debe** tener medidas de seguridad suficientes para asegurar que los despliegues se ajustan a la regulación existente.
- **Debe** haber una masa crítica de usuarios suficiente como para que tenga sentido seguir estas estrategias de despliegue
- Los desarrolladores **deben** disponer de sistemas que permitan conocer qué versión desplegada está utilizando un usuario.

Adopción

2.3.1. Elegir la estrategia de despliegue basada en coste, control de impacto y complejidad

La selección de la estrategia de despliegue apropiada se **debe** basar en las circunstancias del despliegue, coste, duración del rollback, impacto permitido en el usuario, complejidad del setup y rapidez del despliegue. Adicionalmente, las estrategias complejas (como las que se detallan a continuación) son pertinentes cuando los despliegues tienen cierta complejidad.

Se deben tener en cuenta dos tipos de patrones para realizar despliegues seguros:

- **Patrones de release basados en los entornos.** Para aplicar éstos es necesario contar con dos servicios, uno live y otro non-live en el que se despliega el código nuevo. La release se realiza transfiriendo tráfico. Por tanto, estos mecanismos tratan con la capa de infraestructura (por ejemplo network o balanceadores). Son más relevantes en despliegues de mayor magnitud. Las estrategias que implementan a este patrón y las circunstancias adecuadas para su uso son:
 1. **Blue-Green:** Esta estrategia requiere dos entornos (aunque uno de ellos puede ser temporal) una *green* que se usa para probar la nueva versión y otro, *blue*, que es el que contiene la versión anterior. Tras hacer pruebas y monitorizar el estado correcto de la nueva versión del aplicativo, se transferirá el tráfico en su totalidad del entorno blue al entorno green una vez se ha comprobado su correcto funcionamiento..
 2. **Canary:** Esta estrategia introduce la nueva versión del software a uno o más entornos de preproducción para un subset de usuarios de prueba. Una vez se ha testeado la funcionalidad con ese subset de usuarios se hace el roll-out al resto de usuarios
 3. **Rolling deployment:** Consiste en una evolución de la estrategia tradicional stop and run que, a diferencia de éste, evita la indisponibilidad.
- **Patrones de release basados en la aplicación.** Para aplicar éstos es necesario modificar levemente el aplicativo para que se pueda desplegar selectivamente una funcionalidad con pequeños cambios en la configuración. Por tanto, estos mecanismos se configuran y orquestan el propio aplicativo para mitigar el riesgo de los despliegues. Son más relevantes cuando se añaden funcionalidades de forma progresiva a la aplicación. Las estrategias relevantes de este patrón son:
 4. **Feature toggle:** Esta estrategia permite mediante ajustes en el aplicativo, servicio o componente ocultar cierta feature/s en desarrollo.
 5. **Dark Launch:** Consiste en implementar cambios en el aplicativo pero de modo que sean completamente invisibles para el usuario o cliente final.

Las estrategias que aquí se explican no son necesariamente excluyentes entre sí y se pueden combinar cuando sea oportuno. Los criterios para escoger una estrategia de despliegue son:

- **Coste:** Recursos económicos necesarios en términos de infraestructura.
- **Duración de rollback:** Tiempo que requiere realizar un rollback de la versión desplegada.

- **Impacto de error al usuario:** Capacidad de elegir los usuarios con los que se hacen tests de despliegue.
- **Complejidad del setup:** Implica una u otra estrategia en términos de código e infraestructura.
- **Rapidez del despliegue:** Facilidad para desplegar la nueva versión del aplicativo.

En base a las prioridades del equipo de desarrollo, la tecnología y las circunstancias se **debería** seleccionar una estrategia u otra en función de los criterios mencionados. Por ejemplo, en caso de que la reducción de coste sea una prioridad, el equipo de desarrollo y de despliegue debería seleccionar una estrategia de despliegue que minimizara el coste. La figura que se muestra a continuación ilustra las ventajas y desventajas de las distintas estrategias:


























Estrategia	Coste	Duración del Rollback	Impacto de error al usuario	Complejidad del setup	Rapidez del despliegue
Blue-Green					
Canary					
Rolling Deployment					
Feature Toggle					
Dark Launch					

Figura 2: Comparación propiedades de estrategias de despliegue

Indicadores

Los indicadores que apliquen, deben ser proporcionados a los equipos para conocer el grado de madurez respecto a la práctica:

Indicador	Descripción
Uptime	Porcentaje de disponibilidad del servicio (Up Time)
Despliegues Stop & Run	Despliegues Stop & Run / Despliegues totales
Despliegues Big Bang	Despliegues Big Bang / Despliegues totales

A continuación se detallan cada una de las estrategias mencionadas.

2.3.2. Despliegue Blue-green

Precondiciones

- Tener múltiples servidores o instancias donde está corriendo la aplicación
- Contar con un mecanismo para redireccionar el tráfico.
- Retrocompatibilidad entre versión actual y versión a desplegar

Adopción

Blue-Green es una estrategia de despliegue que consiste en transferir el tráfico de los usuarios de una instancia con la versión antigua de la aplicación (*blue*) a otra con la nueva versión (*green*). Es una de las estrategias más potentes ya que reduce el tiempo de indisponibilidad del aplicativo a cero durante el despliegue, aunque puede encarecer el coste de infraestructura de forma significativa.

Para utilizar esta técnica se deben aplicar las siguientes directrices:

- Se **debe** disponer de dos entornos productivos (blue-green) lo más parecidos posibles en términos de software y configuración. No tienen por qué ser permanentes en el tiempo, pueden ser dinámicos y crearse en el momento de hacer el rollout (blue) y destruirse la infraestructura que no está dando servicio (green).
- Se **debe** desplegar la nueva versión del código en uno de los entornos (green) y realizar todos los tests necesarios para asegurar que el software funciona como debería. Revisar el [playbook de testing](#) para más información acerca de los tests a realizar.
- Posteriormente se **debe** transferir el tráfico desde el actual entorno productivo (blue) al nuevo entorno (green). Se **debe** usar un load balancer u otro mecanismo para transferir el tráfico de un entorno a otro.
- Se **debe** monitorizar la estabilidad del nuevo entorno para ser capaz de rollback a la versión anterior en caso de que fuera necesario.
- Una de las complejidades con esta estrategia es la gestión de las dependencias que tenga el aplicativo. Particularmente, la interacción del aplicativo con APIs o base de datos puede ser compleja si es necesario cambiar el esquema cuando se realiza el nuevo despliegue. En estos casos, para poder realizar este tipo de despliegue, los cambios de la nueva versión **deben** aplicarse de modo que sean retrocompatibles. El equipo encargado de la base de datos **debe** seguir las técnicas mencionadas en [prácticas anteriores](#).
- Para conseguir retrocompatibilidad, los desarrolladores **pueden** combinar esta estrategia con feature toggles

