

# Continuous Integration and Continuous Deployment

## Problemática y Definición Formal

El objetivo de la CI/CD, que significa integración y distribución o implementación continuas, es optimizar y agilizar el ciclo de vida del desarrollo del software.

La integración continua (CI) consiste en integrar los cambios del código en un repositorio de código fuente compartido de forma automática y frecuente. La implementación o distribución continua (CD) es un proceso de dos partes que implica la integración, la prueba y la distribución de los cambios en el código. Mientras que en la distribución continua los cambios no llegan a implementarse en la etapa de producción de forma automática, en la implementación continua sí se logra.



## Importancia de la CI/CD

La CI/CD permite que las empresas eviten errores y fallas en el código mientras mantiene un ciclo constante de desarrollo y actualizaciones del software.

A medida que las aplicaciones crecen, las funciones de la CI/CD ayudan a disminuir la complejidad, aumentar la eficiencia y optimizar los flujos de trabajo.

Como la CI/CD automatiza la intervención manual que suele necesitar para que el código nuevo pase de la etapa de confirmación a la de producción, se reduce el tiempo de inactividad y se agilizan los lanzamientos de código. Además, gracias a la capacidad de integrar actualizaciones y cambios en el código más rápido, los comentarios de los usuarios pueden incorporarse con más frecuencia y eficiencia, lo que se traduce en resultados positivos para los usuarios finales y mayor satisfacción para los clientes en general.

## Integración continua

La sigla "CI" en CI/CD hace referencia a la integración continua, un proceso de automatización que permite que los desarrolladores fusionen los cambios de código en una rama principal o una ramificación de manera más sencilla y frecuente. A medida que se realizan estas actualizaciones, se activan los pasos de prueba automatizados para garantizar la confiabilidad de los cambios fusionados.

El objetivo del desarrollo de las aplicaciones modernas es que varios desarrolladores puedan trabajar de forma simultánea en distintas funciones de la misma aplicación. Sin embargo, si una empresa fusiona todo el código fuente diversificado en un solo día (conocido como el "día de la fusión"), las tareas pueden tornarse tediosas, manuales y muy lentas.

Esto ocurre porque puede haber una incompatibilidad entre los cambios que implementen los desarrolladores en la aplicación si trabajan de forma simultánea pero aislada. El problema puede agravarse aún más si cada desarrollador personaliza su propio entorno de desarrollo integrado (IDE) local, en lugar de que todo el equipo adopte un IDE basado en la nube.

Se podría considerar a la CI como una solución al problema de que se desarrolle demasiadas ramificaciones de una aplicación al mismo tiempo, las cuales podrían entrar en conflicto entre sí.

La CI resulta exitosa cuando se fusionan las modificaciones del desarrollador, se validan con la compilación automática de la aplicación y la ejecución de distintas pruebas automatizadas (generalmente, de unidad e integración) para garantizar que los cambios no hayan ocasionado un error. Por lo tanto, se debe probar todo, desde las clases y el funcionamiento hasta los distintos módulos que conforman la aplicación. Una de las ventajas de la CI es que si se detecta un inconveniente entre el código nuevo y el actual durante una prueba automatizada, se puede solucionar de manera sencilla y rápida.

## Significado de "CD" en CI/CD

La sigla "CD" se utiliza para la distribución o la implementación continuas, y se trata de conceptos relacionados que suelen usarse indistintamente. Ambos se refieren a la automatización de las etapas posteriores del canal, pero a veces se usan por separado para explicar el alcance de la automatización. Se utiliza un término u otro según la tolerancia a los riesgos y las necesidades específicas de los equipos de desarrollo y de operaciones.

## Distribución continua

Una vez que se automatizan las compilaciones y las pruebas de unidad e integración en la CI, la distribución continua automatiza el lanzamiento del código validado en un repositorio. Por lo

tanto, para que el proceso de distribución continua sea efectivo, es importante que la CI ya esté integrada en el canal de desarrollo.

En la distribución continua, cada etapa conlleva la automatización de las pruebas y del lanzamiento del código, desde la fusión de los cambios hasta la distribución de las compilaciones listas para la producción. Al finalizar este proceso, el equipo de operaciones puede implementar rápidamente una aplicación en la etapa de producción.

Por lo general, la distribución continua se refiere a que los cambios que implementa un desarrollador en una aplicación se someten a pruebas automáticas de errores y se cargan en un repositorio (como GitHub o un registro de contenedores), para que luego el equipo de operaciones pueda implementarlos en un entorno de producción en tiempo real. Es la solución al problema de la falta de supervisión y comunicación entre los equipos comerciales y de desarrollo. Por eso, el objetivo de la distribución continua es contar con una base de código que siempre esté preparada para implementarse en un entorno de producción y garantizar que la aplicación del código nuevo sea una tarea sencilla.

## Implementación continua

El último paso de un canal de CI/CD consolidado es la implementación continua. Se trata de una extensión de la distribución continua e implica el lanzamiento automático de los cambios del desarrollador, desde el repositorio hasta la producción, para ponerlos a disposición de los clientes.

La CD aborda el problema de la sobrecarga de los equipos de operaciones con procesos manuales que retrasan la distribución de las aplicaciones. Con este tipo de implementación, se aprovechan los beneficios de la distribución continua y se automatiza la siguiente etapa del canal.

En la práctica, los cambios que implementan los desarrolladores en la aplicación de nube podrían ponerse en marcha unos cuantos minutos después de su creación (siempre que hayan pasado las pruebas automatizadas). Esto facilita mucho más la recepción e incorporación permanente de los comentarios de los usuarios. En conjunto, estas prácticas de CI/CD relacionadas reducen los riesgos que conlleva el proceso de implementación, dado que es más sencillo lanzar cambios en las aplicaciones gradualmente en lugar de hacerlo todo a la vez.

Sin embargo, como no hay ninguna entrada manual en la etapa del canal anterior a la producción, la implementación continua depende en gran medida de que la automatización de las pruebas se diseñe correctamente. Por lo tanto, se requieren muchas inversiones iniciales, ya que se deben diseñar las pruebas automatizadas para que se adapten a las distintas etapas de prueba y lanzamiento en el canal de CI/CD.

# Deployment Environments

## Beneficios de Mantener Múltiples Development Environments

- Mantener múltiples development environments ofrece los siguientes beneficios:
- Mejora la productividad del equipo y resulta en ciclos de release más rápidos.
- Hay zero downtime porque los development y testing environments son diferentes del entorno en vivo.
- Múltiples environments ayudan a lograr una mejor seguridad porque puedes aplicar restricciones más estrictas en el servidor en vivo.
- Da la capacidad de comercializar tu producto más rápido.
- Permite experimentar y probar diferentes ideas innovadoras.

Los principales deployment environments usados en el desarrollo de software son production, staging, UAT, development, y entornos de vista previa (o en otras palabras, ephemeral environments – entornos efímeros). Analicemos cada uno y exploremos la necesidad de cada uno de ellos.

## Production Environment

El Production environment es un servidor en vivo que todos los clientes usan para realizar sus actividades de negocio. Este es el environment donde se realiza el final deployment del producto. Se llama entorno "en vivo" porque los clientes interactúan con él en vivo. Usualmente, el acceso al servidor de prod es limitado y altamente seguro.

No se espera que un bug sea encontrado en el servidor de prod directamente; de lo contrario, podría impactar la experiencia del cliente. Un bug encontrado en prod también se llama "defecto escapado" porque escapó de la vista del QA (Quality Assurance) en Staging y User Acceptance Testing (UAT).

Finalmente, la infraestructura de este environment es resiliente para que tu producto esté siempre disponible y pueda tolerar cualquier pérdida de infraestructura.

