Mejora Continua

Integración Continua: Definición

Consiste en agregar todo el código que generan los desarrolladores de un proyecto cuanto antes y lo más a menudo posible. En términos de programador, consistiría en hacer un *merge* a la rama *master* del repositorio de código una vez terminada una funcionalidad concreta.

Desarrollo tradicional

Antes de ir al concepto de integración continua, es importante entender cómo se desarrollaba el software antes de la aparición de la integración continua.

En el desarrollo tradicional, en los repositorios de código suelen existir las ramas **producción preproducción**, **test** y **desarrollo**. Y si no era así, era algo parecido.

Desarrollo Tradicional

De esta forma, los desarrolladores integrarían su código con la rama de **desarrollo** para posteriormente, cuando se acerque la entrega de la nueva funcionalidad, se lleven los cambios que se quiera probar a la rama de **test**.

Una vez los cambios han sido probados de forma más o menos automática éstos se promocionan a la rama de preproducción, donde a menudo se suelen realizar test funcionales manuales.

Desarrollo Tradicional

Finalmente llegada la fecha de la entrega los cambios se llevan a la rama de producción. Todo este proceso suele estar supervisado, o incluso llevado a cabo por un **Release Manager**.

En el desarrollo tradicional lo habitual es que haya de una a tres (o cuatro) subidas a producción o *releases* al año.

Desarrollo Tradicional

Ventajas

- Se aumenta la funcionalidad muy rápidamente (ojo, no quiere decir que haya una buena calidad)
- Solamente una persona se preocupa de las *releases* a producción, dejando a los desarrolladores que se centren exclusivamente en sus tareas.