Indicadores

Los indicadores que apliquen, deben ser proporcionados a los equipos para conocer el grado de madurez respecto a la práctica:

Indicador	Descripción
Tiempo de rollback	Tiempo desde que se detecta bug hasta que finaliza el rollback para toda la base de usuarios

2.3.3. Despliegue Canary

Precondiciones

- Tener múltiples servidores o instancias donde esté corriendo la aplicación
- La user-base **debe** ser de tamaño elevado.
- Contar con un mecanismo para redireccionar el tráfico.

Adopción

La estrategia [Canary](#) expone la nueva versión de la aplicación/servicio a un subconjunto de usuarios antes de hacer una release general a toda la base de usuarios. Una vez se ha probado, se libera la release para todos los usuarios.

Para utilizar esta técnica se **deben** seguir las siguientes directrices:

- Se **debe** seleccionar un subset de usuarios para el despliegue que se va a llevar a cabo. El grupo se puede seleccionar al azar o en base a la geografía, comportamiento de usuario, negocio u otras estadísticas relevantes que permitan validar el correcto funcionamiento de la nueva versión de software.
- El equipo de desarrollo **debe** definir el testing para comprobar que no hay impacto negativo de ningún tipo en términos de usabilidad o en términos de performance para el grupo de usuarios seleccionados. Esto **debe** incluir las métricas con las que se va a medir el impacto en el nuevo grupo y un dashboard para observar su evolución (más detalle en la sección de monitorización del playbook).
- Una vez lo anterior está definido, se **debe** disponibilizar la nueva versión al subconjunto de usuarios. Se distingue así a los usuarios con los que se va a probar la nueva versión del código del resto y se estudia a estos usuarios en detalle.
- Así como los despliegues blue-green, las dependencias deberían gestionarse de forma adecuada para dar soporte a ambas versiones. Por ejemplo, la base de datos debería cumplir la retrocompatibilidad. En caso de que eso no sea posible porque, por ejemplo el esquema cambiase, se **debería** crear un subset de datos apropiados que pueda dar servicio a la nueva versión del software que se pretende probar. Esto último añade complejidad en términos del rollback ya que si se ingestan datos con el nuevo esquema el equipo encargado de la bbdd y el aplicativo deben decidir qué transformaciones hacer para no perderlos.
- Si el resultado de la métrica es satisfactorio, se **puede** hacer el rollout a un grupo mayor de usuarios y repetir el proceso.

Indicadores

Los indicadores que apliquen, deben ser proporcionados a los equipos para conocer el grado de madurez respecto a la práctica:

Indicador	Descripción
Errores encontrados por usuario	Número de errores encontrados por usuarios= usuarios * error por usuario
Tiempo medio de despliegue gradual	Tiempo en minutos entre que se realizan pruebas con usuarios internos (friends and family) y que se libera a la totalidad de usuarios.

2.3.4. Rolling Deployment

Precondiciones

- Tener múltiples servidores donde está corriendo la aplicación

Adopción

Esta estrategia es una evolución de la estrategia de despliegue más tradicional que consistía en desplegar directamente en producción sin otro entorno en el que apoyarse (dejando de lado los tests previos llevados a cabo). Específicamente, el [Rolling deployment](#) va reemplazando el software de forma secuencial y controlada en los diferentes servidores, nodos o instancias que dan servicio. Aunque para la actualización deba haber un reinicio de máquinas o software, al no realizarse la actualización en todos los servidores simultáneamente siempre habrá servidores activos y no habrá indisponibilidad.

Para implementar esta estrategia de forma correcta los equipos de software deberían:

- El equipo encargado de la monitorización **debe** tener visibilidad de los health checks, alertas y errores que se pueden producir en cada uno de los nodos.
- El despliegue se realiza desplegando la nueva versión progresivamente. Conjuntamente, el equipo de desarrollo y el encargado del despliegue (que pueden ser el mismo) **deben** comprobar que no hay errores antes de desplegar el código en el siguiente nodo.
- El equipo de desarrollo **debería** disponer en todo momento de visibilidad sobre las métricas de todos los nodos durante el despliegue
- Los equipos **deberían** adoptar esta estrategia cuando no sea posible o no aplique utilizar las estrategias previamente mencionadas (Blue-Green o Canary). Algunas de las desventajas de esta estrategia con respecto a las anteriores son la mayor complejidad a la hora de orquestar el despliegue y su posible rollback o la mayor lentitud en la puesta en producción de la nueva versión para todos los usuarios. La otra desventaja importante es que la estrategia de Rolling deployment hace

compleja la tarea de controlar qué usuarios están con la nueva versión del código y cuales están en la anterior.

Indicadores

Los indicadores que apliquen, deben ser proporcionados a los equipos para conocer el grado de madurez respecto a la práctica:

Indicador	Descripción
Downtime acumulado	Downtime agregado de todos los servidores involucrados en el despliegue

2.3.5. Despliegue de features mediante Feature Toggle

Si es apropiado, los equipos de desarrollo **pueden** utilizar [Feature Toggles](#) (para más información visitar [documento anexo](#)) para desplegar su software. Esta técnica permite cambiar el comportamiento del sistema para los usuarios (o un subset de ellos) y por tanto probar nuevas funcionalidades. La principal desventaja de esta técnica es que introduce complejidad en el aplicativo. Es importante resaltar que este mecanismo permite activar una feature remotamente sin necesidad de re-desplegar el aplicativo.