Este environment es usado por usuarios reales. Por eso, usualmente no se crea al inicio del desarrollo del producto, porque aún no hay clientes. Es solo cuando tu MVP (Minimum Viable Product) o la versión inicial del producto está lista que creas este environment para ser usado por clientes reales.

Este environment también se usa donde algunas partes de tu sistema están expuestas para ser integradas por otros proveedores. Si haces una API call a un proveedor, accedes a la API deployed en su servidor de prod. Así que, si ves una feature o bug fix en production, entonces es seguro asumir que fue testeado y aprobado no solo en el servidor de Development y Staging sino también en el User Acceptance Testing (UAT). Este environment contiene los datos reales y en vivo de los clientes.

## Staging Environment

Antes de que el producto sea deployado a prod, necesita ser probado correctamente en un pre-production environment, más comúnmente conocido como el Staging environment. Un servidor de staging no es tan estable como el servidor de prod porque las features y los bug fixes aún están bajo testing en Staging.

Cualquier bug o nueva mejora reportada en el servidor de staging se incorpora primero en el servidor de development y luego se mueve al servidor de staging. Puedes pensarlo como un puente entre development y prod. La mayoría de los servidores de staging son réplicas muy cercanas del servidor de prod y a veces se consideran como "pre-production".

El propósito principal de este environment es asegurar que el software ha sido testeado lo suficiente para ser deployado en un production o UAT environment. Aquí es donde los ingenieros de Quality Assurance (QA) realizan pruebas adecuadas y se aseguran de que la aplicación esté funcionando de acuerdo con las expectativas de los interesados. El servidor de staging también se usa para ejecutar pruebas de rendimiento y carga.

## User Acceptance Testing (UAT) Environments

Después de que el software testing se completa en Staging, se deployed al UAT environment para las pruebas del cliente. El software running en UAT debe asegurar que es un producto viable y está listo para los usuarios. Un UAT environment es similar a Staging en términos de testing. Como Staging, se usa primariamente para testing bug fixes y new features. Sin embargo, a diferencia de Staging, el cliente o product owner realiza el testing en este environment.

Este environment se usa para evaluar si el software running en él será aceptado por el usuario final, de ahí el nombre "User Acceptance Testing". Usualmente, los bugs relacionados con la usabilidad de los usuarios se encuentran y corrigen en este environment. Es la última etapa antes de que el producto se pone en vivo. Después de que el software es aprobado en UAT, se deployed a production.

## Development Environment

El Development environment es donde los developers desarrollan el producto y deploy su code por primera vez. Como los developers continuamente cambian su code para actualizar el producto, este environment es altamente volátil. Es perfectamente posible que este environment esté down and inaccessible por un breve período de tiempo. Este environment no contiene datos de clientes, y utiliza datos creados por los developers. Las branches de otros developers también se merged en este environment. Este environment es primariamente para que los developers desarrollen, merge y test sus tickets antes de que vaya al QA team.

Cualquier bug fixes encontrado en cualquier otro environment se reproduce y se corrige en este environment primero. Las new features también se desarrollan aquí primero y son testeadas por developers en este environment.

El development environment es la primera etapa cuando se inicia el desarrollo del producto. Un producto que está en modo de mantenimiento raramente necesitará este environment. Mientras que un producto en su fase MVP lo necesitará más, este es un environment perfecto para debugging cualquier problema y un new feature preview.

Este environment también apoya la colaboración rápida entre diferentes miembros del equipo multifuncional, por ejemplo, la incorporación inicial de feedback del product owner, QA, diseñador de UI, etc.

## Preview Environments

Los Preview environments (también conocidos como Ephemeral Environments) han revolucionado la forma en que las pull requests son verificadas. El enfoque tradicional es así: Después de que el developer pushes code en su branch, clonas esa branch en tu máquina local, deploy los cambios a tu local environment, y actualizas las integraciones externas y fuentes de datos. Esto crea muchas dependencias en el flujo de trabajo.

Los Preview environments resuelven este problema con gran simplicidad. Tan pronto como una pull request es creada, un environment es automáticamente creado con todas las dependencies, incluyendo el new code, configuración, datos, etc. Y se puede acceder a este environment a través de una URL. Esto da gran flexibilidad a otros miembros del equipo para evaluar los cambios de forma aislada sin necesitar hacer ningún cambio a su local environment o configurar un correcto Staging Environment. Como los Preview environments están asociados con el ciclo de vida de la pull request, y son destruidos tan pronto como la pull request is merged in master (se fusiona en la rama master), por eso se llaman ephemeral environments o dynamic environments.

Los Preview environments proporcionan muchos beneficios para el negocio. Tus ciclos de release se acortan porque se gasta menos tiempo en la preparación y deployment de un new environment. El costo también se reduce porque los Preview environments son de corta duración, y no se necesita infraestructura permanente. La calidad del producto mejora porque da poder a usuarios no técnicos como los product owners para test the functionality y proporcionar feedback al instante. Todos estos factores mejoran la satisfacción del cliente y te ayudan a alcanzar tus objetivos de negocio más rápido.

Si quieres visualizar y test los cambios en el code pull request y no eres lo suficientemente técnico para configurar un pre-production environment, entonces un Preview environment es justo lo que necesitas. Ya que el acceso al preview environment es a través de una URL, tienes la flexibilidad de permitir que solo ciertos miembros del equipo accedan a él.

Los Preview environments también son muy útiles cuando necesitas ejecutar diferentes tests de automatizacion y performance de manera aislada y no quieres la molestia de configurar un separate environment.

Production	Staging	UAT	Development	Preview
Highly Stable	Stable	Highly Stable	Less Stable	Less Stable
Used by actual users for business	Used by QA for testing	Used by actual users for testing	Used by developers	Used by developers, testers, product owners
Full production data	Limited production data	Moderate production data	No customer data	No customer data
Deployed after build is approved on UAT	Deployed after build is approved on Dev	Deployed after build is approved on staging	First deployment is made here	Deployed as soon as code PR is created

 Qovery

## Git branching

En el mundo actual de la continuous delivery and deployment (entrega e deployment continuo), una clara branching strategy y un flujo de deployment estructurado son esenciales. Estas prácticas aseguran que el code se mueva sin problemas desde development hasta production, permitiendo a los equipos de development trabajar de manera eficiente y con mínimo riesgo. En esta publicación, exploraremos estrategias comunes de branching, flujos de trabajo de deployment, y cómo habilitan un development y gestión de release efectivos para aplicaciones con múltiples environments.

Una branching strategy (estrategia de ramificación) bien definida es crucial en cualquier equipo de desarrollo de software, especialmente cuando se gestionan múltiples environments como test, staging y production. Al organizar los cambios de code a través de branches (ramas), los equipos pueden:

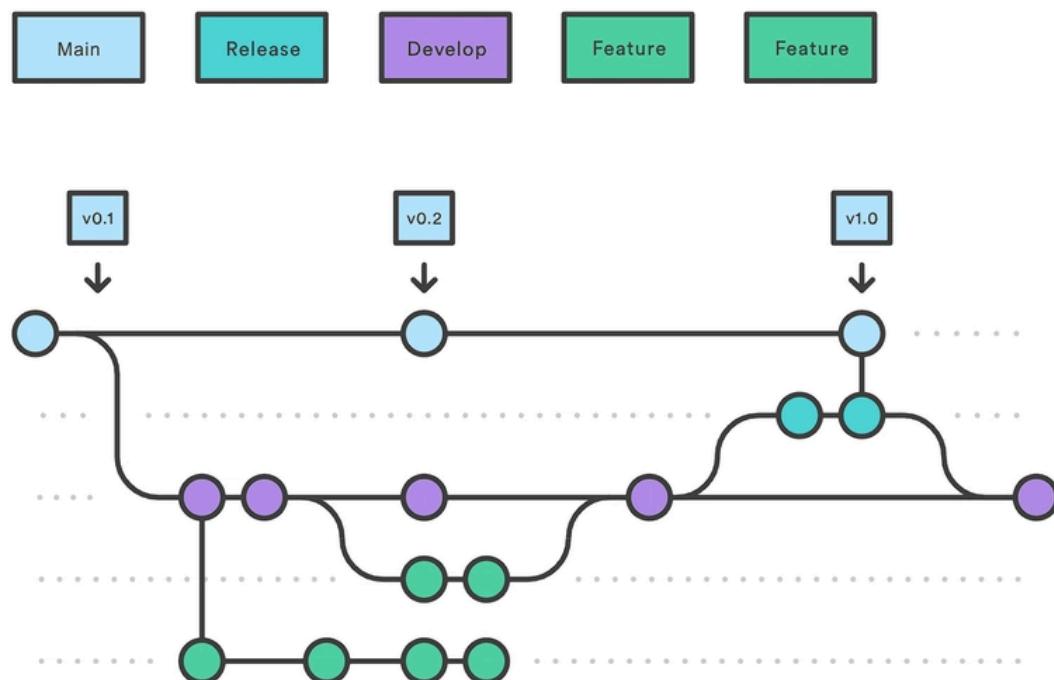
- Colaborar sin afectar el trabajo de otros.
- Test el code de forma incremental en environments controlados.
- Deploy code estable y listo para production.

La branching strategy correcta soporta un flujo suave y continuo desde el desarrollo de features hasta el final production deployment, reduciendo bugs y ayudando a prevenir problemas de último minuto.

Si bien existen muchas branching strategies, aquí hay tres populares que funcionan bien con multi-environment deployments:

## GitFlow

GitFlow es uno de los branching models (modelos de ramificación) más conocidos, introducido por Vincent Driessen en 2010. Define una estructura estricta de branching diseñada en torno a los releases del proyecto.



### Estructura

- Main branches (ramas principales):
  - master (o main): Almacena el historial oficial de release.
  - develop: Sirve como integration branch (rama de integración) para features.
- Supporting branches (ramas de soporte):
  - feature/\*: Para el desarrollo de new features.
  - release/\*: Para la preparación del release.
  - hotfix/\*: Para correcciones urgentes de production.
  - bugfix/\*: Para correcciones a la develop branch.

## Workflow

1. El Development ocurre en la develop branch.
2. El trabajo de feature sucede en feature/\* branches creadas a partir de develop.
3. Cuando hay suficientes features listas, se crea una release/\* branch.
4. Las release branches se merge a master y a develop cuando están listas.
5. Los bugs críticos en production se corrigen en hotfix/\* branches.
6. Los Hotfixes se merge a master y a develop.

## Pros

- Estructurado con clear versioning (versionado claro)
- Aísla el nuevo development del trabajo finalizado.
- Excelente para scheduled releases (releases programados) y QA formal.
- Soporta el development paralelo a través de versiones.

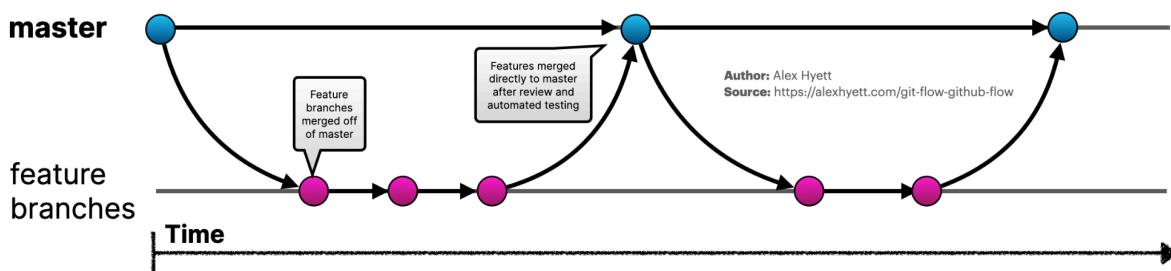
## Contras

- Complejo con muchas branches que gestionar.
- Sobrecarga excesiva para proyectos más pequeños.
- No es ideal para continuous deployment.
- Puede llevar a grandes conflictos de merge.

## GitHub Flow

GitHub Flow es un flujo de trabajo liviano, basado en branches, desarrollado por GitHub. Es más simple que GitFlow y está diseñado para equipos que practican la continuous delivery (entrega continua).

## GitHub Flow



## Estructura

- Main branch:
  - main: Siempre deployable (desplegable) con code estable.
- Supporting branches:
  - Feature/bugfix branches: Creadas desde main y merged de vuelta a main.

## Workflow

1. Crea una branch desde main para cualquier trabajo nuevo.
2. Añade commits a tu branch.
3. Abre una pull request para discusión.
4. Revisa, test y haz cambios según sea necesario.
5. Haz merge a main cuando esté aprobado.
6. Deploy inmediatamente después de hacer merge a main.

## Pros

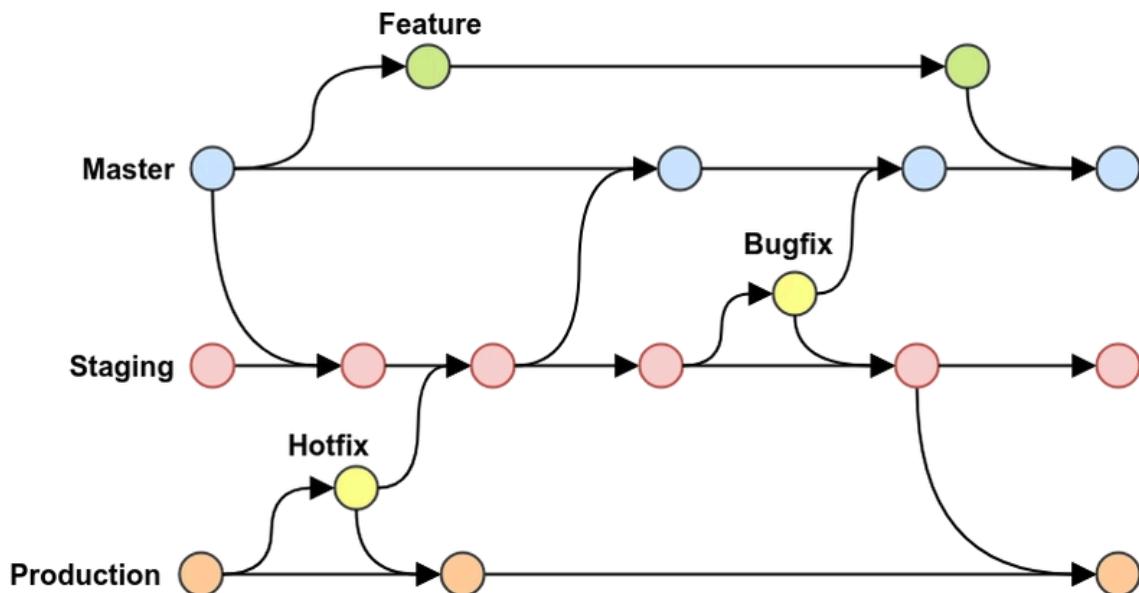
- Simple y fácil de entender.
- Perfecto para continuous deployment.
- Minimiza la gestión de branches.
- Se centra en la colaboración mediante pull request.

## Contras

- Soporte limitado para gestionar releases.
- No hay staging o integration testing claro.
- Arriesgado sin un automated testing robusto.
- Difícil para mantener múltiples versiones.

## GitLab Flow

GitLab Flow es un compromiso entre GitFlow y GitHub Flow, añadiendo environment branches y release branches a la simplicidad de GitHub Flow.