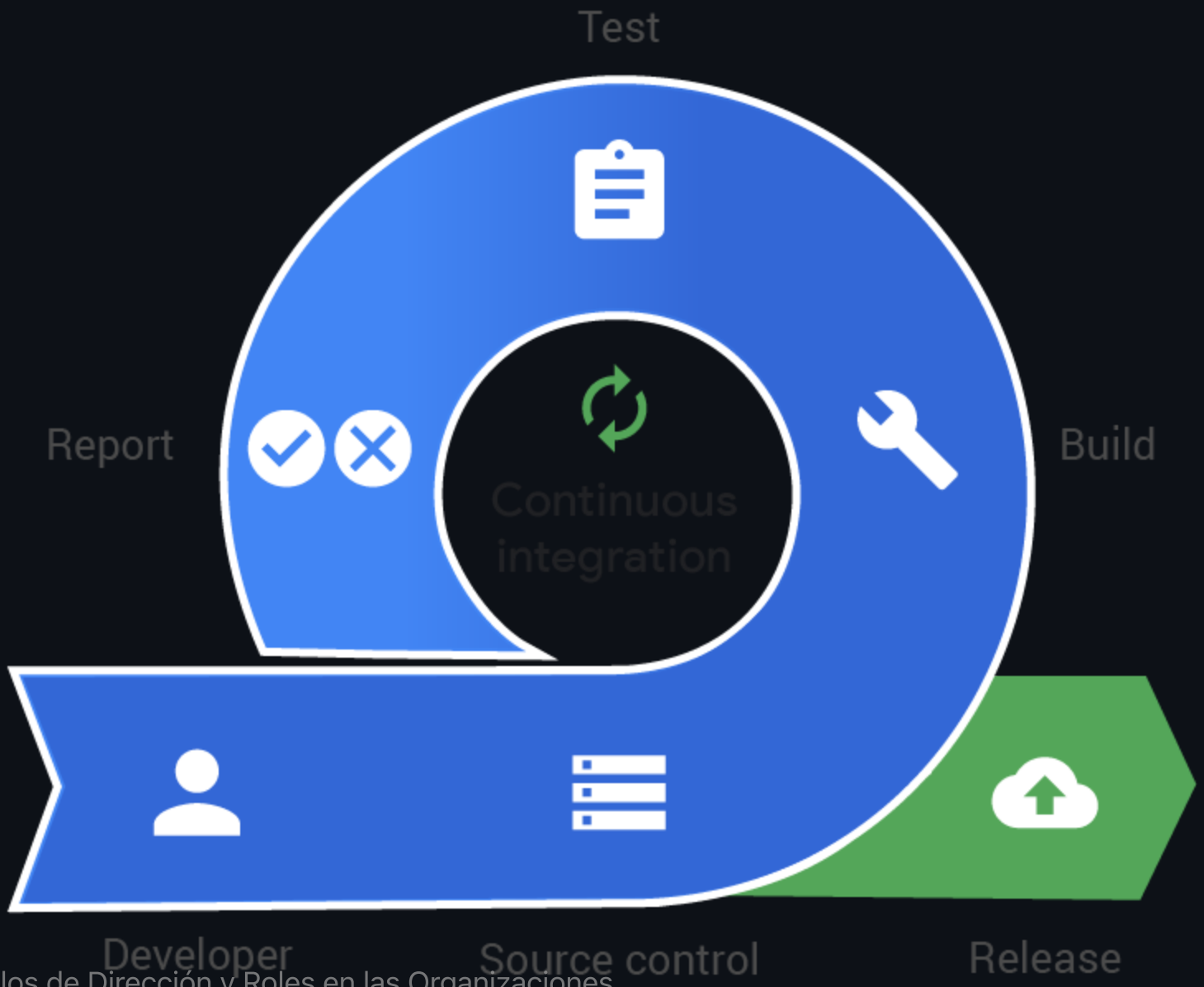
Desarrollo Tradicional

Desventajas

- Se hace dificil probar tanta funcionalidad nueva en poco tiempo, justo antes de la *release*
- Los programadores pierden la noción de la evolución del proyecto a nivel global.
- Los *merges* pueden llegar a ser muy complejos.
- A menudo fallan desarrollos y apenas hay tiempo para arreglarlo

Todas estas desventajas se pueden denominar *Integration Hell*

Integración Continua



Implicaciones

Desarrollo de sistemas de test automáticos

Todos los desarrolladores nos equivocamos, cometemos muchos errores que en un gran porcentaje llegan a producción, por lo que es importante darnos cuenta del error lo antes posible

Por eso es importante tener una buena base de test automatizados, tanto unitarios como de integración. Es importante que siempre funcione el caso bueno o *Happy Path* por ejemplo mediante el uso de smoke test.

Desarrollo de sistemas de tests automáticos

Además, los nuevos cambios no solo deben funcionar, sino que deben asegurar la retrocompatibilidad. Por ejemplo, en el caso de que se esté modificando una API REST, esto se podría conseguir comprobando las claves y los tipos de los campos en los JSON devueltos por la API.

La metodología de software TDD encaja muy bien con un enfoque de integración continua orientado a la calidad del software.

Implicaciones

Test de seguridad automáticos

Para que el desarrollo sea lo más seguro posible es necesario realizar escáneres de seguridad tanto del código como de los artefactos complilados durante la integración continua. De está forma se podrían detectar librerías

Implicaciones

Compilación y generación de artefactos de forma automática

Al igual que ocurría con los test, una vez realizados es necesario generar o artefactos entregables, o todos aquellos componentes que sirvan para test posteriores, como pueden ser de integración con otros sistemas etc... Para ello en los repositorios se suele incluir un fichero ejecutable, por ejemplo un *build.sh* un *Makefile* o incluso un *Dockerfile* donde se encuentran las instrucciones para la compilación y generación de artefactos. Los *Makefile* a menudo también incluyen directivas para poder pasar los test, del tipo: `make build test`

Implicaciones

Pull Requests o Merge Requests

En el proceso de integración del código, es decir, cuando una funcionalidad realizada por un desarrollador en una rama del repositorio se quiere llevar a la rama *master* es conveniente realizar una *Pull Request*, es decir, una petición de *merge* del código donde un compañero del equipo tiene que que inspeccionar el código y comprobar que se han cumplido todos los requisitos de calidad (por ejemplo los estilos del lenguaje de programación) que los test (por ejemplo los unitarios) se han realizado correctamente etc... Estas normas suelen ser definidas por el equipo de desarrollo, junto al equipo de calidad, o DevOps.