En función de la longevidad (cuánto tiempo deben permanecer los toggles) y la sofisticación en la selección de usuarios, hay diversas categorías de Toggles:

- Toggles de **release**: deben ser temporales y generalmente aplican a todos los usuarios a la vez. Estos toggles son aquellos que se utilizan para desplegar código y que pueda ser testeado en producción con un riesgo controlado. Por tanto, este tipo de toggles se utilizan con el objetivo de que al desplegar una nueva feature en producción, se pueda desactivar en caso de que haya una reducción importante en la performance.
- Toggles **internos** ([Champagne Brunch](#)): son aquellos toggles temporales que incorporan una cierta lógica de filtrado de usuarios. Concretamente, apuntan a un set de usuarios internos de la empresa y con ellos prueban las nuevas funcionalidades. Esto permite mitigar el impacto de errores al liberar software nuevo.
- Toggles **experimentales** (A/B testing): son aquellos que deben permanecer el tiempo suficiente en la aplicación para producir resultados estadísticamente significativos. Generalmente, esto suele ser entre unas horas y unas semanas. Más allá de eso los resultados serán inválidos porque cualquier otro cambio en el sistema influirá en el resultado. De la misma manera que los toggles de release son temporales pero la sofisticación en la selección de los usuarios es mayor.
- Toggles de **operaciones**: En algunos casos especiales con features que consumen muchos recursos (p. ej panel de recomendación que se muestra en la homepage) o son críticas en términos de seguridad (por ejemplo, features afectadas por el fraude),

estos toggles pueden mantenerse para tener una mecanismo rápido que desactiva estas features. Aunque pueden ser temporales, habitualmente se mantienen en el tiempo. Además, generalmente, se aplican a todos los usuarios al mismo tiempo y por tanto no hay selección de usuarios.

- Toggles de **permisos**: son aquellos que se utilizan para cambiar la experiencia de usuario y diferenciar entre usuarios. Por ejemplo, estas toggles permiten habilitar determinadas feature para los usuarios de pago en contraposición con el resto de usuarios. Pueden ser permanentes y la lógica de selección de usuarios puede ser compleja. Éstos corresponden a la capa / componente correspondiente a la autorización o autenticación.

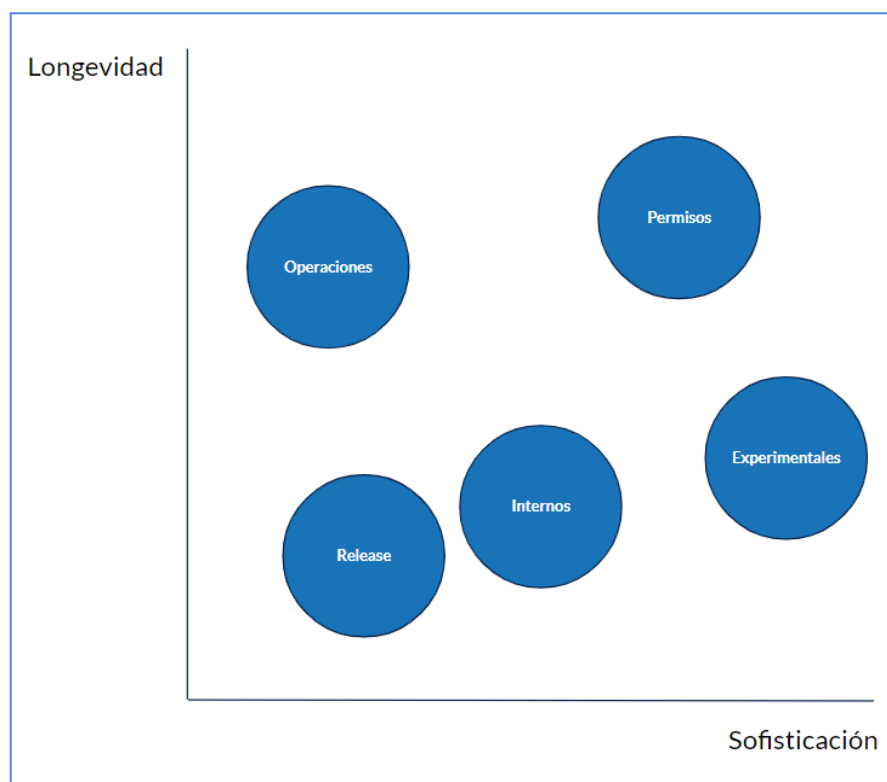


Figura 3: Tipos de feature toggle

Para despliegues y releases los más relevantes son los toggles de release, internos y experimentales que son temporales. Estos toggles que se utilizan para llevar a cabo el despliegue deberían utilizarse teniendo en cuenta que:

- La activación o desactivación se **debería** realizar utilizando una de estas técnicas (de mayor sofisticación a menor):
 - Un fichero de configuración
 - Variables de entorno
 - Utilizar bases de datos que determinen el estado del toggle y que permiten llevar un histórico de activaciones y desactivaciones.
 - Utilizando una IU o un servicio web que controle las feature toggles.
 - Usando SaaS que pueden proveer de este servicio (por ejemplo *LaunchDarkly*, *Split*).

- La implementación de los feature toggles permanentes **debe** ser más compleja que de los temporales ya que al permanecer en el aplicativo, pueden empeorar la calidad y rendimiento del aplicativo.
- Los Feature Toggles **deben** tener un comportamiento por defecto el cual aplicaría a todos los usuarios.
- El equipo que desarrolla el aplicativo, **debe** adoptar naming conventions adecuadas para poder reconocer los feature toggles y de este modo no activarlos o desactivarlos erróneamente.
- El equipo de desarrollo **debe** evitar comportamientos contradictorios de los toggles / flags ya que pueden causar, entre otros problemas, fallos en los tests de integración y regresión.
- Los equipos que han introducido los Feature Toggles asociados a despliegues de features **deben** ser los encargados de eliminarlos cuando ya no sean necesarios para evitar deuda técnica (Toggle Debt).
- Los feature toggles permanentes, **pueden** permanecer en el código indefinidamente.
- El equipo de desarrollo y/o gobierno del aplicativo **debe** establecer un control granular de quién tiene la capacidad de activar o desactivar los toggles.
- Los desarrolladores **deberían** evitar que las mismas feature toggles / flags controlen múltiples partes del aplicativo (por ejemplo, que el flag *mostrar sidebar* aparezca en múltiples partes del aplicativo con funciones diferentes).
- Los desarrolladores **deberían** tener acceso a la activación/desactivación de los toggles.
- Para las feature toggles más complejas, especialmente las de tipo experimentales, se **puede** utilizar un proveedor externo.
- Cuando el despliegue de una funcionalidad depende de otro componente y ese componente aún no está listo para ser liberado, el equipo de desarrollo de esa funcionalidad **puede** utilizar las feature toggles para liberarla. Este uso permite desacoplar despliegues ya que, al desplegar esa feature en modo desactivado, no haya errores por la ausencia de la dependencia. Para estos casos, es clave que cuando se libera el componente del que depende la funcionalidad, se active la feature toggle siguiendo los procedimientos de seguridad apropiados.