## Estructura

- Main branch:

- main: Integration branch que siempre está lista para deployment.
- Environment branches:
  - production, staging, pre-production: Representan los deployed environments.
- Feature branches:
  - Creadas desde main para cualquier trabajo nuevo.

## Workflow

1. Crea feature branches desde main.
2. Haz merge de las feature branches de vuelta a main a través de merge requests.
3. Los cambios fluyen desde main a las environment branches a medida que están listos.
4. Para software con versiones, crea release branches cuando sea necesario.

## Pros

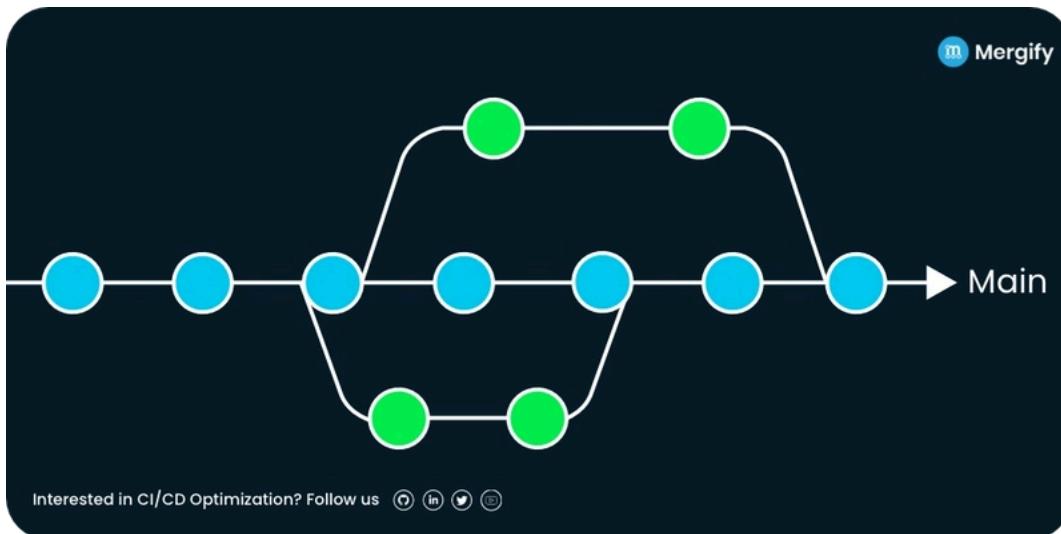
- Equilibra simplicidad con estructura.
- Proporciona etapas claras de deployment.
- Soporta tanto continuous delivery como versioned releases.
- Se centra en el production environment.

## Contras

- Más complejo que GitHub Flow.
- Puede crear "deriva" entre environments.
- Desafiante para múltiples versiones.
- Puede confundir dónde aplicar las correcciones.

## Trunk-Based Development

Trunk-Based Development es un patrón de control de fuente donde los developers colaboran en code en una sola branch llamada "trunk" (a menudo main o master), y evitan feature branches de larga duración.



## Estructura

- Main branch:
  - main: Fuente única de verdad donde ocurre todo el development.
- Supporting branches:
  - Short-lived feature branches (máximo 1-2 días).
  - Release branches (opcional, para estabilización del release).

## Workflow

1. Los developers hacen pequeños, y commits frecuentes a main.
2. Los Feature toggles controlan la visibilidad de las features en progreso.
3. Se hace merge de las short-lived feature branches diariamente.
4. CI/CD asegura que el trunk permanezca estable.
5. Se pueden crear release branches para estabilización.

## Pros

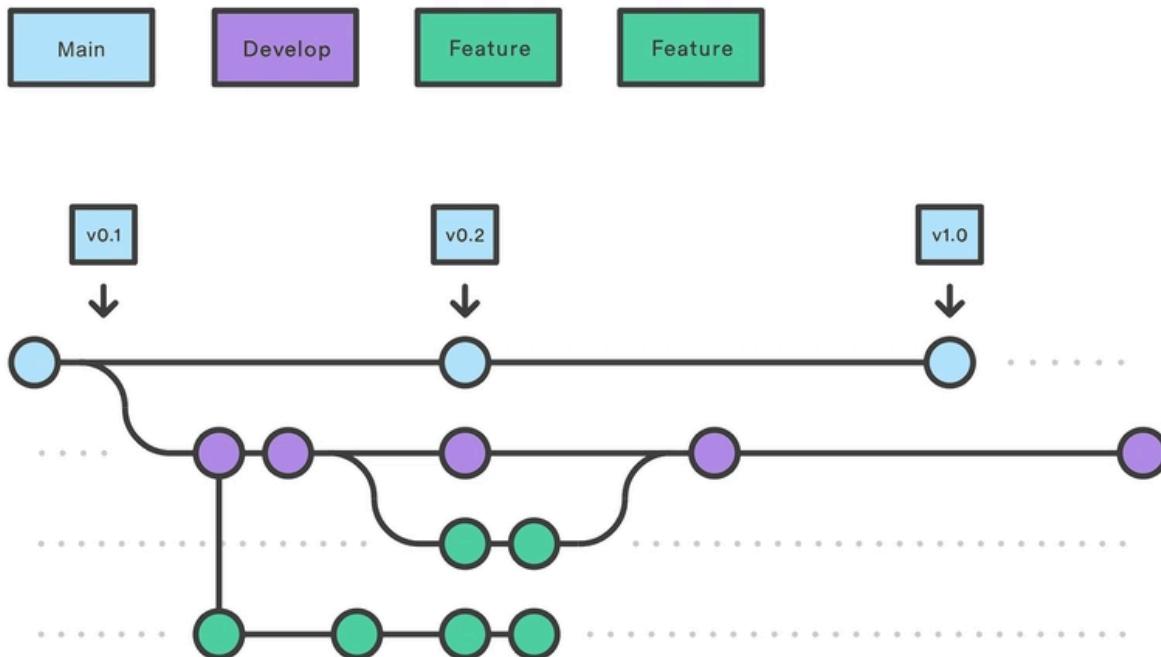
- Aplica continuous integration.
- Minimiza los conflictos de merge.
- Fomenta cambios pequeños e incrementales.

## Contras

- Requiere sophisticated testing (testing sofisticado).
- Se basa en feature toggles.
- Menos aislamiento entre features.
- Mayor riesgo de bugs en main.

## Feature Branching

Feature Branching se centra en aislar todo el trabajo relacionado con una feature específica en branches dedicadas, proporcionando un alto nivel de aislamiento.



### Estructura

- Main branch:
  - main: Code estable que está listo para producción.
- Feature branches:
  - Una branch por feature, bug fix o mejora.

### Workflow

1. Crea una new branch para cada feature o task (tarea).
2. Desarrolla y test la feature de forma aislada.
3. Envía una pull/merge request cuando esté completa.
4. Revisa, test, y haz merge a main cuando esté listo.
5. Borra la feature branch después del merge.