Es importante que no se pueda hacer una subida forzosa (`-f`) de cambios a la rama master

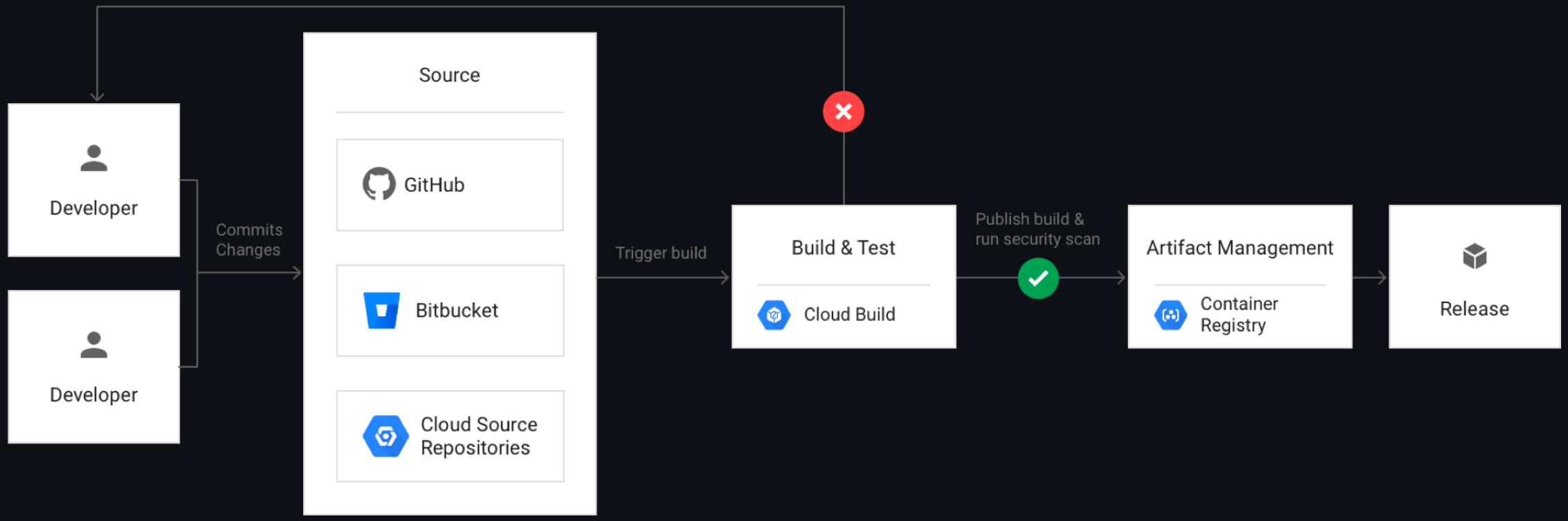
Automatización

Como se puede observar la automatización es uno de los pilares de la integración continua. Sin la automatización se perdería tanto tiempo o más que con los desarrollos de software más tradicionales.

Automatización

Flujo automático

Flujo de ejemplo propuesto en Google Cloud donde se agrega el código de distintos repositorios, para después generar un artefacto, pasar los test, incluyendo un análisis de seguridad y finalmente generar una release con esos artefactos en caso de que todo haya funcionado correctamente



Ventajas

Mejora la eficiencia y productividad durante el desarrollo

Después de cada cambio se puede observar como ha afectado el cambio al programa o sistema global. Aunque pueda parecer lento al principio, esto nos evitará el posterior *Integration Hell* donde se pierde mucho más tiempo. En todo momento se sabe cuál es el nivel.

Ventajas

Entregas con menor riesgo y de mayor calidad

Cada entrega tiene una funcionalidad más controlada, ya que ha pasado por todos los automatismos de test, de seguridad y de despliegue, además en los flujos de CI/CD maduros las entregas suelen ser muy continuadas, por lo que cada una de ellas lleva menos código, reduciendo así el riesgo.

Ventajas

Menos costes

Los procesos de automatización, junto con el menor riesgo y la mayor calidad de las entregas hacen que los desarrolladores se centren en lo que tienen que hacer, desarrollar nueva funcionalidad, que es lo que da valor a un producto de cara a un cliente. Esto se traduce en una reducción de los costes provenientes de tener a personal realizando tareas pesadas y repetitivas como los *Merge Request* o solventando Bugs, que en grandes productos pueden llegar a ser muy difíciles de solucionar.