Indicadores

Los indicadores que apliquen, deben ser proporcionados a los equipos para conocer el grado de madurez respecto a la práctica:

Indicador	Descripción
Tiempo promedio de permanencia de toggles temporales	Número de días de promedio de permanencia de toggle features de tipo temporal (de tipo release o internos)

2.3.6. Despliegue de features mediante Dark Launches

Los desarrolladores junto con el equipo de operaciones **pueden** aplicar la estrategia **Dark Launch** (o subida técnica) para limitar el número de cambios simultáneos o evaluar el impacto en el rendimiento. Hay dos tipos principales dependiendo de si recibe o no peticiones el código desplegado:

- **Recibe peticiones:** El código desplegado recibe peticiones pero es invisible para el usuario. Un ejemplo de este tipo es la modificación de una funcionalidad que recomienda productos en base a la preferencia de los usuarios. Si esta modificación puede afectar negativamente al rendimiento, esta estrategia se aplicaría de modo que ambas versiones del algoritmo están corriendo en paralelo pero mostrando al usuario solamente los resultados de la versión anterior.
- **No recibe peticiones / no se ejecuta:** Un ejemplo habitual de este tipo es aquel despliegue en el cual se libera una parte del código global sin que se ejecute esa parte para así realizar pruebas de rendimiento.

Para poder realizar este tipo de estrategia, los desarrolladores deben adherirse a las siguientes buenas prácticas:

- Los cambios implementados **deben** estar en producción pero los usuarios no deberían tener acceso a la funcionalidad.
- Los desarrolladores **deben** monitorizar el impacto en el rendimiento y el correcto funcionamiento del código en producción una vez se realiza el dark launch. Esto permite controlar si hay anomalías (situaciones no deseadas).
- El código no **debería** permanecer más tiempo del estrictamente necesario.
- Se **puede** combinar con otras estrategias mencionadas.

Indicadores

Los indicadores que apliquen, deben ser proporcionados a los equipos para conocer el grado de madurez respecto a la práctica:

Indicador	Descripción
Subidas dark launch con éxito	Número de despliegues dark launch sin impacto en la performance del servicio

Herramientas (todas las prácticas)

- Herramienta para transferir el tráfico (por ejemplo, Load balancer/ Caudalímetro): sistema para controlar la transferencia de tráfico.

Indicadores (en revisión)

Los indicadores que apliquen, **deben** ser proporcionados a los equipos para conocer el grado de madurez respecto a la práctica:

Indicador	Descripción
Tiempo promedio de respuesta	Diferencia de tiempo promedio de respuesta de los servicios desplegados entre el despliegue actual y el anterior
MTBF (Mean time between failures)	Tiempo medio entre errores en producción
Tiempo medio de recuperación	Tiempo medio de recuperación cuando se produce un incidente. Mide desde el momento en el que se detecta el bug hasta que la totalidad de los usuarios dejan de experimentar errores
Impacto al Usuario	Variación en la valoración del usuario durante el despliegue (actual vs anterior)
Coste	Coste de los despliegues (coste de entornos o infraestructura + coste de tiempo)

Enlaces

- <https://www.cloudbees.com/blog/rolling-deployment>
- <https://martinfowler.com/bliki/CanaryRelease.html>
- <https://martinfowler.com/bliki/BlueGreenDeployment.html>
- <https://martinfowler.com/bliki/DarkLaunching.html>
- <https://martinfowler.com/articles/feature-toggles.html>

2.4. Mantener completa observabilidad antes, durante y después del despliegue

El equipo de desarrollo debe tener observabilidad completa y por tanto:

1. Asegurar trazabilidad completa de la release de forma automática.
2. Comunicar el estado y resultado del despliegue a terceros.
3. Establecer la monitorización de eventos y alertas.
4. Realizar el seguimiento y reaccionar a eventos y alertas.
5. Aprender y mejorar de despliegues pasados.

Beneficios

- Conocer el estado del sistema de forma precisa en cualquier momento. Lo cual es especialmente importante durante un despliegue.
- Diferenciar los posibles problemas derivados del propio despliegue de aquellos producidos por indisponibilidades de elementos terceros coincidentes con el proceso de despliegue.
- Permitir la actuación de equipos dependientes en caso de que haya algún incidente durante el proceso.
- Mejorar la fiabilidad del sistema en su conjunto.

Adopción

2.4.1. Asegurar trazabilidad completa de la release de forma automática

Beneficios

- Mejorar calidad, ya que los despliegues están estructurados, de forma que el testing, la identificación y corrección de errores, y la verificación de otros factores relativos a la calidad del software que se despliega, forman parte de una fase core del proceso.
- Conseguir mayor confiabilidad y estabilidad del sistema, ya que se minimizan los cambios no planificados o inestables en el entorno de producción.
- Incrementar la visibilidad y control, ya que se requiere una planificación y coordinación de los despliegues y de todos los equipos implicados.
- Permitir la visión *end to end* de los despliegues y la capacidad de hacer seguimiento de las versiones y los cambios desplegados.
- Mejorar eficiencia, ya que habilita minimizar los tiempos de inactividad y de reworking
- Identificar el contenido funcional y técnico que se despliega en cada release.
- Reducir de la documentación y controles requeridos, dado que al ser un único proceso global y tener toda la información trazada.

Precondiciones

- **Debe** estar definido el proceso de release y detalladas sus fases.
- El proceso **debe** estar implementado en las herramientas del ciclo de vida del desarrollo e integrado y con las automatizaciones y comunicaciones que garanticen la agilidad del mismo.

Adopción

Se **debe** habilitar un mecanismo que permita conocer el **estado de una release**. Se **debe** conocer cuándo comienza y termina cada fase de la release que está en curso (o la próxima) y el estado y detalle de la misma, asociado de forma unívoca con cada cambio que forma parte del futuro despliegue.