### Pros

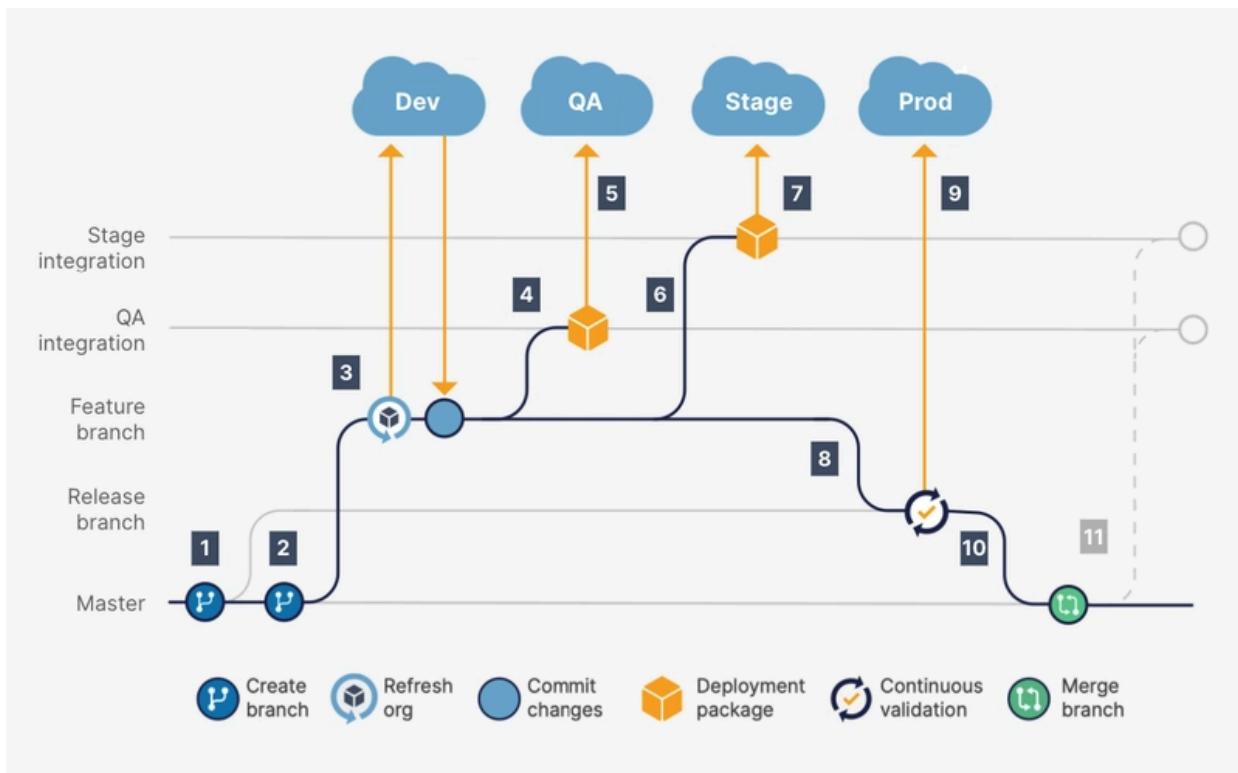
- Aislamiento claro del trabajo.
- Fácil seguimiento de features.
- Permite la experimentación de forma segura.
- Habilita el trabajo independiente paralelo.

### Contras

- Desafíos de integración con long-lived branches.
- Riesgo de demasiadas active branches.
- Puede retrasar la detección de problemas de integración.
- Desafiante con interdependent features.

## Environment Branching

Environment Branching usa dedicated branches para representar diferentes deployment environments en tu infraestructura.



### Estructura

- Environment branches:
  - development: Integration branch para el trabajo en curso.
  - staging: Pre-production testing.
  - production: Entorno en vivo.
- Feature branches:
  - Creadas desde y merged de vuelta a development.

### Workflow

1. El Development ocurre en feature branches a partir de development.
2. Se hace merge de las Features de vuelta a development cuando están listas.
3. Los cambios fluyen de development → staging → production.
4. Cada promoción es una operación de merge.

### Pros

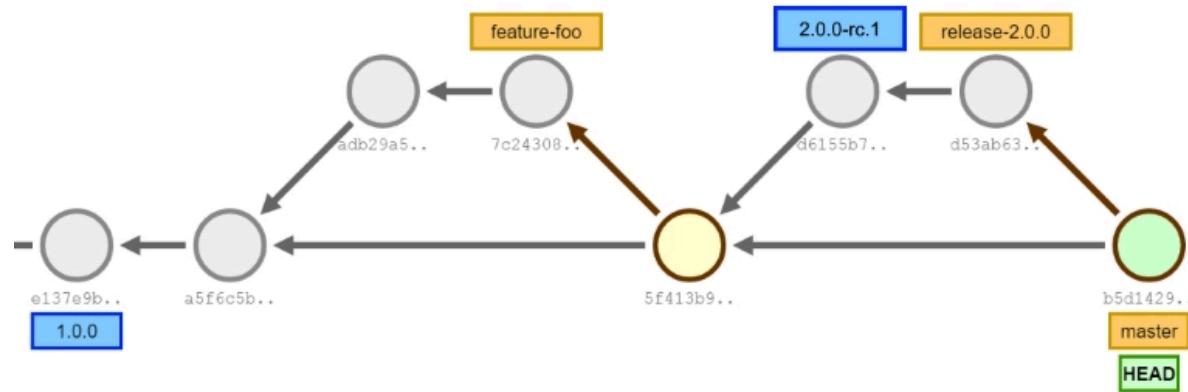
- Representación clara del deployment pipeline.
- Seguimiento visual de deployments.
- Capacidad fácil de rollback.
- Soporta procesos formales de deployment.

### Contras

- Puede crear escenarios de merge complicados.
- Riesgo de environment branch divergence (divergencia de la rama de environment).
- Sobrecarga de mantener múltiples branches.
- No es ideal para releases frecuentes.

## Release Branching

Release Branching se centra en estabilizar code para releases específicos mientras permite que el development continúe para releases futuros.



## Estructura

- Main branches:
  - main: Último development code.
- Release branches:
  - release/x.y.z: Una branch por versión de release planificada.

## Workflow

1. El Development ocurre en main o feature branches.
2. Cuando comienza un ciclo de release, se crea una release/x.y.z branch.
3. Solo los bug fixes se hacen commit a la release branch.
4. Las New features continúan en main.
5. Cuando es estable, el release es merged o etiquetado (tagged).
6. Los fixes importantes son seleccionados (cherry-picked) de vuelta a main.

## Pros

- Habilita el trabajo paralelo en múltiples releases.
- Proporciona stable branches para testing.
- Enfoque claro de versioning.
- Soporta el mantenimiento de múltiples releases.

## Contras

- Requiere seguir los fixes a través de branches.
- Puede crear escenarios de merge complejos.
- Riesgo de que los fixes no se propaguen correctamente.
- Desafiantes para correcciones urgentes de múltiples releases.

## Deployment Pipelines

### Azure DevOps

Azure DevOps es una plataforma basada en la nube que proporciona herramientas integradas para los equipos de desarrollo de software. Incluye todo lo que necesita para planear el trabajo, colaborar en código, compilar aplicaciones, probar funcionalidad e implementar en producción.

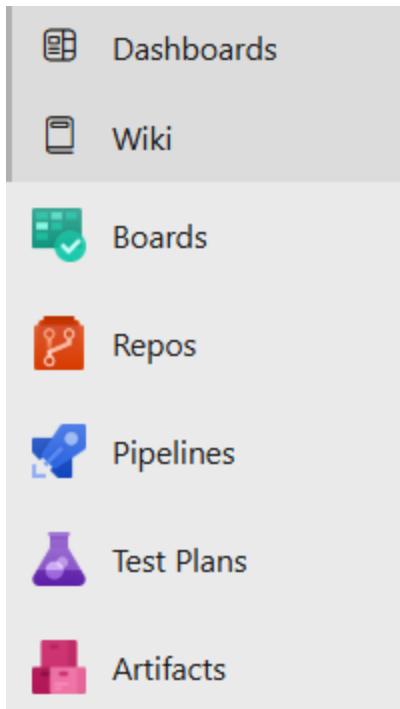
Azure DevOps ofrece un espectro de modelos de servicio para adaptarse a las necesidades únicas de cada equipo. La versión de acceso gratuito ayuda a los equipos pequeños a empezar a trabajar rápidamente, mientras que los planes versátiles de suscripción y pago por uso admiten una administración completa de proyectos.