Ventajas

Agile

La integración continua encaja muy bien con formas de trabajar de tipo *Agile*, de hecho es un pilar necesario para llevar a cabo un proyecto con estas metodologías. Por ejemplo en el marco de trabajo de *Scrum* las tareas, o historias de usuario que van a realizar los desarrolladores no pueden durar más de un sprint, propiciando la integración del código correspondiente a esas historias de usuario.

Despliegue Contínuo y Entrega continua

A partir de ahora nos referiremos a cualquiera de ellas o a CI/CD indistintamente.

La diferencia radicara en si nosotros tenemos una plataforma sobre la que desplegamos nuestro producto, o simplemente lo entregamos a un cliente externo. Por ejemplo, un programa open source habitualmente genera una entrega, que disponibiliza a los clientes en su apartado de *release* o downloads, en cambio AWS, realiza despliegues continuos de sus productos.

Despliegue Contínuo

Zero Downtime

Debido a la complejidad de algunas plataformas, lo recomendable es utilizar técnicas de despliegue que minimicen los posibles problemas del despligue, de forma que no haya una parada de servicio, por ejemplo en caso de que se haya colado un Bug, o que haya alguna incompatibilidad en la API que haga que a un cliente le deje de funcionar el servicio.
Debido a esto, los despliegues continuos van acompañados de sistemas de monitorización, y de *alerting* para que en caso de fallo, porder hacer un *rollback* a la versión anterior.

Zero Downtime

Algunas técnicas para despliegues continuos son:

- Empezar por la plataforma con menos clientes. Con un menor número de clientes menos crítico será el problema.
- De la misma forma si la plataforma está segregada por clientes, actualizar el servicio a los clientes que menos la utilizan.
- En caso de querer actualizar muchos servicios, no hacerlo simultaneamente, habiendo probado la compatibilidad
  de versiones del *path* de la actualización
- *Rolling Update*
- *Blue Green Deployment*

Despliegue Contínuo

Rolling Update

Mientras nuestro servicio está en producción, añadimos nuevas instancias del servicio progresivamente, de forma que un porcentaje de la carga lo van recibiendo las nuevas instancias. Por cada nueva instancia del servicio actualizado que se despliegue, se eliminará una del servicio antiguo, siempre teniendo en cuenta que no puede haber caída del servicio, por ejemplo haciendo *Drain* en el balanceador que envia carga a la instancia para posteriormente poder eliminarla. Así sucesivamente podremos hacer un despliegue continuo con *Zero Downtime*

Ejemplo de como lo hace Kubernetes: https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/

Despliegue Contínuo

Blue Green Deployment

Mientras nuestro servicio está en producción, levantamos el mismo número de instancias del servicio actualizado, una vez que nos hayamos asegurado de que las instancias actualizadas están en funcionamiento, cambiamos la configuración del balanceador (DNS, CDN, o cualquier otra pieza) para que toda la carga llegue a las nuevas instancias con el servicio actualizado.

Despliegue Contínuo

Continuous Delivery Foundation

CI/CD se ha convertido en un componente básico del desarrollo de software actual. Los equipos, desarrollan
flujos de trabajo adaptados a sus necesidades, junto con mecanismos y herramientas que proveen información
(feedback) para seguir optimizando estos procesos. La industria reconoce que el CI/CD es crítico para el
desarrollo del software, hasta el punto de que ha creado una fundación, *The Continuous Delivery Foundation*

Continuous Delivery Foundation

https://cd.foundation/about/

> The Continuous Delivery Foundation (CDF) serves as the vendor-neutral home of many of the fastest-growing projects for continuous integration/continuous delivery (CI/CD). It fosters vendor-neutral collaboration between the industry's top developers, end users, and vendors to further CI/CD best practices and industry specifications. Its mission is to grow and sustain projects that are part of the broad and growing continuous delivery ecosystem.