Los equipos de desarrollo **deben** poder conocer el estado de una release únicamente consultando el estado del pipeline. Además, **deben** poder consultarlo en un sitio público y conocido por todos los ingenieros y stakeholders.

Además, **debe** existir una **herramienta** única y global para dar trazabilidad de forma automática a todos los componentes que forman parte de un despliegue a producción. Para conseguir que exista trazabilidad y que la **herramienta** sea completa:

- **Debe** censar el propósito/funcionalidad que se quiere satisfacer con cada elemento que se despliega en producción (trazabilidad entre la funcionalidad y el código desplegado)
- **Debe** relacionar el ejecutor de cada **despliegue** en producción, con el **componente** desplegado.
- **Debe** relacionar con la release **las tareas necesarias**, adicionales a las propias del despliegue a través del pipeline (e.g., tareas de operación, validaciones adicionales).
- **Debe** identificar de forma unívoca al **responsable del cambio** y también a los equipos impactados.
- **Debe** haber integridad y relación entre las herramientas que componen el ciclo de vida.
- Se **puede** relacionar información de interés con los cambios desplegados (e.g., MSA, TDM).

Indicadores

Indicador	Descripción
Elementos desplegados sin trazabilidad	Número de elementos desplegados sin las relaciones indicadas en la práctica

2.4.2. Comunicar el estado y resultado del despliegue a terceros

Precondiciones

- Los receptores de las comunicaciones **deben** ser conocidos y explícitamente definidos dentro del proceso de release.

Adopción

El equipo de desarrollo encargado del despliegue debe llevar a cabo **dos tipos de comunicación** con otros interlocutores en relación al nivel de afectación del cambio:

- **Los equipos técnicos directamente impactados** (si el cambio impacta otros componentes), **deben** ser informados y/o participar activamente en la implantación y pruebas del cambio en caso de que éstos consideren relevante participar.
- **Los usuarios o clientes finales**, que **deben** ser informados de los cambios.

En el caso de **equipos técnicos directamente impactados** las directrices a aplicar son las siguientes:

- Los equipos de desarrollo **deben** integrar a los equipos de Operaciones en las dinámicas/ceremonias de los equipos de Desarrollo, en los casos en los que éstos sean equipos independientes.
- Los participantes y responsables del despliegue (tanto Desarrollo como Operaciones) **deben** identificarse previamente para tener un contacto claro en caso de detección de errores en el despliegue.
- El equipo de desarrollo **debe** y es responsable de informar sobre el despliegue planificado y facilitar toda la información relevante para el resto de equipos de desarrollo.
- Los participantes encargados del despliegue **deben** asegurarse de tener un canal de comunicación rápido con los equipos de los componentes o servicios dependientes para agilizar la resolución de dudas e incidentes durante el despliegue. El proceso debe ser automático siempre y cuando esto sea posible
- Se debería crear un [Changelog note](#) que se **debería** consumir de forma interna (se dirige a otros equipos técnicos). Ese changelog **debería** generarse automáticamente a través del propio pipeline de CI/CD como se explica en el playbook de [code development and versioning](#).

En el caso de la comunicación con **usuarios o clientes finales**, es importante tener en cuenta el versionado y la comunicación mediante una **release note**.

- El versionado **debe** ser ordenable y claramente identificable cual es la versión anterior y cuál es la versión posterior.
- Los desarrolladores del servicio y sus consumidores **deben** coordinarse de modo que cambios en el código desplegados en producción se vean reflejados (de acuerdo al versionado indicado) en una nueva versión.
- Para los equipos internos que consumen un servicio o utilizan un componente, **debe** existir un límite temporal de mutuo acuerdo para que los consumidores aumenten la versión del servicio con el objetivo de evitar que estén activas versiones muy antiguas del servicio o componente.
- La comunicación de una release para los usuarios y equipos dependientes del servicio **debe** destacar las nuevas funcionalidades o las que han sido deprecadas. Esto permite a los clientes y stakeholders identificar qué cosas se han agregado, cambiado o eliminado.
- En las comunicaciones se **deben** destacar aquellos cambios no retrocompatibles en las infrecuentes ocasiones en las que los haya. Se **deberían** poner ejemplos con los cambios de código requeridos para la nueva versión.

- **Release Note**: describe los cambios y mejoras que se han realizado en una versión de software específica, y se utiliza para informar a los usuarios finales sobre las actualizaciones. La release note **debería** escribirla el desarrollador, el product owner o el equipo de marketing según aplique. La release note **debe** centrarse en la experiencia de usuario y ser consumida por el propio usuario evitando en todo momento el lenguaje técnico.
- La release note **debe** incluir los siguientes elementos:
 - Título: El título debe ser descriptivo y reflejar el contenido de la versión.
 - Fecha: La fecha de la versión debe ser claramente visible.
 - Resumen: Un breve resumen de los cambios y mejoras realizados en la versión junto con las incompatibilidades respecto a la versión anterior.
 - Mejoras: Una lista detallada de las mejoras y cambios realizados en la versión.
 - Correcciones: Una lista detallada de los errores corregidos en la versión.
 - Instrucciones de instalación (cuando aplique): Instrucciones detalladas sobre cómo instalar la nueva versión del software.
 - Instrucciones de uso: Instrucciones detalladas (guía operativa de fácil entendimiento) que explica cómo utilizar las nuevas funcionalidades y mejoras.
 - Contacto: Lista de emails del equipo o de los responsables del componente a desplegar.

Indicadores

Indicador	Descripción
Porcentaje de repositorios con la documentación requerida	Número de despliegues que contienen changelog
Release Note	Porcentaje de los releases que contienen una release Note con los componentes mínimos mencionados

2.4.3. Establecer la monitorización de eventos y alertas

Precondiciones

- Los roles y responsabilidades en el monitoreo de todos los miembros del equipo de desarrollo y de otros equipos **deben** estar definidos y estar actualizados para asegurar la recepción, manejo de alertas y notificaciones relevantes.