### Características clave

- Administración de proyectos de un extremo a otro: Azure DevOps es un conjunto de servicios cohesivo diseñado para admitir el ciclo de vida completo de los proyectos de software. Abarca todo, desde el planeamiento inicial y el desarrollo, a través de rigurosas pruebas hasta la implementación final.
- Entrega de modelos de cliente/servidor: Azure DevOps funciona en un modelo de cliente o servidor, lo que ofrece flexibilidad en la forma de interactuar con sus servicios. La interfaz web proporciona una manera cómoda de utilizar la mayoría de los servicios y es compatible con todos los exploradores principales. Además, algunos servicios como el control de código fuente, las canalizaciones de compilación y el seguimiento de trabajos ofrecen opciones de administración basadas en cliente para un control mejorado.
- Opciones de servicio flexibles y escalables: Azure DevOps atiende a los equipos de todos los tamaños al ofrecer una variedad de opciones de servicio. Para equipos pequeños, muchos servicios son gratuitos, asegurándose de que tiene acceso a herramientas sólidas de administración de proyectos sin ninguna inversión inicial. Para equipos más grandes o necesidades más avanzadas, los servicios son accesibles a través de un modelo de suscripción o de pago por uso.

### Servicios principales

Azure DevOps incluye los siguientes servicios integrados:



**Azure Boards:** planifique y realice un seguimiento del trabajo mediante herramientas de Agile, tableros Kanban, trabajos pendientes y paneles. Cree elementos de trabajo como casos de usuario, errores y tareas. Use el planeamiento de sprints, los gráficos de agotamiento y el seguimiento de velocidad. Personalice los flujos de trabajo y los tipos de elementos de trabajo para que coincidan con el proceso del equipo.

Escenario de ejemplo: Un equipo de producto que planea una característica de aplicación móvil crea casos de usuario para "inicio de sesión de usuario", realiza un seguimiento de los errores encontrados durante el desarrollo y usa paneles de sprint para supervisar el progreso durante las iteraciones de dos semanas.

**Azure Repos:** hospede repositorios de Git privados ilimitados o use el control de versiones de Team Foundation (TFVC) para la administración de código fuente. Las características incluyen políticas de rama, solicitudes de incorporación de cambios con revisiones de código, resolución de conflictos e integración con IDE y editores populares.

Escenario de ejemplo: Los miembros del equipo de desarrollo crean ramas de características para nuevas funcionalidades, envían solicitudes de incorporación de cambios para la revisión de código y usan directivas de rama para asegurarse de que todo el código se revisa y prueba antes de combinar con la rama principal.

**Azure Pipelines:** compile, pruebe e implemente aplicaciones con canalizaciones de CI/CD que funcionan con cualquier lenguaje, plataforma y nube. Admite contenedores de Docker, Kubernetes e implementaciones en Azure, AWS, Google Cloud o local. Incluye trabajos paralelos, filtros de implementación y aprobaciones de liberación.

Escenario de ejemplo: Cada confirmación de código desencadena una canalización automatizada que compila una aplicación web de .NET, ejecuta pruebas unitarias, crea un contenedor de Docker e implementa en el entorno de ensayo para realizar pruebas antes de la versión de producción.

**Test Plans de Azure:** planee, ejecute y realice un seguimiento de las pruebas con casos de prueba manuales, sesiones de pruebas exploratorias e integración de pruebas automatizadas. Cree conjuntos de pruebas, realice un seguimiento de los resultados de las pruebas, capture capturas de pantalla y vídeos y genere informes de prueba detallados.

Escenario de ejemplo: El equipo de control de calidad crea casos de prueba para el flujo de registro de usuarios, ejecuta pruebas manuales en distintos exploradores, capture capturas de pantalla de problemas y vincula los resultados de las pruebas a casos de usuario para la rastreabilidad.

**Azure Artifacts:** cree, hospede y comparta paquetes como NuGet, npm, Maven, Python y Universal con su equipo y organización. Intégrese con canalizaciones de compilación, administre versiones de paquetes y controle el acceso con orígenes ascendentes y políticas de retención.

Escenario de ejemplo: El equipo de desarrollo crea una biblioteca de autenticación compartida, la publica como un paquete NuGet en Azure Artifacts y hace referencia a ella en varios proyectos mientras controla el acceso a paquetes internos.

## Linked services

Esto no es un servicio como tal pero es una funcionalidad clave para que Azure sea usable en un entorno comercial.

Los enlaces de servicio automatizan las interacciones con servicios externos y responden a eventos de proyecto. Configure enlaces para enviar notificaciones, desencadenar acciones o integrarse con herramientas que no son de Microsoft cuando se produzca un error en las compilaciones, se confirme el código o cambien los elementos de trabajo. Nos permite conectarnos a servicios externos como Azure o GitHub pudiendo acceder a sus recursos y consumirlos en las Azure Pipelines en cualquier otro servicio de Azure DevOps

## Azure Pipelines

es la parte de Azure DevOps que combina la integración continua, las pruebas continuas y la entrega continua para compilar, probar e implementar proyectos de código automáticamente en cualquier destino. Azure Pipelines admite todos los lenguajes principales y tipos de proyecto, y puede automatizar flujos de trabajo en las tecnologías y marcos elegidos, tanto si la aplicación es local como en la nube.

The screenshot shows the Azure DevOps Pipelines interface for the 'FabrikamFiber' project. On the left, there's a sidebar with navigation links: Overview, Boards, Repos, Pipelines (which is selected and highlighted with a red box), Environments, Releases, Library, Task groups, and Deployment groups. The main area is titled 'Pipelines' and shows a table of recently run pipelines. The table has two columns: 'Pipeline' and 'Last run'. It lists two entries: 'pipelines-dotnet-core' (last run 32m ago, manually triggered) and 'FabrikamFiber' (last run 1h ago, manually triggered). A 'New pipeline' button is located in the top right corner of the main area.

Pipeline	Last run
pipelines-dotnet-core	#20191209.2 • Set up CI with Az... Manually triggered ⚡ main 32m ago 42s
FabrikamFiber	#20191209.3 • Set up CI with Az... Manually triggered ⚡ main 1h ago 1m 13s

## Ventajas de Azure Pipelines

Azure Pipelines proporciona una manera rápida, fácil y segura de automatizar la creación de proyectos con código coherente y de alta calidad.

Azure Pipelines ofrece las siguientes ventajas:

- Despliega en diferentes tipos de destinos simultáneamente
- Se integra con implementaciones de Azure
- Se integra con GitHub
- Funciona con cualquier lenguaje o plataforma
- Funciona en máquinas Windows, Linux o Mac
- Funciona con proyectos de código abierto

## Construcción de pipelines

En esta sección, especificaremos los pasos que Azure Pipelines realiza para construir tu código, como la instalación de dependencias, la compilación y la ejecución de pruebas. Vamos a expresar estos pasos en un archivo de configuración en nuestro repositorio de código fuente utilizando YAML, una forma sencilla y legible por humanos de describir los pasos de una receta.

## Sintaxis de la canalización YAML

Aquí tienes un ejemplo sencillo de una canalización YAML:

```
trigger:
- main

pool:
  vmImage: 'ubuntu-latest'

steps:
- script: echo Hello, world!
  displayName: 'Run my first pipeline script'
```

Vamos a desglosar el código paso a paso:

- **trigger**: Especifica la rama que activa la canalización (por ejemplo, `main`).
- **pool**: Define el agente de construcción (por ejemplo, `ubuntu-latest`).
- **steps**: Enumera las tareas a ejecutar, como instalar dependencias y ejecutar un script de compilación.