Despliegue Contínuo

Herramientas más comunes para el despliegue contínuo:

- GitHub Actions
- Gitlab CI/CD
- Jenkins (está cayendo en desuso)
- CircleCI
- Ansible

Nuevos Roles Profesionales: DevOps Engineers & SRE

DevOps Engineers

El objetivo: reducir los ciclos de desarrollo de software y de entrega de software de años o meses a horas y minutos

Funciones:

- Definición, desarrollo y mantenimiento de flujos de CI/CD
- Definición y Desarrollo de herraientas que devuelvan feedback del CI/CD para la mejora iterativa de los procesos
- Mantener y gestionar las herramientas, relacionadas con CI/CD
- Automatización de procesos

DevOps Engineer

Habilidades técnicas:

- Desarrollo con lenguajes de scripting
- Interpretar y desarrollar código fuente
- Gestión de infraestructura (Redes, servidores, BBDD)
- Familiarizado con empaquetado de software (.deb, .rpm, Docker)
- Familiarizado con herramientas de control de versiones
- Familiarizado con proveedores cloud (AWS, AZURE, GCP)
- Familiarizado con herramientas de test y análisis de seguridad
- Familiarizado con herramientas de monitorización

SRE

Muy relacionados también con los flujos de CI/CD debido a su conocimiento de operaciones, el SRE es un rol más relacionado con la arquitectura, que conoce la gestión de la producción, y que con sus conocimientos de desarrollo de software puede colaborar en la mejora de éstos de cara a una mejor productivización de los productos. Además por esta misma razón suelen definir y participar en los flujos de QA de los productos, partes tambien importantes en un sistema de CI/CD. Finalmente, también se encargará de diseñar y montar el sistema donde los DevOps Engineers podrán trabajar

Digamos que es un rol más maduro, con una mayor experiencia y que tiene una visión más arquitectónica de los sistemas. Se podría decir que es la evolución de un DevOps Engineer.

Nota

Todos los años tengo que cambiar los nombres de los roles, porque la industria cambia mucho, y los roles también.

Por ejemplo, hace unos años se hablaba de *Release Manager*, que era el encargado de llevar a producción los productos, y que a menudo era un programador senior, o un arquitecto de software. Ahora, con la llegada de la integración continua, y de la entrega continua, este rol ha desaparecido, apareciendo el de DevOps engineer.

Pero es que recientemente, con la llegada de la nube, y de la necesidad de tener sistemas que no fallen, ha aparecido el rol de SRE, que en muchos casos es el mismo que el de DevOps Engineer, pero con una mayor experiencia.

Herramientas

Las herramientas no son un fin. Los flujos y procesos sí

De cara a un buen CI/CD es necesario buscar las herramientas que más se ajusten a las necesidades del producto, proyecto o incluso organización. Una vez se tenga la lista de herramientas que satisfacen nuestros criterios técnicos de cara al CI/CD habrá que fijarse en los costes, tanto por licencia como de mantenimiento, en la integración con otras herramientas de la organización etc...

Herramientas

Puntos clave para la selección de herramientas

- ¿Puedo después de un *commit*, un *merge*, o una *pull request* lanzar un flujo de CI/CD?
- ¿Costes de licencia?
- ¿Costes de mantenimiento?
- ¿Ayuda a mejorar la calidad de las aplicaciones?
- ¿Ayuda a mejorar la seguridad de las aplicaciones?
- Trazabilidad: desde la historia de usuario, o *issue* hasta la release
- ¿Me ayuda a saber lo que tengo desplegado en producción?

Herramientas

Ejemplos

Control de versiones:

- Git
- Gitlab
- Github
- Bitbucket
- Aws

Herramientas

Pipelines

- Gitlab
- Jenkins
- AWS
- GitHub Actions
- CircleCI
- Ansible

Herramientas

Test de rendimiento

- Robot
- Gatling
- Locust

Herramientas

Métricas, Logs y su visualización

- Filebeat
- Metricbeat
- Elasticsearch
- Kibana
- Grafana
- Cloudwatch

Herramientas

Entornos de despliegue en nube

- AWS
- Azure
- Google Cloud

Herramientas

Simulación de Entornos / Nube privada

- Openstack
- Nomad
- Kubernetes
- ECS

Almacenamiento artefactos

- JFrog
- Artifactory
- Docker Registry (y GitLab Registry, GitHub, AWS, Azure...)

Lecturas recomendadas

- https://www.slideshare.net/jallspaw/10-deploys-per-day-dev-and-ops-cooperation-at-flickr
- https://circleci.com/blog/why-continuous-integration-is-key-to-stepping-up-deployment-frequency/
- https://circleci.com/blog/page/3/
- https://circleci.com/blog/how-bolt-optimized-their-ci-pipeline-to-reduce-their-test-run-time-by-over-3x/