Adopción

El equipo de desarrollo **debe** estar involucrado en el monitoreo de los sistemas, especialmente durante el despliegue, es lo que permite tener un control del estado del sistema y, más específicamente, de un despliegue (incluyendo como parte del mismo despliegue el proceso de preparación anterior y observabilidad posterior al despliegue). El

establecimiento de eventos y [alertas](#) (i.e. notificaciones que pueden ser leídas por un humano) ayudando a **identificar incidentes y tomar acciones correctivas**.

Los equipos de software **deben** establecer o estar involucrados en el establecimiento de las alertas y las métricas que miden el estado del despliegue. El sistema de alertas **debe** seguir las siguientes directrices:

- Los equipos **deben** establecer y disponer de un **sistema de monitorización continua** que les permita identificar incidentes en sus aplicativos en todo momento
- El sistema de monitorización **debe** monitorizar **métricas** útiles para reconocer incidentes que se puedan producir durante el funcionamiento de software. Las métricas **deben** ser acordadas entre el equipo de desarrollo, el equipo de operaciones y el equipo encargado de gestionar las incidencias (si estos equipos no son los mismos).
- El **sistema de monitorización debe** analizar el estado del software y su despliegue realizando health checks de los mismos y por tanto:
 - **Debe** medir múltiples **métricas o señales** para conocer el estado del servicio en torno a un despliegue. Señales importantes que se deben medir son las relacionadas con la **latencia** (tiempo de respuesta), el **tráfico** (e.g. HTTP requests por segundo), los **errores** (e.g. HTTP 500) y la **saturación** (e.g. desbordamiento del sistema).
- El equipo de operaciones **debe** realizar, mediante las métricas mencionadas y otras que puedan ser relevante, múltiples tipos de monitorización continua sobre:
 - El estado de la infraestructura ([infrastructure monitoring](#)).
 - El rendimiento de la aplicación ([application performance monitoring](#)).
 - Logs que genera la aplicación ([log management monitoring](#)).
 - Seguridad ([security monitoring](#)). para poder ser contrastado antes, durante y después del proceso de despliegue.
- El equipo **debe** disponer de un sistema de modificación o un mecanismo de navegar y profundizar temporalmente al nivel de log de manera rápida y sencilla.
 - El equipo de desarrollo **debe** establecer o colaborar en el establecimiento de un **nivel de logging** necesario dependiendo de las situaciones, incluyendo: **Error, Warn e Info** (el nivel Debug se debería restringir para la etapa de desarrollo). Tener la especificación de la severidad de un evento de logging facilita la lectura y permite un filtrado más eficiente.
 - Los logs **deberían** contener como mínimo timestamp, nivel, mensaje y nombre del logger que originó el mensaje
 - Los logs **pueden** estructurarse en JSON o formatos similares para facilitar su navegabilidad.
- Las alertas **deberían** ser *forward-looking*. Es decir, que además de dar una visión de la situación actual también puedan dar visión de la tendencia y la posible situación futura. Esto se puede llevar a cabo con algoritmos de predictibilidad que puedan alertar de una posible caída del servicio.

- El equipo encargado de establecer las alertas **puede** ser el equipo responsable del servicio
- Se **debe** configurar las alertas de forma adecuada, optimizando la ratio señal-ruido.
- Service owner **puede** disponer de alertas específicas para despliegues críticos como se especifica en el [playbook correspondiente](#).
- Los sistemas usados para observar el funcionamiento de los servicios (logs, alertas, métricas, health-checks sintéticos, etc.) **deben** permitir identificar fácilmente el componente/servicio origen de cualquier indisponibilidad, el detalle del error y a los responsables de los servicios.
- El equipo de desarrollo **puede** establecer con las áreas de operaciones, procesos de [monitoreo sintético](#).

Indicadores

Indicador	Descripción
Despliegues con observabilidad	Despliegues con miembro de equipo de desarrollo con acceso a logs / Total despliegues
Tiempo para recibir permisos de logs	Tiempo que ha transcurrido entre que se ha efectuado la petición de los permisos para acceder a los logs del despliegue hasta que se han recibido

2.4.4. Realizar el seguimiento y reaccionar a eventos y alertas

El equipo de desarrollo encargado del despliegue y el responsable del servicio/componentes a desplegar **debe** tener un sistema de monitoreo ampliamente disponible:

- El equipo de desarrollo **debe** tener acceso a las alertas y logs del aplicativo relacionados con el despliegue.
- El acceso puede ser temporal y **debe** existir un registro de los permisos que se otorgan a los desarrolladores.
- **Debe** existir una sola fuente donde consultar las métricas principales para conocer el estado del sistema y del despliegue. Las métricas deben estar disponibles a tiempo real.
- Los desarrolladores **deben** realizar seguimiento y alertas sobre toda funcionalidad validada y aceptada previamente por los responsables.
- Los sistemas usados para "observar" el funcionamiento de los servicios (logs, alertas, métricas, health-checks sintéticos, etc.) **deben** permitir exportar la información necesaria para que sea explotable y visualizable fácilmente por los responsables de los servicios y de los despliegues (si existieran actuaciones centralizadas en equipos técnicos) sin tener acceso físico necesariamente a los sistemas productivos que los soportan.

- Para los escenarios en los que no sea viable el análisis del origen de la situación "observada", **debe** existir un protocolo y procedimiento ágil para facilitar acceso temporal a las personas adecuadas a los sistemas implicados para realizar el troubleshooting (resolución de problemas) detallado.
- Los equipos de soporte **deben** disponer de acceso permanente a los eventos, alertas y logs del aplicativo.
- Los equipos de soporte **deben** comunicar por el medio que sea relevante para cada servicio los errores y alertas a los equipos encargados del despliegue y/o a los equipos encargados de solucionar los eventos destacables como alertas cuando estas se produzcan.
- Los logs **deben** tener cifrados los datos sensibles de clientes y del aplicativo en sí.
- La monitorización **debería** incluir la trazabilidad de la utilización y funcionamiento de las funcionalidades entregadas. Esto permitirá comprobar la utilidad de la funcionalidad.

2.4.5. Aprender y mejorar de despliegues pasados

Una de las características de desplegar regularmente es la capacidad de aprender progresivamente para cometer menos errores en un proceso inherentemente complejo.