### Configurar los pasos de construcción

Para añadir pasos de compilación a tu pipeline YAML, debes definir el "script" en la sección "pasos".

Para una aplicación sencilla de `Node.js`, por ejemplo, nuestro archivo `azure-pipelines.yml` de abajo contiene código que Azure DevOps creó automáticamente.

```
trigger:
- main

pool:
  vmImage: 'ubuntu-latest'

steps:
- task: NodeTool@0
  inputs:
    versionSpec: '20.x'
  displayName: 'Install Node.js'

- script: |
  npm install
```

```
npm run build  
displayName: 'npm install and build'
```

Echemos un vistazo a la construcción `steps` en el canal YAML anterior:

1. **Instala Node.js:** Utiliza la tarea `NodeTool@0` para especificar la versión de Node.js.
2. **Instala las dependencias:** Ejecuta `npm install` para instalar las dependencias del proyecto.
3. **Compila el código (construye el proyecto):** Ejecuta `npm run build` para compilar el código.

## Ejecutar la pipeline

Nuestro pipeline puede activarse manualmente o configurarse para que se ejecute automáticamente cuando se produzcan eventos como el empuje de ramas.

- **Ejecucion manual:** Haz clic en **Ejecutar canalización** en la interfaz Azure DevOps para activar la canalización manualmente.



- **Ejecucion automática:** Establece desencadenantes basados en eventos del repositorio para enviar cambios a la rama `main` y desencadenar la canalización automáticamente. He aquí un ejemplo:

```
trigger:  
- main
```

## Agents

Necesita al menos un agente para compilar el código o implementar el software mediante Azure Pipelines. A medida que crece el código base y el equipo, necesita varios agentes.

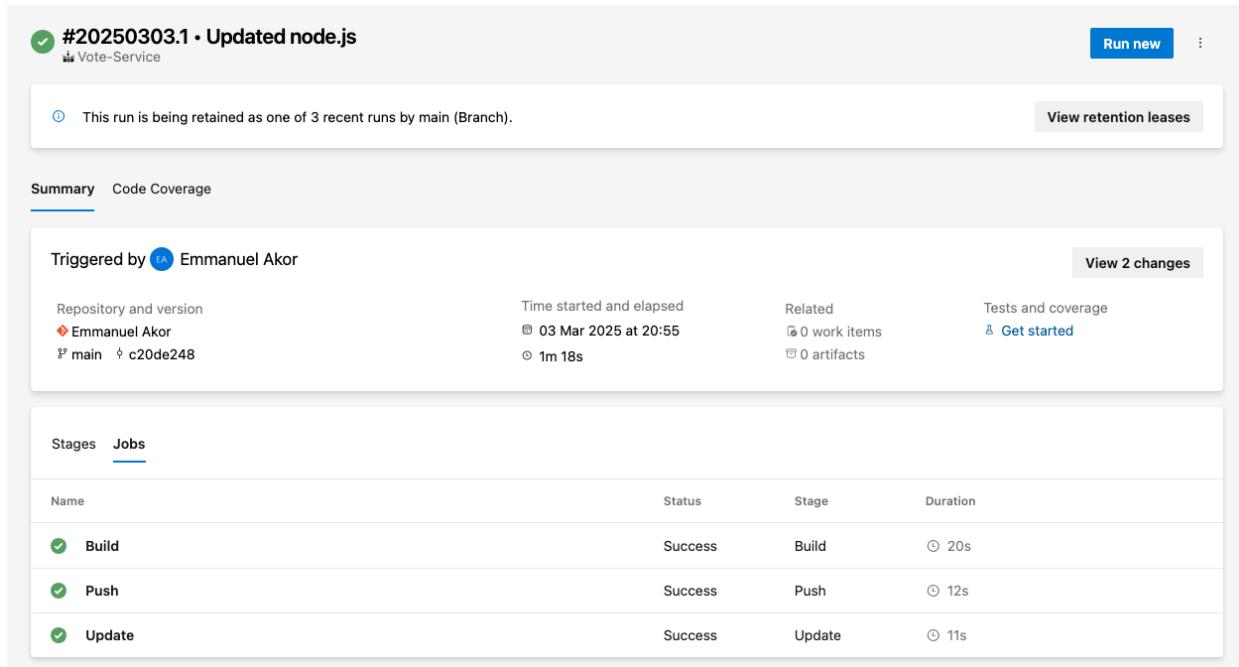
Cuando la canalización se ejecuta, el sistema inicia uno o varios trabajos. Un agente es una infraestructura informática con software de agente instalado que ejecuta un trabajo de uno en uno.

Azure Pipelines proporciona varios tipos diferentes de agentes.

- Agentes hospedado por Microsoft: el mantenimiento y las actualizaciones se producen automáticamente.
- Agentes autohospedados: Un *agente autohospedado* es un agente que configura para ejecutar trabajos y que usted mismo administra. La instalación es on premise, es decir en la PC o en un servidor local
- Agentes de Azure de los Conjuntos de Escala de Máquinas Virtuales: una forma de agentes autohospedados que se pueden escalar automáticamente para satisfacer sus demandas.
- Agentes de grupos de DevOps administrados: grupos de agentes de Azure DevOps adaptados a las necesidades específicas de un equipo.

Después de ejecutar tu pipeline, puedes ver los registros y solucionar cualquier problema.

Para acceder a los registros, ve al resumen de ejecución de tu pipeline y selecciona el trabajo o tarea específicos. Esto mostrará los registros de ese paso; como en la imagen de abajo, puedes elegir cualquier trabajo (Construir, Empujar o Actualizar) para ver los registros sin procesar.



The screenshot shows the Azure DevOps pipeline run summary for job #20250303.1. The 'Jobs' tab is selected, displaying three stages: Build, Push, and Update, all of which completed successfully. The 'Summary' tab is also visible at the top left.

Name	Status	Stage	Duration
Build	Success	Build	20s
Push	Success	Push	12s
Update	Success	Update	11s

The screenshot shows two side-by-side windows from the Azure Pipelines interface. The left window is titled 'Jobs in run #20250303.1' for the 'Vote-Service' project. It lists a series of build steps: 'Build' (status green, duration 20s), 'Initialize job' (status grey, duration 2s), 'Checkout Emmanuel ...' (status green, duration 3s), 'Build the image' (status green, duration 12s), 'Post-job: Checkout E...' (status green, duration <1s), and 'Finalize Job' (status grey, duration <1s). Below these are 'Push' (status green, duration 12s) and 'Update' (status green, duration 11s) steps, followed by 'Finalize build' and 'Report build status'. The right window is titled 'Finalize Job' and shows the log output for this specific step:

```
1 Starting: Finalize Job
2 Cleaning up task key
3 Start cleaning up orphan processes.
4 Finishing: Finalize Job
```

Para solucionar problemas comunes, puedes comprobar tu canalización en busca de problemas comunes como tiempos de espera, limitaciones de recursos o configuraciones erróneas.

## Precios de Azure Pipelines

Azure DevOps ofrece un nivel gratuito de trabajos paralelos a cada organización para proyectos tanto alojados por Microsoft como autohospedados, tanto privados como públicos. En el caso de los proyectos privados, el nivel gratis proporciona un trabajo paralelo que puede tardar hasta 60 minutos en ejecutarse, hasta 1800 minutos al mes. En el caso de los proyectos públicos, la concesión gratuita proporciona un trabajo paralelo con minutos ilimitados para agentes autohospedados o hasta 10 trabajos paralelos para proyectos hospedados por Microsoft.

Los proyectos públicos y algunos proyectos privados de las nuevas organizaciones de Azure DevOps no obtienen automáticamente la concesión gratuita de trabajos paralelos de forma predeterminada. Debe solicitar el otorgamiento gratuito de trabajos paralelos al completar la solicitud de Azure DevOps Parallelism Request. La solicitud puede tardar varios días laborables en procesarse.