- Los desarrolladores **deben** documentar los procesos de despliegue para que otros miembros del equipo puedan realizar los despliegues.
- Los desarrolladores **pueden** sugerir mejoras en el pipeline de despliegue que mejore alguna de las prácticas mencionadas más adelante.
- Cuando se producen incidentes, los equipos de software del Banco **deben** documentar de forma transparente lo ocurrido creando un **blameless postmortem** (detalle describiendo el motivo del incidente sin detallar los culpables de forma explícita). Esto tiene el objetivo de **asegurar que los incidentes no se vuelvan a producir** comprendiendo las causas principales por las cuales se produjo.
- La intención del blameless postmortem no **debe** señalar a la/s persona/s que cometieron un error sino qué falló en el proceso para que se produjera el error. Para llevar a cabo un **postmortem** deben seguirse las directrices siguientes:
 - El postmortem **debe** especificar los equipos responsables de la acción correctora.
 - No solo el que se encargó de resolver el incidente debería encargarse del documento sino que todo el/los equipo/s involucrados **deben** encargarse de la forma lo más colaborativa posible.
 - El reporte **debe** recoger especialmente errores debidos a falta de automatización de procesos y se debe proponer un modo de automatizarlo.
 - El reporte se **debe** compartir en un site público al que los equipos relevantes de la organización puedan tener acceso.
 - El plan para evitar un nuevo incidente **debe** constar en el reporte y se debe involucrar a todos los equipos relevantes para lograrlo.
 - Los documentos post-mortem **deben** incluir como mínimo:

- Explicación del Error
 - Causa del error
 - Status
 - Pérdida de datos de cualquier tipo
 - Resolución
 - Tiempo de resolución
 - Detección / Fallos en la monitorización
- Actuaciones para remediar el error y *Timeline*:
 - Intervención del ingeniero
 - Intervención de los equipos de soporte
 - Intervención de operaciones
- Recomendaciones, lecciones aprendidas y conclusiones

Indicadores

Indicador	Descripción
Porcentaje de incidentes sin post-mortem	De todos los incidentes reportados en cada uno de los equipos, qué porcentaje no tienen post-mortem
Incidentes con causa desconocida	Número postmortem sin que se haya detectado la causa raíz del incidente
Incidentes sin acción correctora propuesta	Número postmortem sin que los responsables hayan propuesto una solución
% de acciones implementadas	Acciones implementadas / Acciones propuestas (en los postmortem)

Herramientas (todas las prácticas)

- Herramienta que permita informar y desplegar un proceso global de releases (por ejemplo, JIRA o SMART⁴).
- Herramienta que recoja toda la documentación no técnica del proyecto y esté enlazada a la release (por ejemplo, Confluence).
- Herramientas de ingesta de logs, parametrización de métricas y alertas ligadas a los Servicios (por ejemplo, Semaas Suite, Splunk, Datadog, PagerDuty, [ELK Stack](#) y/o Opsgenie).
- Plataforma de Visualización de indisponibilidades planificadas y no planificadas de Servicios y componentes.

⁴ Solución propia desarrollada en Turquía

- Dashboard que permita hacer el seguimiento de los despliegues y visualizar los health checks del sistema (por ejemplo, changestimeline⁵, [dynatrace](#) o [instana](#)).
- Aplicaciones de mensajería instantánea para el envío de alertas por medio de API (por ejemplo, Slack, Microsoft Teams o Google Webhooks).
- Herramientas de monitorización de estrategias complejas de despliegue (por ejemplo, SMART⁶).
- Site público que recoja los postmortem.

Indicadores (en revisión)

Los indicadores que apliquen, **deben** ser proporcionados a los equipos para conocer el grado de madurez respecto a la práctica:

Indicador	Descripción
Tasa de detección activa	Errores detectados por equipo despliegue / Errores totales detectados Errores totales detectados = Errores detectados por equipos de monitorización + Errores detectados por equipo despliegue
MTTD (Mean time to detection)	Tiempo de detección= Tiempo en minutos entre el despliegue que contiene bugs y el momento que se detecta el error

Enlaces

- <https://geekflare.com/release-management-guide/>
- <https://www.simplilearn.com/release-management-process-article>
- <https://es.wikipedia.org/wiki/Changelog>
- <https://github.com/kubernetes/kubernetes/blob/master/CHANGELOG/CHANGELOG-1.18.md>
- https://es.wikipedia.org/wiki/Notas_de_publicaci%C3%B3n
- <https://sre.google/sre-book/monitoring-distributed-systems/>
- <https://martinfowler.com/articles/qa-in-production.html>
- <https://martinfowler.com/bliki/SyntheticMonitoring.html>
- <https://www.dynatrace.com/news/blog/what-is-infrastructure-monitoring-2/>
- https://csrc.nist.gov/glossary/term/information_security_continuous_monitoring
- <https://sre.google/sre-book/postmortem-culture/>

⁵ Herramienta inhouse desarrollada en Turquía que permite observar el estados de los despliegues (<https://devjobs/changestimeline>)

⁶ En la misma solución SMART, se puede observar el estado de servidores en despliegues Canary o Blue-Green

3. Proceso

Las prácticas revisadas anteriormente se operativizan a través del **proceso de gestión y despliegue de releases**. Se define el proceso de gestión y despliegue de releases como una parte integral del ciclo de vida del software que se encarga de planificar, coordinar y controlar la liberación de nuevas versiones de software o actualizaciones, asegurando que se realicen con éxito y minimizando los riesgos e impactos negativos en los sistemas existentes.



Figura 4: Proceso de gestión y despliegue de releases

El proceso de release que se trata a continuación se circunscribe al despliegue de software en producción y entrega de una nueva release. Consta de fases que pueden variar en función de la naturaleza de los proyectos y los cambios que se quieren desplegar, pero a continuación se indican algunas fases que deberían seguirse:

1. Planificación

Implica la definición de los objetivos y requisitos de la release, la asignación de recursos, la identificación de riesgos y elaboración de un plan detallado de las actividades necesarias para poder llevar a cabo la implementación de los cambios. No tiene relación con la planificación del proyecto. En función de lo complejo del cambio (volumen de desarrollo, cantidad de equipos involucrados, etc.) puede variar la forma en la que se materializa (cambios, que únicamente aplica a un equipo y que trabaja con Continuous Delivery pueden sustentarse en "acuerdos de equipo", mientras que cambios en productos con múltiples tecnologías y equipos puede necesitar de herramientas externas, comités, etc.).

El **equipo de desarrollo** es el responsable de iniciar esta fase y debe de disponer autonomía para ello. Para sistemas complejos (compuesto por múltiples componentes que necesitan sincronización) el **release manager** puede dar soporte llevando a cabo distintas actividades

(e.g., generar la release note para el usuario final). El **Service Owner** debe estar informado del inicio del proceso de release, para más información leer el [playbook de service owner](#).

2. Generación release

El **equipo de desarrollo** define los componentes a empaquetar y se genera la release del aplicativo haciendo uso de pipelines automatizados e idempotentes.

3. Pruebas

Antes del despliegue en entorno productivo es necesario que el equipo de desarrollo y los perfiles relacionados con el testing prueben, y evidencien el correcto funcionamiento de los incrementos de código (mediante pruebas de regresión u otro tipo de pruebas), acorde a los criterios que se establezcan por parte del equipo y de los stakeholders interesados (Service Owner, Product Owner, Seguridad, Certificación de rendimiento, etc.). Esta fase se detalla en profundidad en el [playbook de testing](#).

4. Aprobación

Una vez superada la fase de pruebas, es necesario que se cumplan las revisiones y aprobaciones del resto de **stakeholders** del proceso de despliegue (e.g., seguridad, cumplimiento regulatorio).

El **equipo de desarrollo y los responsables necesarios** en función de los cambios (e.g., de seguridad en proyecto, cuando no estén integrados en el equipo) deben dar dicha aprobación y opcionalmente, en caso de sistemas complejos, el **release manager** velará por la coherencia de la release cuando haya varios componentes implicados (e.g., varios equipos que tienen dependencias). Además, el **product owner** debe dar soporte en la aprobación de la release validando el incremento funcional cuando sea necesario

5. Implementación

En esta fase los cambios son implementados en el entorno productivo por parte del **equipo de desarrollo**. Además debe validarse que el despliegue ha sido correcto (pruebas, revisión trazas, etc.), en función de la naturaleza del cambio y las opciones existentes (e.g., habrá sistemas que no permitan validar de forma interactiva).

6. Validación

Después de la implementación, el **equipo de desarrollo** debe realizar un seguimiento para monitorizar que los cambios desplegados no generan un efecto adverso en producción y que cumplen el propósito y los requerimientos definidos en la fase de planificación. En algunos casos donde los despliegues implican múltiples sistemas y equipos el equipo puede apoyarse en la figura del **release manager**.

7. Cierre

Una vez se ha verificado el cumplimiento de los objetivos en la fase de validación, se procede con la fase de cierre de la release. En esta fase el **equipo de desarrollo** junto con el **product owner, el release manager y el service owner** deben dar por finalizado el despliegue y documentar los resultados del despliegue y hacer revisión del resultado del despliegue de la release en con el objetivo de buscar puntos de mejora.

Las tareas de operación de la infraestructura estarán integradas en el **equipo de desarrollo**, para romper el modelo que existe en muchos sistemas de “caja negra”. Cuando los equipos de desarrollo no tengan la capacidad de operar una infraestructura (modelos on-premise principalmente) se debe integrar en sus equipos perfiles con las capacidades y atribuciones necesarias para realizar las tareas de operación relacionadas con el despliegue y observabilidad. Es decir, avanzar hacia un **modelo DevOps**, que facilita la adopción de las prácticas descritas en este playbook y la eficiencia de los ciclos de Release.

Durante el proceso descrito se pueden dar situaciones no deseadas (errores o bugs encontrados en alguna de las etapas) lo que implica que se debe hacer rollback y dejar el entorno tal y como se encontraba inicialmente. En el caso de despliegues críticos que requieran cierta rapidez (**despliegue de hotfixes**) se llevarán los mismos pasos descritos aquí, siempre que el tiempo dedicado a cada fase se reduzca (e.g., la validación debe ser rápida y todas las partes involucradas deben hacer el esfuerzo de completarla lo antes posible) y consiguiendo así **desplegar de forma ágil**.

De forma transversal en todos los pasos descritos anteriormente nos apoyaremos en la función de [Release Manager](#), la cual para equipos pequeños y/o autónomos, podrá ser desempeñada por uno de los integrantes técnicos del propio equipo, y para casos en los que en la Release participan muchos equipos distintos, se recomienda que dicha función sea realizada por alguien ajeno al equipo con una visión cross del Servicio/Producto, y con una intensa coordinación con el equipo DevOps (si aplica) y con los Service Owner y/o App Owner del servicio (dichos roles están definidos en el playbook del Service Owner).

Alineado con la [práctica 2.4](#), durante todo el proceso de release se tiene que mantener una completa **observabilidad** (i.e., monitorización) de las acciones y sus resultados. Facilitando información acerca del estado de la release de forma accesible para los responsables del despliegue (equipo de desarrollo, release manager y service owner).

GLOSARIO

Equipo: Conjunto de personas trabajando sobre un mismo backlog de tareas asociado a un producto de software.

Servicio: Productos de software (e.g., aplicación web, aplicación móvil, librerías de transacciones).

Despliegue: Instalación de una versión de software específica en un entorno determinado de la infraestructura. En este playbook el despliegue hace referencia siempre al entorno de producción.

Pipeline de despliegue: manifestación automatizada del proceso que lleva el software desde un repositorio o sistema de control de versiones a las manos de los usuarios⁷.

Release / Liberación: Consiste en hacer una feature o conjunto de features disponibles para los usuarios o una grupo significativo de ellos.

Rollback: Reversión de un sistema y la modificación de sus dependencias a su estado anterior al despliegue.

Hotfix: Adición de código desarrollada para corregir un bug o error producido en una release.

⁷ Farley D., Humble Jez (2010). Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley.

REFERENCIAS

- Farley D., Humble Jez (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley.
- Nygard, Michael T. (2017). *Release It!: Design and Deploy Production-Ready Software*. Pragmatic Bookshelf
- Sander Rossel, (2017) *Continuous Integration, Delivery, and Deployment: Reliable and faster software releases with automating builds, tests, and deployment*. Packt Publishing.
- Kim, Gene & Debois, Patrick & Humble, Jez & Allspaw, John (2016). *The devOPS Handbook: How to Create World-Class Agility, Reliability and security in Technology Organizations*. O'Reilly