Si el nivel gratuito de trabajos paralelos no es suficiente para el proyecto, puede comprar más capacidad por trabajo paralelo o comprar más trabajos paralelos. Los trabajos paralelos de pago pueden tardar hasta 360 minutos en ejecutarse y no tener ningún límite de tiempo mensual.

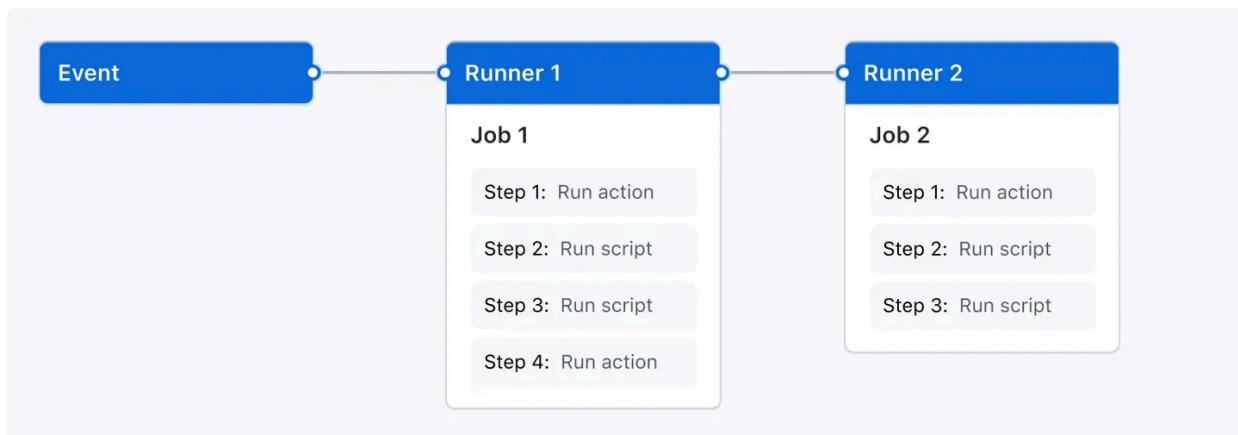
## GitHub Actions

GitHub Actions es una plataforma de integración y despliegue continuos (IC/DC) que te permite automatizar tu mapa de compilación, pruebas y despliegue. Puedes crear flujos de trabajo y crear y probar cada solicitud de cambios en tu repositorio o desplegar solicitudes de cambios fusionadas a producción.

GitHub Actions va más allá de solo DevOps y te permite ejecutar flujos de trabajo cuando otros eventos suceden en tu repositorio. Por ejemplo, puedes ejecutar un flujo de trabajo para que agregue automáticamente las etiquetas adecuadas cada que alguien cree una propuesta nueva en tu repositorio.

GitHub proporciona máquinas virtuales Linux, Windows y macOS para que ejecutes tus flujos de trabajo o puedes hospedar tus propios ejecutores auto-hospedados en tu propio centro de datos o infraestructura en la nube.

Puede configurar un flujo de trabajo de GitHub Actions que se desencadene cuando se produzca un evento en el repositorio, por ejemplo, la apertura de una solicitud de incorporación de cambios o la creación de una incidencia. El flujo de trabajo contiene uno o varios trabajos que se pueden ejecutar en orden secuencial o en paralelo. Cada trabajo se ejecutará dentro de su propio ejecutor de máquina virtual o dentro de un contenedor, y tendrá uno o varios pasos que pueden ejecutar un script que defina, o bien una acción, que es una extensión reutilizable que puede simplificar el flujo de trabajo.



## Workflow

Un flujo de trabajo es un proceso automatizado configurable que ejecutará uno o más trabajos. Los flujos de trabajo se definen mediante un archivo de YAML que se verifica en tu repositorio y se ejecutará cuando lo active un evento dentro de este o puede activarse manualmente o en una programación definida.

Los flujos de trabajo se definen en el directorio `.github/workflows` de un repositorio. Un repositorio puede tener varios flujos de trabajo, y cada uno puede realizar un conjunto diferente de tareas, como las siguientes:

- Compilar y probar de solicitudes de incorporación de cambios
- Implementar la aplicación cada vez que se crea una versión
- Agregar una etiqueta cada vez que se abre una incidencia nueva

Puede hacer referencia a un flujo de trabajo dentro de otro flujo de trabajo.

## Eventos

Un evento es una actividad específica en un repositorio que desencadena una ejecución de flujo de trabajo. Por ejemplo, una actividad puede originarse desde GitHub cuando alguien crea una solicitud de cambios, abre una propuesta o sube una confirmación a un repositorio. También puede desencadenar una ejecución de flujo de trabajo según una programación, mediante la publicación en una API de REST o manualmente.

## Trabajos

Un \*\*trabajo\*\* es un conjunto de pasos de un flujo de trabajo que se ejecuta en el mismo ejecutor. Cada paso puede ser un script de shell o una acción que se ejecutarán. Los pasos se ejecutarán en orden y serán dependientes uno del otro. Ya que cada paso se ejecuta en el mismo ejecutor, puedes compartir datos de un paso a otro. Por ejemplo, puedes tener un paso que compile tu aplicación, seguido de otro que pruebe la aplicación que se compiló.

Puede configurar las dependencias de un trabajo con otros trabajos; de manera predeterminada, los trabajos no tienen dependencias y se ejecutan en paralelo. Cuando un trabajo depende de otro, espera a que se complete el trabajo dependiente antes de ejecutarse.

También puedes usar una matriz para ejecutar el mismo trabajo varias veces, cada una con una combinación diferente de variables, como sistemas operativos o versiones de lenguaje.

Por ejemplo, es posible que configure varios trabajos de compilación para distintas arquitecturas sin dependencias de trabajo y un trabajo de empaquetado que dependa de esas compilaciones. Los trabajos de compilación se ejecutan en paralelo y, cuando se han completado correctamente, se ejecuta el trabajo de empaquetado.

## Acciones

Una acción es un conjunto predefinido y reutilizable de trabajos o código que realiza tareas específicas dentro de un flujo de trabajo, lo que reduce la cantidad de código repetitivo que escribes en los archivos de flujo de trabajo. Las acciones pueden realizar tareas como las siguientes:

- Extraer tu repositorio de Git de GitHub
- Configurar la cadena de herramientas correcta para el entorno de compilación
- Configurar la autenticación en el proveedor de nube

Puedes escribir tus propias acciones o puedes encontrar acciones para utilizar en tus flujos de trabajo dentro de GitHub Marketplace.

## Ejecutores

Un ejecutor es un servidor que ejecuta los flujos de trabajo cuando se desencadenan. Cada ejecutor puede ejecutar un solo trabajo a la vez. GitHub proporciona ejecutores de Ubuntu Linux, Microsoft Windows y macOS para ejecutar los flujos de trabajo. Cada ejecución de flujo de trabajo se ejecuta en una máquina virtual recién aprovisionada.

GitHub también ofrece ejecutor más grande, que están disponibles en configuraciones más grandes. Para más información, consulta Uso de ejecutores más grandes.

Si necesita otro sistema operativo o una configuración de hardware específica, puede hospedar ejecutores propios.

## CI/CD - Caso Práctico

Les dejo una serie de videos a mi canal de Youtube donde realizamos un caso práctico de deployment de una API con base de datos en Azure usando una pipeline de CI / CD de Github Actions.

 [11 - API Portfolio - Deployment - Set Up de recursos en Azure](#)

 [12 - API Portfolio - Deployment - CI/CD con GitHub Actions y Azure](#)